© 2014 by William Ernest Mansky. All rights reserved.

SPECIFYING AND VERIFYING PROGRAM TRANSFORMATIONS WITH PTRANS

BY

WILLIAM ERNEST MANSKY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Elsa L. Gunter, Chair and Director of Research
Associate Professor Grigore Roşu
Professor Vikram Adve
Associate Professor Sara Kalvala, University of Warwick

Abstract

Software developers, compiler designers, and formal methods researchers all stand to benefit from improved tools for compiler design and verification. Program correctness for compiled languages depends fundamentally on compiler correctness, and compiler optimizations are usually not formally verified due to the effort involved. This is particularly true for optimizations on parallel programs, which are often more difficult to specify correctly and to verify than their sequential counterparts, especially in the presence of relaxed memory models. In this thesis, we outline a Verification Framework for Optimizations and Program Transformations, designed to facilitate stating and reasoning about compiler optimizations and transformations on parallel programs. Most verified compilation projects focus on a single intermediate language and a small number of input and output languages, later adding new targets as extensions; our framework, on the other hand, is designed with language-independence as a first principle, and we seek to generalize and reuse as much as possible across multiple target languages. Our framework makes use of the novel PTRANS transformation specification language, in which program transformations are expressed as rewrites on control flow graphs with temporal logic side conditions. The syntax of PTRANS allows cleaner, more proof-amenable specification of program optimizations. PTRANS has two sets of semantics: an abstract semantics for verification, and an *executable* semantics that allows specifications to act as prototypes for the optimizations themselves, so that candidate optimizations can be tested and refined before going on to formally verify them or include them in a compiler. We address the problems of parallelism head-on by developing a generic framework for memory models in VeriF-OPT, and present a method of importing external analyses such as alias analysis to overcome potential limitations of temporal logic. Finally, we demonstrate the use of the framework by prototyping, testing, and verifying the correctness of several variants of redundant store elimination in two markedly different intermediate languages.

To my parents, Art and Shelley Mansky, with love and gratitude

Acknowledgments

I owe a debt of gratitude to Elsa Gunter, whose unwavering support these past six years has been a constant aid. She constantly pushed me forward, reminded me of the big picture, and kept me convinced that the work we were doing was worthwhile. Her advice has been my starting point, for teaching, research, and surviving grad school. Whatever I accomplish in the future will be in large part due to what she's taught me. I can't thank her enough.

I've benefited greatly from being able to discuss ideas with my colleagues, and in particular with Dennis Griffith, who's always been happy to work through the details of a new idea. The fact that he's been a good friend on top of that is more than I could ask. Carl Evans, Brandon Moore, and Liyi Li also deserve my thanks, for being willing to listen to my complaints, offer suggestions, and help me keep moving forward. Susannah Johnson has been my strongest supporter, and never let me stop believing in myself.

I'd be remiss not to mention my first users, Tod Middlebrook, Grant Czajkowski, Kay Byun, Elizabeth Kelly, and Matt Cooper, who threw themselves enthusiastically into my fledgling project. Thanks to their continued effort, the framework here described is on the road from my personal project to a real-world system that could benefit the entire field.

I'd like to thank my committee: Grigore Roşu, who reminded me that my work needed to be usable as well as theoretically interesting; Vikram Adve, whose perspective on real-world languages and compilers (and, of course, especially LLVM) was unparalleled; and Sara Kalvala, who put me on this route five years ago and stayed invested and engaged from an ocean away. It was only with their help that my scattered thoughts were transformed into a coherent thesis.

Finally, I'd like to thank my parents, who have followed me through every twist and turn, and read every one of my papers whether they understood it or not. I couldn't have made it this far without them.

This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Table of Contents

Chapter 1 Introduction 1
1.1 Research Contributions 4

Chapter 2 Background and Related Work 5
2.1 Computation Tree Logic 5
2.2 The TRANS Approach 7
2.3 Compiler Correctness 9

Chapter 3 The PTRANS Specification Language 11
3.1 PTRANS: Adapting TRANS to Parallel Programs 11
3.1.1 The Parameterized Approach 12
3.2 Parallel Control Flow Graphs 13
3.3 Temporal Logic on tCFGs 14
3.3.1 The Problem of Next 16
3.4 The Semantics of PTRANS 18
3.5 Integrating External Analyses 20
3.5.1 Mutual Exclusion in CTL 21

Chapter 4 Intermediate Languages for PTRANS 25
4.1 Compiler Intermediate Representations 25
4.2 Language Semantics on (t)CFGs 26
4.2.1 Single-Thread CFG Semantics 26
4.2.2 Memory Operations and Concurrency 28
4.3 Intermediate Language 1: MiniLLVM 29
4.3.1 Syntax 29
4.3.2 Instruction Types and CFG Edge Labels 30
4.3.3 Semantics 31
4.4 Intermediate Language 2: GraphBIL 33
4.4.1 Baby IL 33
4.4.2 From BIL to GraphBIL: Syntax 34
4.4.3 Instruction Types and CFG Edge Labels 34
4.4.4 From BIL to GraphBIL: Semantics 35
4.5 Modeling Memory Models 38

Chapter 5 Executable Semantics and Testing 43
5.1 From Specification Language to Design Tool 43
5.2 Exploration via Executable Semantics 44
5.2.1 CTL Model Finding 44
5.2.2 Executable Semantics for Strategies 48
5.3 Designing and Prototyping Optimizations in PTRANS 49
5.4 Implementation 52
5.5 Usability 54

Chapter 6 Optimization Verification 55
6.1 Defining Correctness 55
6.1.1 PTRANS and Simulation 56
6.2 Verifying a MiniLLVM Optimization 59
6.2.1 Specifying the Optimization 60
6.2.2 Verification 62
6.3 Verifying a GraphBIL Optimization 67
6.3.1 Specifying the Optimization 68
6.3.2 Verification 68
6.4 Factoring Out Common Elements 69

Chapter 7 Conclusions 71
7.1 Discussion 72
7.1.1 Comparison with Related Work 74
7.2 Ongoing and Future Work 75
7.2.1 The Big Picture 76
7.3 Summary 78

Appendix A Code Listing 79
A.1 Isabelle Proofs 79
A.2 Executable Semantics in F# 364

References 394

Chapter 1

Introduction

Of the various phases of a modern compiler, optimization is generally considered to be the most complex. At the point of optimization, programs have usually been parsed and transformed into some internal representation – often a control flow graph, in which nodes are labeled with instructions in some intermediate language and edges represent jumps in control flow. Before generating the low-level code that actually executes on a machine, the compiler attempts to rearrange the graph to improve its time and memory performance, without changing the behavior of the program in ways that would be considered undesirable. Optimizations are often stated as complex algorithms on program code, with only informal justifications of correctness based on an intuitive understanding of program semantics. While the transformations involved may be simple, the conditions under which they are safe to apply, which often rely on extensive program analysis, are easily misstated. In practice, even widely used compilers such as GCC sometimes transform code incorrectly [78], and some of these bugs have been shown to result from mishandling concurrency [11]. Insufficiently analyzed optimizations may result in unreliable execution of parallel code; compiler writers may even end up having to limit the scope and complexity of the optimizations they develop, in the absence of a method to demonstrate the safety of parallel optimizations.

The problem of compiler correctness can be generally stated as follows. Suppose we have a source program in a high-level language, such as C or Java, that has some desirable property φ. This program is transformed by a compiler into a target program in a low-level language, such as machine code for some specific architecture. Does the low-level program still have the property φ? In practice, φ might be a *correctness* property, asserting that the program produces a particular correct result, or a *safety* property, asserting that the program avoids some undesirable outcome, such as a segmentation fault or a race condition.

To make the problem more concrete, consider the simple transformation shown in Figure 1.1. This sort of optimization is near-universal in single-threaded code: if we know in advance the value that will be read by the load operation (by checking that the value is not modified between the store and the following load), we can remove the time-consuming memory operation and replace it with the known value. However, this simple optimization immediately risks introducing an error in the presence of concurrency. If p is the location of a

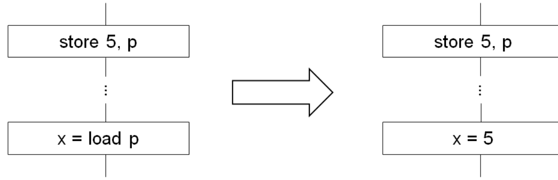


Figure 1.1: An example optimization

shared resource, then the value at p might be changed by the time we reach the load instruction; it might even be *impossible* for the value 5 to be read into x, as in Figure 1.2, due to synchronization between threads that guarantees that another value will be written first. In this case, applying our sequential optimization as-is to a concurrent program can introduce behaviors that were impossible in the original program, resulting in incorrect compilation. Coming up with a modification of this transformation that preserves correctness in the concurrent case is a non-trivial problem, and as we will see, the required modification may even depend on our implementation of concurrency.

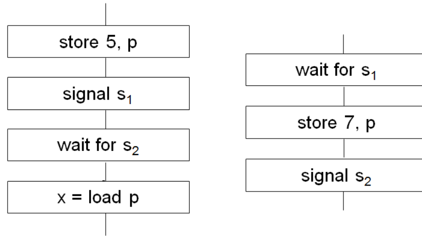


Figure 1.2: A concurrent program that causes problems for a sequential optimization

One well-established approach to correct compilation is translation validation, in which each source program is compared with the corresponding low-level output of the compiler, and checked to ensure that the translation preserves the property φ [65]. This approach treats the compiler as a black box, producing unreliable output that must be verified every time. An alternative approach is to take on the compiler directly, using formal tools to show that the compiler will produce correct output not just in certain instances but in general. Perhaps the greatest success of this second approach has been the CompCert project [39, 41], a compiler for a subset of C written entirely in the Coq proof assistant [8], and formally proven to produce machine code that is semantically equivalent to its input. Here we take this second, "white-box" approach: following in the footsteps of CompCert, we propose a general method of specifying the desired behavior of compiler transformations, aimed at verifying not just the output of the compiler but (portions of) the compiler itself.

In order to accomplish this, we must address several significant sub-problems. For any compiler verification effort, we must have a formal semantics for the language(s) to be compiled, allowing us to make

statements about the behavior of input and output programs without relying on possibly faulty compilation. We must then have some notion of equivalence or faithfulness of the compiled output to the input, so that we can state what it means for the compilation to be correct. If we are to take the white-box verification approach, we must have a formal semantics for the compilation process itself as well, so that we can reason about the effects of transformations on programs in general rather than analyzing individual cases. We must have a method of reasoning about the various components involved; that is, the semantics we have must not only be formal, but formal in a way accessible to our method of proving theorems (e.g., a proof assistant such as Coq [8] or Isabelle [62]).

Furthermore, we would like our framework to be *usable*; even if it takes a theorem-proving specialist to verify an optimization, it should be possible for compiler designers without formal methods training to provide the specification to be verified, and to demonstrate to their satisfaction that the specification accurately reflects the intended transformation. Especially when writing new optimizations for parallel code, we should be able to do so in an *exploratory* fashion, trying out variations on an approach and observing their effects on sample programs to be optimized. We aim to accomplish this by making our formal specifications *executable*; the specifications verified are not only abstract descriptions of desirable behavior, but also executable prototype optimizations that can be run on real code. In a production compiler we would likely want to re-implement the optimization for efficiency, but the specification prototype gives a quick way to test and explore the behavior of a candidate optimiation. Furthermore, the executable semantics can be shown to be faithful to the verification semantics, guaranteeing that the optimizations we test are the optimizations we verify. Giving executable semantics to optimization specifications considerably improves the usability of any compiler verification effort.

The VeriF-OPT project aims to address all of these facets of the correctness problem. The overall goal is a general language-independent method of specifying the desired behavior of compiler optimizations and verifying the correctness of those specifications. The process is supported by a framework designed for clean and modular specification of compiler transformations, and backed by formal semantics for both the compiler specification language and the target language of the compiler. The framework leverages existing formal methods tools, in particular the proof assistant Isabelle [62], to allow compiler designers to formally prove the properties of their specifications. The language-independent approach allows us to reuse proof components across optimizations and target languages, so that we build up a library of general facts about the specification language as we construct our proofs. The executable semantics allows the specification language to also be used to design and test optimizations, creating a direct link between compiler construction and compiler verification.

1.1 Research Contributions

We present the following original research contributions, each of which serves as a component of a novel specification and verification framework for compiler optimizations:

- PTRANS, a language-independent specification language for optimizations and program transformations on parallel programs
- A methodology for giving concurrency- and memory-model-aware operational semantics to programs represented as control flow graphs, either based on existing semantics or from scratch
- Formal semantics for two low-level languages, MiniLLVM and GraphBIL, that represent two opposite points in the spectrum of compiler intermediate representations
- An executable semantics and interpreter for PTRANS, allowing it to be used to design, prototype, and test candidate optimizations
- A verification technique for optimizations based on lifting a correct simulation relation for a single modified thread to a correct simulation relation for a multithreaded program
- Two simple verified optimizations for MiniLLVM or GraphBIL code under various memory models

Chapter 2

Background and Related Work

2.1 Computation Tree Logic

Temporal logics, used to express and verify properties over paths in transition systems, have long been used to verify properties of programs. In contrast to traditional first-order and higher-order logics, temporal logics have a built-in notion of progress through time, and have "next" and "until" as first-class concepts. The two most commonly used temporal logics are LTL [64], which is used to express properties on single paths, and CTL [15], which is used to express properties on sets of paths, viewed as trees that branch over time. In this section, we will review CTL and some of its relevant variants.

CTL (Computation Tree Logic) expresses properties on states in branching trees of computation. CTL formulae are constructed from a set of atomic propositions p according to the following grammar:

$$\varphi ::= \mathsf{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid AX\varphi \mid EX\varphi \mid A\,\varphi\,\mathcal{U}\,\varphi \mid E\,\varphi\,\mathcal{U}\,\varphi$$

The temporal quantifiers break down along two axes: A vs. E, where A-quantifiers quantify over all paths forward from the current state and E-quantifiers quantify over some such path, and X vs. $\mathcal{U}$, where X specifies the state immediately following the current state and $\mathcal{U}$ asserts that some property holds at all points along a path until the first point at which some other property holds. The derived operators AF, EF, AG, and EG are also often used:

$$AF\ \varphi \triangleq A\ \mathsf{true}\,\mathcal{U}\ \varphi \qquad EF\ \varphi \triangleq E\ \mathsf{true}\,\mathcal{U}\ \varphi \qquad AG\ \varphi \triangleq \neg EF\ \neg\varphi \qquad EG\ \varphi \triangleq \neg AF\ \neg\varphi$$

Intuitively, $AF\ \varphi$ (likewise EF) expresses the property that along all (some) paths, *eventually* (at some point now or in the future) φ will hold, while $AG\ \varphi$ (likewise EG) expresses the property that along all (some) paths φ will *always* hold.

Formally, the satisfaction of a CTL formula is defined with respect to a labeled transition system $\mathcal{C}$ and a state q in that transition system. We write $\mathsf{Paths}(\mathcal{C}, q)$ to denote the set of (non-branching) paths forward

from q in $\mathcal{G}$, where a path from q is simply a sequence of states $q_0\ q_1$ ... such that $q_0 = q$ and for all i, there is some label ℓ such that (q_i, ℓ, q_{i+1}) is a valid transition in $\mathcal{C}$. (We write λ_i to indicate the ith element of a path λ.) Then the CTL satisfaction relation $\mathcal{C}, q \models \varphi$ is defined as follows:

- $\mathcal{C}, q \models \mathsf{true}$
- $\mathcal{C}, q \models p$ if p holds at q
- $\mathcal{C}, q \models \varphi \wedge \psi$ if $\mathcal{C}, q \models \varphi$ and $\mathcal{C}, q \models \psi$
- $\mathcal{C}, q \models \neg\varphi$ if it is not the case that $\mathcal{C}, q \models \varphi$
- $\mathcal{C}, q \models AX\varphi$ if $\forall \lambda \in \mathsf{Paths}(\mathcal{C}, q).\ \mathcal{C}, \lambda_1 \models \varphi$
- $\mathcal{C}, q \models EX\varphi$ if $\exists \lambda \in \mathsf{Paths}(\mathcal{C}, q).\ \mathcal{C}, \lambda_1 \models \varphi$
- $\mathcal{C}, q \models A\,\varphi\ \mathcal{U}\ \psi$ if $\forall \lambda \in \mathsf{Paths}(\mathcal{C}, q).\ \exists i.\ \mathcal{C}, \lambda_i \models \psi$ and $\forall j < i.\ \mathcal{C}, \lambda_j \models \varphi$
- $\mathcal{C}, q \models E\,\varphi\ \mathcal{U}\ \psi$ if $\exists \lambda \in \mathsf{Paths}(\mathcal{C}, q).\ \exists i.\ \mathcal{C}, \lambda_i \models \psi$ and $\forall j < i.\ \mathcal{C}, \lambda_j \models \varphi$

An important property of a logic on transition systems is its *model checking complexity*, the computational complexity of determining whether a given formula holds on a given transition system. Clarke et al. [15] gave an algorithm for model-checking CTL in $O(|\varphi||\mathcal{C}|)$ time, where $|\varphi|$ is the number of subformulae of φ and $\mathcal{C}$ is the sum of the number of states and the number of transitions of $\mathcal{C}$, making CTL model checking quite efficient (as compared to, for instance, LTL model checking, which is *PSPACE*-complete).

Various extensions of CTL have been proposed to check more complex temporal properties of systems. The two most relevant extensions to our work are past-time CTL and first-order CTL; we will review the semantics of each here. Since ordinary CTL quantifies only over paths forward from a given state, it is natural to add quantifiers over paths backwards from a state as well. Finite and infinite pasts, straight-line and branching pasts, strict (i.e., not including the present) pasts and non-strict pasts have all been considered. Our formulation of past-time CTL is closest to variants such as the SUTL of Ramakrishna et al. [66], in that it treats past-time operators as symmetric to future-time operators:

$$\varphi ::= \ldots \mid A\,\varphi\ \mathcal{B}\ \varphi \mid E\,\varphi\ \mathcal{B}\ \varphi$$

We use $\mathsf{RPaths}(\mathcal{C}, q)$ to denote the set of paths backward from q in $\mathcal{C}$, where we restrict our backward paths to those that eventually reach the (or a) start state of the transition system. The semantics of the past-time temporal operators is given by:

- $\mathcal{C}, q \models A\,\varphi\,\mathcal{B}\,\psi$ if $\forall\lambda \in \mathsf{RPaths}(\mathcal{C}, q).\ \exists i.\ \mathcal{C}, \lambda_i \models \psi$ and $\forall j < i.\ \mathcal{C}, \lambda_j \models \varphi$

- $\mathcal{C}, q \models E\,\varphi\,\mathcal{B}\,\psi$ if $\exists\lambda \in \mathsf{RPaths}(\mathcal{C}, q).\ \exists i.\ \mathcal{C}, \lambda_i \models \psi$ and $\forall j < i.\ \mathcal{C}, \lambda_j \models \varphi$

We also follow Ramakrishna et al. in dropping "next" and "previous" operators when dealing with concurrent systems, since the meaning of "next" is unclear when we have multiple threads, each of which may be executing on its own timeline.

A less well-studied variant is the addition of first-order quantifiers to CTL. In this variant, called First-Order CTL (FOCTL) by Bohn et al. [12], the atomic propositions p are in fact terms that may contain variables (or, equivalently, predicates that take arguments), and temporal formulae may also quantify over these variables. There are several equivalent approaches to specifying the semantics of FOCTL; the one we present here adds an additional argument to the satisfaction relation, a substitution σ that maps variables to concrete values (the domain of values is determined by the set of atomic propositions). Formally:

$$\varphi ::= \ldots \mid \exists x.\ \varphi$$

- $\mathcal{C}, \sigma, q \models p$ if $\sigma(p)$ holds at q

- $\mathcal{C}, \sigma, q \models \exists x.\ \varphi$ if $\exists o.\ \mathcal{C}, \sigma(x \mapsto o), q \models \varphi$

and all other semantic rules carry σ through without using or changing it.

2.2 The TRANS Approach

Lacey et al. [32] proposed a novel method of specifying optimizations using temporal logic. Rather than modeling the behavior of a program as an FSA, they evaluated CTL formulae directly on paths through the control flow graph (CFG) of the program. Compiler optimizations can then be expressed as transformations on control flow graphs conditioned on the satisfaction of CTL formulae over the graphs. This allows a more modular formulation of many optimizations than the traditional algorithms, reducing the amount of context that needs to be drawn into the proof of each step of optimization. This approach was put into practice in the TRANS language of Kalvala et al. [28], which forms the starting point for our work. TRANS provides a uniform framework for expressing various types of optimizations. The use of temporal logic side conditions makes the assumptions of the transformation explicit in a formal sense, and narrows the gap between the theoretical semantics of the optimization and its actual implementation.

An earlier success in using the temporal-logic approach to optimization verification is the Cobalt speci-

fication system [37, 38]. Cobalt optimizations have structures of the form

$$\psi_1 \text{ followed by } \psi_2 \text{ until } s \Rightarrow s' \text{ with witness } \mathcal{P}$$

where ψ_1 and ψ_2 are drawn from a language of state predicates. This optimization finds some node n such that (1) there is a path from the start of the CFG to a node n_0 at which ψ_1 holds, (2) $A\ \psi_2\ \mathcal{U}\ n$ holds starting immediately after n_0, and (3) s is a pattern describing the instruction at n. If such a node n can be found, the instruction at n is replaced with s'. Each component of the optimization may contain *metavariables* that are instantiated in the process of evaluating the condition: for instance, the transformation

$$stmt(Y := C) \text{ followed by } \neg def(Y) \text{ until } X := Y \Rightarrow X := C$$

finds a node n such that (1) there is a node n_0 labeled with an assignment statement, (2) the LHS of the assignment at n_0 (Y) is not redefined until the node n is reached, and (3) the instruction at n is an assignment whose RHS is Y. If such a node n is found, the RHS of the assignment at n is replaced with the RHS of the assignment at n_0, making this a form of constant propagation.

The witness element is not used in the semantics of the optimization, but instead provides an invariant for use in verification: the goal is to provide a predicate $\mathcal{P}$ on execution states such that (1) $\mathcal{P}$ always holds after executing a node at which ψ_1 holds, (2) $\mathcal{P}$ is preserved when executing nodes such that ψ_2 holds, and (3) if $\mathcal{P}$ holds at a node n in an execution state η, then executing the statement s at n under η will produce the same result state as executing s'. If such a $\mathcal{P}$ is given, then the optimization is guaranteed to be sound, in the sense that every execution state reached by a transformed program is also reachable in the original program and vice versa. These proof obligations are automatically set up and discharged by the Simplify theorem prover [59]. The system can also express backward properties using a past-time variant of the above formula, as well as "pure analyses" that, rather than transforming the code, place labels on nodes for use in future transformations.

The strengths and limitations of Cobalt both stem from its use of a fully automatic theorem prover. The only contribution of the optimization writer to the verification process is to provide a suitable witness. If the witness is sufficient, the optimization is verified in a matter of seconds; if not, the user must use the output of the theorem prover to figure out what went wrong, and continue to provide witnesses until one proves sufficient. The automation also comes at the cost of expressiveness: Cobalt is limited to a single until-formula per optimization, and can only describe optimizations that modify the instruction at a node (rather than adding/removing edges or nodes). In our work, we aim to provide the user with the full expressiveness

of the original TRANS system, which means we must accept the cost of interactive theorem proving, and make an effort wherever possible to minimize the proof burden on the user.

2.3 Compiler Correctness

Compiler verification efforts date back at least to a 1973 paper by Morris [57]. Early efforts include the construction of an interpreter for the Piton language in the Boyer-Moore theorem prover [55, 56] and the verification of a compiler for a subset of the Gypsy language [79]. Also notable is Elsa Gunter's work in formalizing the syntax and semantics of SML, as well as its module system, in HOL-90 (the predecessor to Isabelle) [27, 48].

The proof assistants Coq [9] and Isabelle [62] have used as platforms for various significant formal verification efforts, including several in the area of verified compilation. Early Isabelle projects included an interpreter for a small functional language verified aginst its denotational semantics [13] and a verified lexer [60]. More recent developments have included verification of a code generation algorithm from SSA form [10], a bytecode verifier for the Java Virtual Machine (JVM) [29], and compilers for substantial subsets of C [35, 36] and Java [73]. Gesellensetter et al. [21] have also used a translation-validation approach in Isabelle to find bugs in GCC. The Jinja project has formalized a significant subset of Java, including its concurrency model, in Isabelle [30, 47]. Coq projects have included verification of the JavaCard smartcard platform [6], development of a formally specified and executable JVM [5], verification of an algorithm for sequentializing parallel assignments [68], verification of CPS transformations for functional languages [16], and creation of a verified compiler for simply-typed lambda calculus [14].

Perhaps the most impressive achievement in compiler verification to date is the CompCert project, culminating in a compiler from a subset of C to PowerPC machine code that has been written and fully verified in Coq [39, 40, 41]. Though complete formal compiler verification was previously considered near-impossible by the formal analysis community (see for instance [65]), the CompCert project has provided a powerful counterexample to this argument. While it does not cover the entire C specification, the compiler does include a range of real-world optimizations, and introduces a general method for proving correctness of simple optimizations involving dataflow analysis. Further work on the project includes a framework for adding new optimizations [74] and an extension for garbage collection [52], as well as a C-like memory model for verifying pointer-based programs [42].

The work of CompCert continues with CompCertTSO, a parallel extension of CompCert following the TSO memory model. Ŝevčik et al. [71] have lifted the proofs of correctness for several of CompCert's

optimizations, including constant propagation and common subexpression elimination, to the parallel case. As part of this effort, they have verified a fence elimination optimization, the first concurrency-specific optimization to be verified in the CompCert framework [76]. Sewell et al. [61] continue to explore and develop formalizations for relaxed memory models for use in optimization verification, and our understanding of TSO and PSO owes much to their work.

The Vellvm project [80] has created a framework for reasoning about the LLVM intermediate language. Its formalization of the LLVM IL has a considerably more sophisticated treatment of memory layout than MiniLLVM, but does not take concurrency or concurrent memory models into account. Zhao et al. [81] have used the framework to verify a complex LLVM optimization, specifically taking advantage of the assumptions of static single assignment form.

In order to verify properties of concurrent systems, we need representations of those systems that can serve as a basis for analysis. Following in the footsteps of TRANS [28], we use control flow graphs [2] as our basic program representation, and use these graphs to check syntactic properties of the programs to be verified. Various other approaches exist to representing concurrent programs, many of them focusing on describing their dynamic behavior, including process algebras such as the π-calculus [54], the actor model [1], and statecharts [23], which are a graph-based refinement of the concept of transition systems.

Separation logic [67], an extension of Hoare logic for reasoning about pointers and memory, is a commonly used tool for proving properties of programs. Hobor et al. [26] extended separation logic with explicit memory sharing, locks, and dynamic thread creation, and used it to verify programs in the Cminor intermediate language used by CompCert. VST [4] is a project building on CompCert that seeks to link the results of static analysis tools and separation logic proofs on high-level programs to the behavior of compiled code, with a particular focus on concurrency. The single-thread analyses of VST so far rely on programs being *well-synchronized*, that is, on obtaining the proper locks before using shared resources. While separation logic and systems based on it do not directly address the problem of compiler correctness, they interact interestingly with it, raising the question of exactly which properties are preserved across the various stages of compilation.

Li and Slind [44] have also made efforts to construct fully verified compilers, for higher-order functional languages and later for a subset of C [43], using higher-order logic and term rewriting. Their work is built on HOL-4, a theorem prover closely related to Isabelle. By defining the semantics of their source and target languages in HOL-4 (or, in the former case, using languages built into the prover), they are able to take a uniform approach to proving semantic preservation, and their verification approach similarly proceeds by showing that each transformation rewrite is semantics-preserving.

Chapter 3

The PTRANS Specification Language

3.1 PTRANS: Adapting TRANS to Parallel Programs

In this chapter, we introduce PTRANS, the tranformation specification language at the core of our framework. The basic approach of the PTRANS specification language is that set out by Kalvala et al. in TRANS [28]: optimizations are specified as rewrites on program code in the form of control flow graphs, with side conditions given in temporal logic. Intuitively, the rewrite portion of an optimization expresses the particular transformation to be made, and the side condition characterizes the situations in which the optimization is allowed to be applied. In previous work, we formalized the syntax and semantics of TRANS for sequential programs in the Isabelle theorem prover [50]; this formalization serves as our starting point as we move into the parallel domain.

The basic units of rewriting in PTRANS, as in TRANS, are *actions*, atomic graph rewrites defined as follows:

$$A ::= \mathsf{add_edge}(n, m, \ell) \mid \mathsf{remove_edge}(n, m, \ell) \mid \mathsf{split_edge}(n, m, \ell, p) \mid \mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m$$

The actions add_edge and remove_edge add and remove (ℓ-labeled) edges between the specified nodes; split_edge splits an edge between two nodes, inserting a new node between them; and replace replaces the instruction at a given node with a sequence of instructions, adding new nodes to contain the instructions if necessary (or removes the node if the sequence is empty). Kalvala et al. have shown that a wide variety of common program transformations can be expressed using these basic rewrites. The arguments to the atomic actions represent nodes and instructions in the program graph, but may contain *metavariables* that are instantiated to program objects when the rewrites are applied. In the case where these rewrites are nonsensical (e.g., add_edge on a pair of nodes that do not exist in the graph), they fail to apply, halting the transformation and producing no result graphs.

A transformation in PTRANS is built out of conditional rewrites combined with *strategies*, defined as

follows:

$$T ::= A_1, ..., A_m \text{ if } \varphi \mid \text{MATCH } \varphi \text{ IN } T \mid T \text{ THEN } T \mid T \,\Box\, T \mid \text{APPLY_ALL } T$$

The transformation $A_1, ..., A_m$ if φ is the basic pairing of a rewrite with a CTL side condition (details of the side conditions in our parallel formulation will be given in Section 3.3). The expression MATCH φ IN T provides an additional side condition for a set of transformations, and also allows metavariables to be bound across multiple rewrites. The THEN and $\Box$ operators provide sequencing and (nondeterministic) choice respectively, and APPLY_ALL T recursively applies T wherever possible until it is no longer applicable to the graph under consideration.

Thus far, the syntax of PTRANS is identical to that of TRANS: action and transformation specifications can be applied straightforwardly to the parallel case. The major changes to the language appear in the program graph model, the semantics of transformations, and the definition of CTL formulae: in the remainder of this chapter, we will discuss these changes in detail.

3.1.1 The Parameterized Approach

The primary aim of the PTRANS framework is to serve as a tool and a starting point for the verification of new compiler optimizations and transformations. As such, one of its main design goals is generality: it should be possible to work with as many target languages and express as many kinds of transformations as possible. To this end, we *parameterize* by details of language and logic whenever possible. Rather than incorporate details of language structure or expected transformation patterns into the framework, we state a few simple axioms about the object in question and then reason generally. In the Isabelle implementation of the framework, this is accomplished through the use of polymorphic datatypes and *locales*, bundles of constants and assumptions that form the context for further reasoning. Locales provide an easy formalization for declaring axioms, writing proofs with those axioms as assumptions, and then later demonstrating that specific languages or structures satisfy those axioms, thus automatically specializing the general proofs to the specific languages or structures in question.

In order to parameterize by the target language wherever possible, we must clearly distinguish between the elements of a specification or a semantics that are part of PTRANS, and the elements that are provided by the target language. Thus far, we have seen that the syntax of actions and transformations is part of PTRANS, but the syntax of instruction patterns is provided by the target language. In the next section, we will see another parameter provided by the target language: the relationship between the instruction labels and the edges of a well-formed control flow graph. In this chapter, we will make a point of noting each place where we assume that some element is provided by the target language, and give our semantic functions for

PTRANS specifications independently of the semantics (and to the extent possible the syntax) of the target language.

3.2 Parallel Control Flow Graphs

The TRANS approach depends fundamentally on a notion of control flow graph (CFG). The atomic rewrites are rewrites on CFGs, and the CTL side conditions are evaluated on paths through CFGs. Thus, we require a parallel analogue to the CFG in order to extend the approach to parallel programs. The particular model used here, adapted from the work of Krinke [31], is the threaded control flow graph (tCFG). In our framework, a tCFG is simply a collection of non-intersecting CFGs, one for each thread in a program. This model combines simplicity and flexibility: concepts from single-threaded CFGs can be straightforwardly extended to tCFGs, and nothing prohibits a tCFG from adding or removing threads over the lifetime of a program as in a fork-join model (although languages with dynamic thread creation are outside the scope of this thesis). Formally:

Definition 1. *A* CFG *is a labeled directed graph* $(N, E, Start, Exit, L)$ *where* N *is a set of nodes,* $E \subseteq N \times T \times N$ *is a set of* T*-labeled edges (where* T *is given by the target language, but must contain the label* seq*),* $Start, Exit \in N$ *are the distinguished Start and Exit nodes of the graph, and* $L : N \rightarrow I$ *assigns a program instruction to each node, such that: Start has no incoming edges, Exit has no outgoing edges, and the outgoing edges of each node except Exit correspond properly to the instruction label at that node, where the required correspondence is determined by the target language. A* tCFG *is a collection of disjoint CFGs, one for each thread in the program being represented. If* $\mathcal{G}$ *is a tCFG and* t *is a thread, we write* $\mathcal{G}_t$ *for the CFG of* t *in* $\mathcal{G}$*.*

Here we see two parameters that must be provided by the target language: the domain of edge types (which must include the sequence edge type seq, but is otherwise unrestricted) and a correspondence between instruction labels and the types of outgoing edges. For instance, in most programming languages, an assignment statement should have only one outgoing edge, indicating the next instruction to be executed; a conditional branch statement, on the other hand, should have two outgoing edges, one clearly marked as belonging to each branch. Generalizing this correspondence as a parameter allows us to reason about CFGs and tCFGs independently of any particular programming language.

The revised semantics of TRANS rewrites [50] can be straightforwardly extended to tCFGs: since individual CFGs do not intersect, the nodes mentioned in each atomic action uniquely identify (at most) one thread on which to perform the rewrite. The semantics of an action A on a tCFG $\mathcal{G}$, then, is the set of

graphs produced by the following process:

1. Identify the thread t such that the nodes referenced by A are in $\mathcal{G}_t$.

2. Perform the transformation indicated by A (adding, removing, or splitting an edge, or replacing a node) on $\mathcal{G}_t$, leaving the rest of the graphs in $\mathcal{G}$ untouched.

If nodes from several different threads are mentioned in a single atomic action, the rewrite fails to apply, as in the case where the nodes mentioned do not exist in the graph. In Section 3.4, we will give the full formal semantics for PTRANS specifications, but first we must describe one last element of a PTRANS specification: the CTL side condition.

3.3 Temporal Logic on tCFGs

The side conditions in PTRANS are given in a variant of the branching-time temporal logic CTL. A CTL formula expresses a property over a (possibly infinite) tree of *states*, and at each branching point quantifies over the possible *paths* forward from that state. Thus, in order to evaluate CTL formulae on tCFGs, we must define the notions of state and path in a tCFG.

Definition 2. *A* state *q in a tCFG $\mathcal{G}$ is a vector of nodes, one in each of the constituent CFGs $\mathcal{G}_t$, representing a potential program point in the execution of the parallel program represented by $\mathcal{G}$. A sequence of states λ is a* path *through $\mathcal{G}$ starting from q iff $\lambda_0 = q$ and for each pair of consecutive states λ_i, λ_{i+1}, for each thread t, either*

1. *$\lambda_i(t) = \lambda_{i+1}(t)$, or*

2. *there is an edge from $\lambda_i(t)$ to $\lambda_{i+1}(t)$ in $\mathcal{G}_t$.*

Since CFGs are closed under the edge relation, we are guaranteed that $\lambda_i(t)$ is a node in $\mathcal{G}_t$ as long as $q(t)$ was a node in $\mathcal{G}_t$.

The statements of our side conditions can sometimes be simplified by considering backward as well as forward paths through a graph; reverse paths are defined similarly to forward paths:

Definition 3. *A sequence of states λ is a* reverse path *through $\mathcal{G}$ starting from q iff $\lambda_0 = q$ and for each pair of consecutive states λ_i, λ_{i+1}, for each thread t, either*

1. *$\lambda_i(t) = \lambda_{i+1}(t)$, or*

2. *there is an edge from $\lambda_{i+1}(t)$ to $\lambda_i(t)$ in $\mathcal{G}_t$.*

In addition, for each thread t, there must be some i such that $\lambda_i(t) = Start_t$.

This additional condition means that we consider only reverse paths that eventually reach the Start node of each CFG $\mathcal{G}_t$. This allows us to restrict our analysis to *reachable* states, those that can occur on some path from the initial state of the tCFG.

We can then give CTL formulae their standard interpretations on infinite paths (in fact, we use first-order CTL, as described in Section 2.1), and with the aid of a few useful atomic predicates, use them to state the necessary correctness conditions for various optimizations. The TRANS language [28] distinguished between "node conditions" evaluated at a particular node and "side conditions" evaluated over the entire graph, but with a suitable set of atomic predicates, we can make this distinction unnecessary and use only one category of CTL formulae; when we check a CTL formula on a tCFG, we check it beginning in the start state of the tCFG (that is, the state in which each component CFG is at its Start node). Given a set of atomic predicates p, a *side condition* for an action is of the form:

$$\varphi ::= \mathsf{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid A\ \varphi\ \mathcal{U}\ \varphi \mid E\ \varphi\ \mathcal{U}\ \varphi \mid A\ \varphi\ \mathcal{B}\ \varphi \mid E\ \varphi\ \mathcal{B}\ \varphi \mid \exists x.\ \varphi$$

The derived "finally" and "globally" operators EF, AF, EG, AG are defined from the $\mathcal{U}$ operators in the usual way. The first-order existential quantifier $\exists$ is used to quantify over metavariables in a formula; these metavariables may then appear in the atomic predicates of a formula, enhancing the expressive power of the side conditions. For instance, if we have a predicate $\mathsf{node}(n)$ that is interpreted as "the current node is n" and an equality predicate $=$, we can build the formula

$$\mathsf{node}(n) \wedge (\exists n'.\ n' \neq n \wedge E\ \mathsf{node}(n)\ \mathcal{U}\ \mathsf{node}(n'))$$

which is true if and only if n has a successor that is not n. Formulae of this sort are extremely useful in expressing properties over program graphs.

In general, the atomic predicates available for use in side conditions are another parameter provided by the target language under consideration, but certain atomic predicates are useful across almost every target language. We present here some simple, general predicates that can be interpreted in both of our (significantly different) target languages. These predicates break down into two types: those that depend on the state in which they are evaluated, and those that do not (i.e., those that check some global property of the tCFG under consideration). State-based predicates include:

- $\mathsf{node}_t(n)$, which is true of a state q when $q(t) = n$.

- $\mathsf{stmt}_t(p)$, which is true of a state q when the instruction label of $q(t)$ in $\mathcal{G}_t$ is p.

- $\mathsf{out}_t(ty, n')$, which is true of a state q when $q(t)$ has an edge out to n' with label ty in $\mathcal{G}_t$, and symmetrically $\mathsf{in}_t(ty, n')$.

State-independent predicates include:

- $\mathsf{is}(x, y)$, which is true when the metavariables x and y represent the same program object (number, node, instruction, etc.).

- $\mathsf{int_eq}(a, b)$, which is true when a and b are arithmetic expressions that statically evaluate to the same value (according to some reference semantics for arithmetic operations).

- $\mathsf{fresh}(e)$, which is true when e represents a program variable that appears nowhere in $\mathcal{G}$.

Note that all of these predicates are purely syntactic static properties of tCFGs. In general, PTRANS optimizations can be stated and performed independently of the semantics of the target language, so that PTRANS may serve as a design tool even in the absence of formal semantics for the target language. Of course, semantics will be required to reason about the correctness of optimizations.

We also provide several extended predicates that allow the integration of outside analyses into CTL conditions. These predicates are:

- $\mathsf{cannot_alias}_t(e, e')$, which is true of a state q when alias analysis can show that e and e' are not pointers to the same location in t at q.

- $\mathsf{in_critical}_t(e, x)$, which is true of a state q when mutex analysis can show that $q(t)$ is part of a critical section for e protecting the value of x.

- $\mathsf{protected}_t(e, x)$, which is true when mutex analysis can show that the value of x is only changed in critical sections for e.

Details about the semantics and implementation of these predicates can be found in Section 3.5.

3.3.1 The Problem of Next

As observed by Ramakrishna et al. [66] among others, the CTL "next" operators AX and EX are significantly less useful on concurrent programs than on sequential ones. In particular, from the perspective of any given thread in a concurrent execution, a path may "stutter" indefinitely: since different threads proceed independently and at different speeds, in any given transition within a path, the program point assigned to

a particular thread may remain constant, and paths may contain long sequences of states in which some or all component CFGs make no progress. This makes the traditional interpretation of the "next" operators AX, EX largely useless, since the next state at any given point in a path may be identical to the current state. While in the single-threaded case we could use the formula $EX\mathsf{stmt}(i)$ to express the condition that "the current node has a successor labeled with instruction i", $EX\mathsf{stmt}_t(i)$ holds in the parallel case if the current node itself is labeled with i and the path under consideration happens to stutter on the thread t. In general, we would like to be able to provide "stuttering-invariant next" operators, which assert that a given property holds as soon as a given thread takes a step away from the current node. Similar operators are found in the temporal logic of actions [33]: in that system, states assign values to variables, and we may concern ourselves only with steps in which the value of a certain variable is changed. We can construct stuttering-invariant next operators from the "until" operators as follows:

$$EX_t\ \varphi \triangleq \exists n.\ \mathsf{node}_t(n) \wedge (E\ \mathsf{node}_t(n)\ \mathcal{U}\ (\neg\mathsf{node}_t(n) \wedge \varphi))$$

$$AX_t\ \varphi \triangleq \exists n.\ \mathsf{node}_t(n) \wedge (A\ \mathsf{node}_t(n)\ \mathcal{W}\ (\neg\mathsf{node}_t(n) \wedge \varphi))$$

The past-time equivalents EP_t and AP_t can be constructed analogously. Note that AX uses the (defined) "weak until" operator $A\ \psi\ \mathcal{W}\ \varphi \triangleq \neg(E\ \neg\psi\ \mathcal{U}\ (\neg\varphi \wedge \neg\psi))$ in the place of $\mathcal{U}$. This reflects a tricky corner case in the A quantifier over stuttering paths: included in the set of all paths is the path that stutters indefinitely, never reaching the next node in t. Thus, rather than requiring that all paths eventually reach the next node (implied by the semantics of $\mathcal{U}$), we allow for the possibility that this node will never be reached (and t will remain at its current node forever). Note also that we cannot distinguish between a step in which a thread steps from its current node to the same node along a looping edge, and one in which the thread does not advance. In general, we expect that well-formed CFGs will not have looping edges (i.e., edges that start and end at the same node), since in most intermediate languages instructions do not pass control directly back to themselves.

We can also construct analogous next-operators that state properties on next/previous nodes along edges of a given type, using the out_t predicate in addition to node_t. For instance:

$$EX_{t,ty}\ \varphi \triangleq \exists n, n'.\ \mathsf{node}_t(n) \wedge \mathsf{out}_t(ty, n')\ \wedge (E\ \mathsf{node}_t(n)\ \mathcal{U}\ (\mathsf{node}_t(n') \wedge \varphi))$$

By omitting the conventional next operators AX and EX, and instead providing the stuttering-invariant EX_t and AX_t, we can usefully state properties on next (and previous) nodes in a given component CFG of

a tCFG. In this way, we recover the "next" operators of single-threaded TRANS using only $\mathcal{U}$ operators in PTRANS.

3.4 The Semantics of PTRANS

We are now ready to give the mathematical semantics of PTRANS, which we do by defining the set of graphs that can be produced by applying a PTRANS specification to a given input graph. The semantics of actions is defined by a function $[\![A]\!](\sigma, \mathcal{G})$ that takes an action, a substitution, and a tCFG and returns the tCFG that results when the action is performed (or fails if the action is impossible). The substitution assigns values to the metavariables in A, and will be used to transform instruction, node, and other patterns into concrete objects in the CFG. Note that since every action specifies at least one node and the nodes of CFGs in a tCFG are disjoint, each action implicitly specifies at most one CFG $\mathcal{G}_t$ on which to perform the action (if two nodes mentioned are in two different graphs, the action simply fails). Suppose we have $\mathcal{G}_t = (N_t, E_t, s_t, x_t, L_t)$; then the semantics of actions are then defined as follows:

$$[\![\mathsf{add_edge}(n, m, \ell)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t, E_t \cup \{(\sigma(n), \sigma(\ell), \sigma(m))\}, s_t, x_t, L_t))$$

$$[\![\mathsf{remove_edge}(n, m, \ell)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t, E_t - \{(\sigma(n), \sigma(\ell), \sigma(m))\}, s_t, x_t, L_t))$$

$$[\![\mathsf{replace}\ n\ \mathsf{with}\ ]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t - \{\sigma(n)\}, E_t - \{(a, \ell, b) \mid a = \sigma(n) \vee b = \sigma(n)\}, s_t, x_t, L_t))$$

$$[\![\mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t \cup \{n_2, ..., n_m\}, \{\mathsf{remap_succ}(\sigma(n), n_m, e) \mid e \in E\} \cup$$
$$\{(n_i, \mathsf{seq}, n_{i+1}) \mid 1 < i < m\}, s_t, x_t, L_t + (n_1 \mapsto \sigma(p_1), ..., n_m \mapsto \sigma(p_m)))$$

where $n_1 = \sigma(n)$ and $n_2, ..., n_m$ are new nodes not in $\mathcal{G}$, and $\mathsf{remap_succ}$ is defined below

$$[\![\mathsf{split_edge}(n, m, \ell, p)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t \cup \{n'\},$$
$$E_t - \{(\sigma(n), \sigma(\ell), \sigma(m))\} \cup \{(\sigma(n), \sigma(\ell), n'), (n', \mathsf{seq}, \sigma(m))\},$$
$$s_t, x_t, L_t + (n' \mapsto \sigma(p)))) \text{ where } n' \text{ is a new node not in } \mathcal{G}$$

In the replace action, we must not only introduce new seq edges between the added nodes (recall that every target language must provide seq as an edge type), but also move the outgoing edges of the initial node n_1 instead to be outgoing edges of the last added node n_m. To do this we use the auxiliary remap_succ function, defined as

$$\mathsf{remap_succ}(n, n', (a, \ell, b)) \triangleq \text{if } a = n \text{ then } (n', \ell, b) \text{ else } (a, \ell, b)$$

The semantics of a list of actions $A_1, ..., A_m$ is the composition of the semantic functions of the individual actions, i.e., the graph resulting from applying all of the actions in left-to-right order. While the majority of this definition is independent of the target language under consideration, note that we do require that some notion of substitution on instruction patterns $\sigma(p)$ be defined; in other words, the semantics of actions are parameterized by the definition of substitution provided by the target language. While we generally expect that this will be a straightforward process of replacing metavariables with values in a term, some languages may require more complex substitution functions (e.g., replacement of a metavariable within a list with a list segment to be integrated into the list).

The semantics of strategies is defined by a function $[\![T]\!](\tau, \mathcal{G})$ that takes a strategy expression (often called simply a *transformation*), a partial substitution, and a tCFG and returns the set of tCFGs that can be produced by the transformation. In order to give semantics to the APPLY_ALL strategy, we must define the result of applying a transformation function some finite (but unbounded) number of times:

$$\frac{}{\mathsf{apply_some}(T, \tau, G, G)} \qquad \frac{G' \in T(\tau, G) \quad \mathsf{apply_some}(T, \tau, G', G'')}{\mathsf{apply_some}(T, \tau, G, G'')}$$

Then the semantics of strategies are defined as follows:

$$[\![A_1, ..., A_m \text{ if } \varphi]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \exists \sigma.\ \sigma|_{\mathrm{dom}(\tau)} = \tau \wedge \mathcal{G}, \sigma \models \varphi \wedge \mathcal{G}' = [\![A_1, ..., A_m]\!](\sigma, \mathcal{G})\}$$

$$[\![\text{MATCH } \varphi \text{ IN } T]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \exists \sigma.\ \sigma|_{\mathrm{dom}(\tau)} = \tau \wedge \mathcal{G}, \sigma \models \varphi \wedge \mathcal{G}' \in [\![T]\!](\tau + \sigma|_{\mathrm{fv}(\varphi)}, \mathcal{G})\}$$

where $\mathrm{fv}(\varphi)$ is the set of free variables of φ

$$[\![T_1 \text{ THEN } T_2]\!](\tau, \mathcal{G}) = \bigcup_{\mathcal{G}' \in [\![T_1]\!](\tau, \mathcal{G})} [\![T_2]\!](\tau, \mathcal{G}')$$

$$[\![T_1 \,\square\, T_2]\!](\tau, \mathcal{G}) = [\![T_1]\!](\tau, \mathcal{G}) \cup [\![T_2]\!](\tau, \mathcal{G})$$

$$[\![\text{APPLY_ALL } T]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \mathsf{apply_some}([\![T]\!], \tau, \mathcal{G}, \mathcal{G}')\} - \{\mathcal{G}' \mid \exists \mathcal{G}'' \neq \mathcal{G}'.\ \mathcal{G}'' \in [\![T]\!](\tau, \mathcal{G}')\}$$

where we use $\mathcal{G}, \sigma \models \varphi$ to mean $\mathcal{G}, \sigma, \mathsf{Starts}(\mathcal{G}) \models \varphi$, where $\mathsf{Starts}(\mathcal{G})$ gives the start node for each CFG in $\mathcal{G}$. Note in particular the semantics for APPLY_ALL, which produces the set of graphs that result from applying the transformation T repeatedly and in various ways such that, ultimately, T can no longer be applied to modify the graph. This gives us a formal definition of the graphs that can be produced by a PTRANS specification; in the following chapters, we will see how we can make use of this definition to state and verify optimizations, and also explore its limitations.

3.5 Integrating External Analyses

The CTL side conditions of PTRANS provide a concise and powerful mechanism for expressing the conditions under which optimizations should be performed. However, as mentioned in Section 3.3, we may also want to incorporate information gathered by means other than checking a CTL condition. There may exist analyses whose logic cannot be expressed in CTL, or, more generally, we may care less about the details of the implementation of a static analysis than the (dynamic) guarantees provided by the analysis. One common example of this is *alias analysis*, the process of determining whether two variables or program expressions refer to the same location in memory. A variety of algorithms exist for alias analysis, all of which provide fundamentally the same service: for each program point and pair of expressions, they report that the two expressions either may, must, or cannot refer to the same location. This makes it easy to abstract away from the details and treat alias analysis as a black box; in the LLVM compilation framework, for instance, there are four different alias analysis algorithms that can be called via command-line options, ranging from the minimally precise analysis that returns "may" for every query to fairly sophisticated algorithms [34]. Various optimizations use the results of alias analysis, and their correctness does not depend on the details of the algorithm used, only on the algorithm's correctness (though one implementation may be more effective than another).

Bringing a generalized analysis such as alias analysis into PTRANS is a two-to-three-step process. First, we must axiomatize the analysis, describing the functions it must provide and the properties that those functions must satisfy. In the case of alias analysis, one function is provided – for the purposes of our example, we can simplify this to a function cannot_alias that takes a point in a CFG and a pair of expressions and returns a boolean. The primary requirement on this function is that if cannot_alias is true of a pair of expressions at a program point, then those two expressions must not evaluate to the same location in any execution of the program. We might also want to impose additional requirements: for instance, that cannot_alias is never true of two identical expressions, or that it catches the most obvious cases of non-aliasing (e.g., that two different global variables cannot alias). Second, we add to our set of atomic predicates predicates that call the axiomatized functions, as described in Section 3.3. We can then specify optimizations that rely on the results of the analysis by simply including these predicates in our CTL side conditions. If desired, we can also provide implementations of the analyses, and, by proving that they satisfy the appropriate axioms, use them to obtain more concrete instances of our specifications. These implementations may be provided as algorithms, CTL side conditions, or general mathematical functions. In the formal semantics of PTRANS, we accomplish this axiomatization/implementation procedure through the use of Isabelle's locale mechanism: we state a collection of functions and axioms, and can then instantiate

it with any set of functions specifiable in Isabelle, which may include executable functions in its OCaml-like functional sublanguage or logical characterizations of the relationship between input and output.

In this section, we axiomatize two external analyses: alias analysis, as described above, and mutual exclusion ("mutex") analysis. When applying an optimization for sequential code to a thread in a parallel program, we often find that the transformation is only safe if we are guaranteed that no other thread can see or affect the variables and memory locations involved. If some of those variables or memory locations are shared across threads, we need to know that they will not be modified by other threads while the optimized thread is executing its modified code. This can be ensured through a static analysis for mutual exclusion, which provides a notion of *critical section* such that no two threads can be in a critical section at the same time. If a shared resource is only modified in critical sections, then expressions involving it can be modified without fear of interference from other threads. Our axiomatization of mutex analysis provides two functions: in_critical, which is true on points in a thread that the analysis can show to be in a critical section ensuring mutual exclusion, and protected, a whole-program property that is true of a resource only if that resource is only modified in a critical section. For a given programming language, there may be many different ways to implement a critical section, and multiple methods may be used within a single program; nonetheless, as long as an analysis satisfies the properties of protection (protected resources are only modified in critical sections) and mutual exclusion (two different threads cannot be in_critical for the same resource at the same time), it can be used to construct correct optimizations.

3.5.1 Mutual Exclusion in CTL

To demonstrate the optional third step of incorporating an analysis, we provide a simple implementation of a mutex analysis, and show that it satisfies the relevant axioms. In the spirit of PTRANS, we implement the in_critical and protected functions as CTL formulae on tCFGs (making use of the external alias analysis predicate cannot_alias), and then show that these formulae implement a static analysis for mutual exclusion. Our locking mechanism relies on an atomic compare-and-swap instruction, which we will call `cmpxchg` (syntax drawn from the analogous instruction in the LLVM intermediate language). The instruction $\%\texttt{x} = \texttt{cmpxchg ty}^* \ \texttt{y}, \texttt{ty}\ \texttt{v}_1, \texttt{ty}\ \texttt{v}_2$ assigns the value previously stored at the pointer `y` to `x`, and then checks whether that value is equal to $\texttt{v}_1$ and if so, stores $\texttt{v}_2$ to `y` (the % sign marks a local variable). The code fragment shown attempts to obtain `x` as a lock by checking atomically whether the value stored at `x` is 0 and if so writing 1 to `x`, then proceeds if the lock was successfully obtained and tries again if not, using the conditional branch instruction `br %v`. The lock is released simply by storing 0 to `x`.

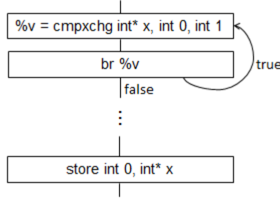


Figure 3.1: A lock implemented with the `cmpxchg` instruction

We can easily construct CTL conditions describing the three nodes in this locking mechanism:

$$\begin{aligned}
\mathsf{locks1}_t(x) &\triangleq \exists v, n.\ \mathsf{stmt}_t(\mathtt{br}\ \%v) \wedge (EX_{t,true}\ (\mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\%v = \mathtt{cmpxchg}\ \mathtt{int}^*\ x, \mathtt{int}\ 0, \mathtt{int}\ 1))) \wedge \\
&\qquad AP_t\ \mathsf{node}_t(n) \\
\mathsf{locks2}_t(x) &\triangleq \exists v, n.\ \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\%v = \mathtt{cmpxchg}\ \mathtt{int}^*\ x, \mathtt{int}\ 0, \mathtt{int}\ 1)) \wedge \\
&\qquad EX_t\ (\mathsf{stmt}_t(\mathtt{br}\ \%v) \wedge \mathsf{out}_t(\mathsf{true}, n)) \\
\mathsf{unlocks}_t(x) &\triangleq \mathsf{stmt}_t(\mathtt{store}\ \mathtt{int}\ 0, \mathtt{int}^*\ x)
\end{aligned}$$

Note that the predicate $\mathsf{locks1}_t$ describes the second (`br`) node of the locking process, while locks2 describes the first (`cmpxchg`) node; this is because we will usually be looking backwards along paths to determine whether they are in the critical section, and so will come to the second node first.

Now we can define the in_critical predicate:

$$\begin{aligned}
\mathsf{locked}_t(x) &\triangleq A\ \neg\mathsf{unlocks}_t(x)\ \mathcal{B}\ \mathsf{locks1}_t(x) \\
\mathsf{in_critical}_t(x) &\triangleq AP_t\ \mathsf{locked}_t(x)
\end{aligned}$$

A thread t has "locked" x if it has not released the lock since its last attempt to acquire it (which may currently be in progress). If x must have been locked in all previous states, then the thread is in the critical section for x.

Before we can define protected, we need to make sure that our locking mechanism is actually effective. While one thread may treat x as a lock, if another thread stores arbitrary values to x – or, still worse, if the memory at x is deallocated and then allocated by another thread – it will not guarantee mutual exclusion. The former can be prevented by a not_touches predicate that ensures that a given memory location is not read or modified by a given thread. Using the alias analysis predicate described above, this predicate can

be defined as follows:

$$\mathsf{not_touches}_t(e) \triangleq \forall e'\ x\ ty_1\ e_1\ ty_2\ e_2\ ty_3\ e_3.\ (\mathsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e') \vee \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e', ty_2\ e_2, ty_3\ e_3) \vee \mathsf{stmt}_t(\%x = \texttt{load}\ ty_1^*\ e')) \Rightarrow \mathsf{cannot_alias}_t(e', e)$$

This predicate ensures that if the current instruction in t is a memory instruction, then the pointer involved cannot alias to e, so the memory referenced by e will not be modified. As long as threads only interact with a memory location by locking and unlocking it, only unlock it when they have it, and only acquire the lock in the process of entering a critical section, the location serves as a lock:

$$\mathsf{good_lock}_t(x) \triangleq AG\ ((\neg\mathsf{locks2}_t(x) \wedge \neg\mathsf{unlocks}_t(x) \Rightarrow \mathsf{not_touches}_t(x)) \wedge (\mathsf{unlocks}_t(x) \Rightarrow \mathsf{in_critical}_t(x)) \wedge ((EP_t\ \mathsf{locks}_t(x)) \Rightarrow \mathsf{in_critical}_t(x)))$$

Now we can define a CTL formula that is true when a memory location is protected by a lock:

$$\mathsf{protected}(x, e) \triangleq \mathsf{gvarlit}(e) \wedge \forall t.\ \mathsf{good_lock}_t(x) \wedge AG\ (\neg\mathsf{in_critical}_t(x) \Rightarrow \mathsf{not_touches}_t(e))$$

where $\mathsf{gvarlit}(e)$ is a language-specific predicate that asserts that e is a global variable. This requirement is important because it gives us a way of identifying the protected location across multiple threads; while in theory a local variable could also become a shared resource by exposing its address to other threads, it would take a much more complicated analysis to identify whether a memory reference in one thread could alias to a local variable in another thread.

Now we can show that these predicates satisfy the axioms of mutex analysis discussed in the previous section. For more on the transition relation described here and its use in verification, see Section 4.3.

Theorem 1 (Protection). *Suppose that*

- $\mathsf{protected}(x, e)$ *holds on a tCFG* $\mathcal{G}$,
- $(states, m)$ *is a reachable state in* $\mathcal{G}$,
- $\mathcal{G}_t, t, m \vdash states(t) \xrightarrow{a} s'$ *for some thread* t, *and*
- a *involves the memory at* e.

Then $\mathsf{in_critical}_t(x)$ *holds at* $states$.

Proof. Since $\mathsf{protected}(x, e)$ implies that e is a global variable, its memory cannot be allocated or deallocated. Thus, the instruction at $states(t)$ must be one of `load`, `store`, and `cmpxchg`. However, this means that $\mathsf{not_touches}_t(e)$ does not hold, so by the definition of protected, $\mathsf{in_critical}_t(x)$ must hold. □

Theorem 2 (Mutual Exclusion)**.** *Suppose that* $\mathsf{protected}(x, e)$ *holds on a tCFG* $\mathcal{G}$*, that* $(states, m)$ *is a reachable state in* $\mathcal{G}$*, and that* $\mathsf{in_critical}_t(x)$ *and* $\mathsf{in_critical}_u(x)$ *both hold. Then* $t = u$*.*

Proof. We prove the theorem from a stronger result with two cases: the case in which the lock has been claimed by some thread (i.e., $m(\ell) = 1$, where ℓ is the location of x), and the case in which it has not. When the lock has been claimed, we will show that either some thread is in the critical section, or some thread has just successfully completed the `cmpxchg` instruction to claim the lock, and no other thread is in the critical section or successfully completed the `cmpxchg` instruction. When the lock has not been claimed, we will show that no thread is in the critical section and no thread has claimed the lock. We can see that if these properties holds of a state, then mutual exclusion also holds. Throughout the proof, we use the definitions of the protected and good_lock predicates to show that the location x behaves properly as a lock.

The proof of this result proceeds by induction on the execution by which $(states, m)$ is reachable. In the base case, *states* is the initial state s_0; in this case, no thread is in the critical section, since the start node of each thread is the first node in that thread. In the inductive case, we consider each case of the inductive hypothesis separately. If $m(\ell) = 1$ and some thread has claimed the lock or is in the critical section, then no other thread can successfully claim the lock or enter the critical section, since the `cmpxchg` instruction will fail; if the thread that has the lock releases it, then $m(\ell)$ is set to 0 and we reach a state in which no thread is in the critical section or has claimed the lock. If $m(\ell) \neq 1$ and no thread is in the critical section, then after a step either this state of affairs continues to hold, or some thread claims the lock, and thus sets $m(\ell)$ to 1 and becomes the unique thread that has claimed the lock. We can conclude that the protected and in_critical predicates correctly implement mutual exclusion. □

While this locking mechanism and this result are specific to the language used to implement the locking mechanism (a variant of the LLVM intermediate language called MiniLLVM, which will be presented in full detail in Section 4.3), the in_critical and protected predicates can be used in side conditions for programs in any language, with either a different implementation or no implementation at all: an implementation is required if we want to execute the transformations, and helps to demonstrate that the axioms are satisfiable, but it is not necessary in order to state or verify a transformation, since the required information is completely captured in the axioms of mutual exclusion. This approach extends the flexibility of PTRANS and allows us to make use of graph analyses independently of their implementation details.

Chapter 4

Intermediate Languages for PTRANS

4.1 Compiler Intermediate Representations

Throughout the history of compilers, various intermediate representations have been used to store and optimize program code. In general, the front end of a compiler parses code into an abstract syntax tree (AST), removing ambiguities and reporting syntax errors. However, in an AST, shared computations, redundancies, and other features of interest in optimization may be hidden inside high-level constructs or buried deep inside the tree. The common practice for as long as high-level languages have been compiled has been to translate the AST into a representation in which these features are more obvious, and to analyze and optimize code at this level before generating the low-level output. The intermediate representation (IR) is, as its name suggests, an intermediate level between the (usually high-level) input to the compiler and its (usually low-level) output. In fact, most modern compilers have several levels of intermediate representation, moving gradually from high-level human-readable code to architecture-specific machine code.

Most intermediate representations are specific to the compiler in which they occur, and may include code representations, data and control flow graphs, information on shared expressions, and other information as appropriate to the language being compiled. For PTRANS, we focus on a particular subset of intermediate representations. A target IR for PTRANS must store code in the form of *instruction-labeled control flow graphs*. Each node is labeled with an *instruction*, an atomic unit of computation with no complex subexpressions, and has outgoing edges indicating the next instruction to be executed. If the node has multiple outgoing edges, one will be selected (usually deterministically) based on some property of the computation; for instance, a conditional branch instruction will generally have a `branch` edge that is followed when the condition is true and a `seq` edge that is followed when the condition is false. The structure of a control flow graph is generally more or less independent of the language under consideration; thus, we will speak of a *target language* or *intermediate language* for PTRANS as a language of instruction labels along with some relationship between instruction labels and outgoing edges.

Within this framework, there is still room for variety in program representation, and in particular in the

language of instruction labels. We can identify two major program models used in compiler IRs, both of which fit into the control-flow-graph approaches. The first includes three-access-code and other assembly-language-like languages, generally referred to as *register machines*, in which an instruction consists of an operator together with several operands (constants, variables, or simple expressions), and intermediate results are stored in temporary locations called registers. In contrast to this, *stack machines* store intermediate results on a stack of computed values. Stack machine instructions take their arguments from the top of the stack and place their results back onto the stack, so operands need not be given explicitly. This makes the program representation more compact, but may also obscure redundant computations and is generally more difficult to analyze than the register-machine approach. Many optimizations that are commonly performed on register machines (e.g., removal of redundant assignments to registers) cannot be transferred to stack machines, and vice versa; other optimizations (e.g., those involving references to shared memory) are performed similarly under both approaches.

In this chapter, we will introduce two sample target languages for PTRANS, illustrating two different approaches to intermediate language semantics, and discuss the differences between them. We will also consider the general problem of giving semantics to intermediate languages in terms of (parallel) control flow graphs, with particular attention paid to the role of concurrent memory models. In the following chapters, we will show the use of PTRANS tools and approaches on both languages, and when possible generalize to language-independent results.

4.2 Language Semantics on (t)CFGs

4.2.1 Single-Thread CFG Semantics

Formal operational semantics for programming languages are often given in terms of small-step relations [63]; given a program command and a configuration representing the execution state of the program, we write $(c, s) \rightarrow (c', s')$ to indicate that a command c may be reduced to a command c', while changing the state from s to s'. We use this basic approach to give semantics to our target languages for PTRANS, but some aspects of our approach are shaped by the fact that our program model is always a threaded control flow graph. For each language, we will begin with a step relation describing the behavior of a single program thread (modeled as a single-thread CFG). This relation has the form $t, G, m \vdash s \xrightarrow{a} s'$, where t is the name of the executing thread, G is a CFG, m is a shared memory, s and s' are configurations, and a is the set of memory operations performed in the step (we will often write $t, G, m \vdash s \xrightarrow{\emptyset} s'$ as $t, G, m \vdash s \rightarrow s'$). The configuration includes the node of the instruction to execute, the *program point*, which we will write as

$\mathsf{point}(s)$; it also contains thread-local state information appropriate to the language, such as the values of registers or the evaluation stack. We separate the shared memory from the per-thread configuration, and the memory operations performed from the actual effect on memory, to allow us to parameterize by the memory model used; for more on our approach to memory models, see Section 4.5.

Although each language has its own step relation, there are certain commonalities in their structures. For instance, consider the following rule for an assignment statement of the sort that appears in many programming languages:

$$\frac{(e, m, s) \Downarrow v}{(x := e; \vec{c}, m, s) \rightarrow (\vec{c}, m, s(x \mapsto v))}$$

where $\vec{c}$ is the remaining program, s is the local state and m is the shared memory. The analogous CFG rule is:

$$\frac{\mathsf{Label}\ G\ n = x := e \quad n \neq \mathsf{Exit}\ G \quad (e, m, s) \Downarrow v}{t, G, m \vdash (n, s) \rightarrow (\mathsf{next}\ G\ \mathsf{seq}\ n, s(x \mapsto v))}$$

where $\mathsf{next}\ G\ \ell\ n$ chooses the node n' such that $(n, \ell, n') \in \mathsf{Edges}\ G$ (we expect the instruction-to-edge-types relation for the language to ensure that there is exactly one such n'). Similarly, given a conditional branch rule such as

$$\frac{(b, m, s) \Downarrow v \quad c' = \text{if } v \text{ then } \mathsf{code}(L) \text{ else } \vec{c}}{(\mathtt{branch_if}\ b\ L; \vec{c}, m, s) \rightarrow (c', m, s)}$$

where $\mathsf{code}(L)$ gets the code to be executed at label L, the corresponding CFG rule would be

$$\frac{\mathsf{Label}\ G\ n = \mathtt{branch_if}\ b \quad n \neq \mathsf{Exit}\ G \quad (b, m, s) \Downarrow v \quad \ell = \text{if } v \text{ then } \mathsf{branch} \text{ else } \mathsf{seq}}{t, G, m \vdash (n, s) \rightarrow (\mathsf{next}\ G\ \ell\ n, s)}$$

These examples indicate several patterns in CFG semantics:

- Expression computation is performed as in non-CFG code (although we will usually insist that expressions be relatively simple, reflecting the low level of our target languages).
- We store the program point as a node in the CFG in our configuration, and use it to determine the next instruction to execute.
- Control flow is entirely encoded in the CFG's edges; control-flow-in-syntax elements such as labels are unnecessary.
- We usually have a sequence edge type that represents continuing with straight-line execution, as well as one or more types indicating more complex control flow.
- The new program point is reached by following one of the edges of the CFG, and is completely determined by the kind of edge to be followed (in these examples, seq or branch).

- We never execute the Exit node of a graph, since it has no outgoing edges.

These patterns are not universally followed – for instance, when returning from a procedure in a language with procedure calls, the new program point may be determined by the call stack as well as the CFG's edges – but they apply to most of the instructions in the languages we have examined thus far. Automated generation of CFG semantics for a language from its AST or instruction-list semantics would be an interesting area for future work; for the time being, these guidelines make it relatively easy to perform the translation manually, or to construct new CFG semantics given a familiarity with ordinary small-step semantics. For stronger guarantees of correctness, if we already have operational semantics for a language, we can prove that the CFG semantics we have written is sound and complete with respect to the original semantics.

4.2.2 Memory Operations and Concurrency

As mentioned above, we make a point of separating out the memory operations performed by an instruction in our semantics. For instance, suppose we had a `store` instruction with the usual straight-line semantics:

$$\frac{(e, m, s) \Downarrow v \quad (e', m, s) \Downarrow p}{(\texttt{store}\ e\ e'; \vec{c}, m, s) \rightarrow (\vec{c}, m(p \mapsto v), s)}$$

Then our CFG rule for the instruction would be:

$$\frac{\mathsf{Label}\ G\ n = \texttt{store}\ e\ e' \quad n \neq \mathsf{Exit}\ G \quad (e, m, s) \Downarrow v \quad (e', m, s) \Downarrow p}{t, G, m \vdash (n, s) \xrightarrow{\text{write}\ t\ p\ v} (\mathsf{next}\ G\ \mathsf{seq}\ n, s)}$$

This allows us to parameterize our semantics by a memory model. Transitions are labeled with a set of memory operations, where memory operations are chosen from:

$$a ::= \mathsf{read}\ t\ loc\ v \mid \mathsf{write}\ t\ loc\ v \mid \mathsf{arw}\ t\ loc\ v\ v \mid \mathsf{alloc}\ t\ loc \mid \mathsf{free}\ t\ loc$$

The read and write operations represent memory reads and writes, alloc and free represent allocating and freeing memory locations, and arw represents an atomic read-and-write operation (as performed by the `cmpxchg` instruction in LLVM, for instance).

We can extend our memory-model-agnostic step relation to a memory-model-aware single-thread semantics with a single, language-independent inference rule:

$$\frac{t, G, m \vdash s \xrightarrow{a} s' \quad \mathsf{update_mem}\ m\ a\ m'}{t, G \vdash (s, m) \rightarrow (s', m')}$$

where update_mem is a relation provided by the memory model, indicating that m' is a possible result of updating m with the operations in a (see Section 4.5 for details). A similar, also language-independent rule allows us to extend our single-thread step relation to a concurrent semantics. A concurrent configuration has the form $(states, m)$, where $states$ is a map from threads to thread-local states and m is a shared memory. Then given a tCFG $\mathcal{G}$, the concurrent step relation is completely characterized by the rule:

$$\frac{\mathcal{G}_t = G \quad states_t = s \quad t, G, m \vdash s \xrightarrow{a} s' \quad \mathsf{update_mem}\ m\ a\ m'}{\mathcal{G} \vdash (states, m) \rightarrow (states(t \mapsto s'), m')}$$

Thus, once we have a single-thread step relation for our target language, we automatically receive a concurrent semantics suitable for verification of optimizations on multithreaded programs.

4.3 Intermediate Language 1: MiniLLVM

Our first target language is MiniLLVM, a language based on the LLVM intermediate representation [34]. LLVM is a compiler infrastructure which has been used in compilers for a range of target languages, from C and Java to Python and Haskell. As a language-independent intermediate representation that represents code as control flow graphs, the LLVM IR is an ideal target for PTRANS. The LLVM IR uses a register-machine approach; an infinite number of registers is assumed, and each program has exactly one assignment statement for each register it uses – the Static Single Assignment (SSA) constraint. Each instruction specifies a simple operation to be performed on a small collection of constants, registers, and memory locations, with its result (if any) stored to a register. Registers are assumed to be thread-local and temporary, while memory locations are shared and persistent.

Syntactically, MiniLLVM is a subset of the LLVM IR with some slight modifications. Labels are omitted, since the targets of jump instructions can be determined from the control flow graph (MiniLLVM has no indirect jumps). We also omit several formal details of the IR: for instance, while types are included in the syntax, we perform very little dynamic type checking, and while the LLVM IR is always in static single assignment form, MiniLLVM programs are not in SSA by default. This simplifies the formalization and reduces the number of well-formedness constraints that need to be carried through proofs of correctness.

4.3.1 Syntax

The syntax of MiniLLVM is defined as follows:

$$expr ::= \%\mathrm{x} \mid @\mathrm{x} \mid \mathrm{c} \qquad\qquad type ::= \texttt{int} \mid type\ \texttt{pointer}$$

$$
\begin{aligned}
\textit{instr} ::= \;& \%\text{x} = \text{op } \textit{type expr}, \textit{ expr} \mid \%\text{x} = \texttt{icmp} \text{ cmp } \textit{type expr},\textit{expr} \mid \texttt{br } \textit{expr} \mid \texttt{br} \mid \\
& \%\text{x} = \texttt{call } \textit{type } (\textit{expr}, ..., \textit{expr}) \mid \texttt{return } \textit{expr} \mid \texttt{alloca } \%\text{x } \textit{type} \mid \\
& \%\text{x} = \texttt{load } \textit{type}^* \textit{ expr} \mid \texttt{store } \textit{type expr}, \textit{ type}^* \textit{ expr} \mid \\
& \%\text{x} = \texttt{cmpxchg } \textit{type}^* \textit{ expr}, \textit{ type expr}, \textit{ type expr} \mid \texttt{is_pointer } \textit{expr}
\end{aligned}
$$

MiniLLVM instructions include arithmetic operations (where op is an arithmetic operator), comparison operations (where cmp is a comparison operator), conditional and unconditional branches, function calls and returns, memory allocation, loads from and stores to memory, an atomic compare-and-exchange instruction, and `is_pointer`, which checks whether a given expression is pointer-valued (for use in loads and stores). (Note that the *'s indicate not repetition but pointer types.) Because the targets of control-flow instructions are implicit in the CFG, the label arguments to `br` instructions and function names in `call` instructions are omitted. The `cmpxchg` instruction performs a conditional atomic read-write: $\texttt{cmpxchg } \texttt{t}_1^* \texttt{ e}_1, \texttt{t}_2 \texttt{ e}_2, \texttt{t}_3 \texttt{ e}_3$ first reads the value stored at the location $\texttt{e}_1$ and compares it with the value of $\texttt{e}_2$. If they are equal, it stores the value of $\texttt{e}_3$ to $\texttt{e}_1$; in either case, it returns the value that was previously at $\texttt{e}_1$, and all these operations are performed atomically. This instruction can be used to enforce synchronization between multiple threads, and, as described in Section 4.5, also acts as a memory fence in relaxed memory models.

4.3.2 Instruction Types and CFG Edge Labels

Recall from Section 3.2 that our definition of a control flow graph is affected by the choice of target language. In particular, the target language must provide a relationship between instructions (which will be used as node labels) and outgoing edges in a CFG for that language. For MiniLLVM, the relationship is defined as follows:

- A node labeled with a conditional branch `br` e has one outgoing edge labeled true and one labeled false.
- A node labeled with a function call `call` ty $(e_1, ..., e_n)$ has one outgoing edge labeled call (leading to the function body) and one labeled seq (indicating the location to which to return once the call is complete).
- A node labeled with a return instruction `return` e can have any number of outgoing edges, all labeled ret.
- A node with any other label has one outgoing edge labeled seq.

4.3.3 Semantics

As laid out in Section 4.2.1, we give semantics to our target languages by specifying a labeled transition relation on configurations. In MiniLLVM, a configuration is a tuple (p, env, st, al), where p is a program point, env is an environment giving values for thread-local variables, st is the call stack for the thread, and al is a record of the memory locations allocated by the currently executing procedure. The semantics of an individual thread are then given by a transition relation $t, G, m \vdash (p, env, st, al) \xrightarrow{a} (p', env', st', al')$, where t is the thread name, G is the CFG representing the thread, m is the shared memory, and a is set of memory operations performed by the thread (in MiniLLVM, usually either one or no operations are performed in a step). The semantics are parameterized by a *memory model* that provides functions can_read, free_set, and update_mem for interacting with the shared memory (as explained in detail in Section 4.5). The semantic rules for MiniLLVM instructions are shown in Figure 4.1. We make use of a helper function init *env args* that creates a new environment that is empty except for the binding of formal parameters `arg0`, `arg1`, etc. to the values given by *args* in the environment *env*.

The `alloca`, `call` and `return` instructions merit particular attention. In LLVM, the `alloca` instruction allocates memory on the stack frame of the currently executing function, to be freed when the function returns [46]. We model this by maintaining a set al of allocated memory for the currently executing function in each thread. When a function call is performed, a stack frame (p, x, env, al) is added to the stack, where p is the return address of the function, x is the variable that will receive the function's return value, env is the environment in effect when the function is called, and al is the memory allocated by the calling function (*not* the function being called, which has so far allocated no memory). When a function returns, the call stack is popped and the state carried in the stack frame restored; at the same time, all the memory in the returning function's al set is freed, as in LLVM.

The `cmpxchg` instruction is the only one whose semantics is given by multiple rules: depending on whether the value at the location indicated by e_1 equals the test value e_2, it will either write the value of e_3 to the location or else leave it as is. In either case, the memory operation performed is an arw operation. A failed `cmpxchg` might alternatively be modeled as a `read` operation, but by producing arw instead, it indicates that an atomic operation has been performed, allowing the memory model to update itself accordingly: as we will see in Section 4.5, some memory models have special behavior when atomic operations are performed, regardless of whether they succeed or fail.

$$\frac{\textsf{Label}\ G\ p = (\%x = \mathrm{op}\ ty\ e_1, e_2) \qquad (e_1\ \mathrm{op}\ e_2, env) \Downarrow v}{t, G, m \vdash (p, env, st, al) \to (\textsf{next seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\textsf{Label}\ G\ p = (\%x = \texttt{icmp}\ \mathrm{cmp}\ ty\ e_1, e_2) \qquad (e_1\ \mathrm{cmp}\ e_2, env) \Downarrow v}{t, G, m \vdash (p, env, st, al) \to (\textsf{next seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\textsf{Label}\ G\ p = (\texttt{br}\ e) \qquad (e, env) \Downarrow v \qquad v = 0}{t, G, m \vdash (p, env, st, al) \to (\textsf{next false}\ p, env, st, al)} \qquad \frac{\textsf{Label}\ G\ p = (\texttt{br}\ e) \qquad (e, env) \Downarrow v \qquad v \neq 0}{t, G, m \vdash (p, env, st, al) \to (\textsf{next true}\ p, env, st, al)}$$

$$\frac{\textsf{Label}\ G\ p = (\texttt{br})}{t, G, m \vdash (p, env, st, al) \to (\textsf{next seq}\ p, env, st, al)}$$

$$\frac{\textsf{Label}\ G\ p = (\texttt{alloca}\ \%x\ ty) \qquad loc \in \textsf{free_set}\ m}{t, G, m \vdash (p, env, st, al) \xrightarrow{\textsf{alloc}\ t\ loc} (\textsf{next seq}\ p, env(x \mapsto loc), st, al \cup \{loc\})}$$

$$\frac{\textsf{Label}\ G\ p = (\%x = \texttt{call}\ ty\ args)}{t, G, m \vdash (p, env, st, al) \to (\textsf{next call}\ p, \textsf{init}\ env\ args, (\textsf{next seq}\ p, x, env, al); st, \emptyset)}$$

$$\frac{\textsf{Label}\ G\ p = (\texttt{return}\ e) \qquad (e, env) \Downarrow v \qquad (p, \textsf{ret}, p') \in \textsf{Edges}\ G}{t, G, m \vdash (p, env, (p', x, env', al'); st, al) \xrightarrow{\textsf{free}\ t\ al} (p', env'(x \mapsto v), st, al')}$$

$$\frac{\textsf{Label}\ G\ p = (\%x = \texttt{load}\ ty^*\ e) \qquad (e, env) \Downarrow loc \qquad v \in \textsf{can_read}\ m\ t\ loc}{t, G, m \vdash (p, env, st, al) \xrightarrow{\textsf{read}\ t\ loc\ v} (\textsf{next seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\textsf{Label}\ G\ p = (\texttt{store}\ ty_1\ e_1, ty_2{}^*\ e_2) \qquad (e_1, env) \Downarrow v \qquad (e_2, env) \Downarrow loc}{t, G, m \vdash (p, env, st, al) \xrightarrow{\textsf{write}\ t\ loc\ v} (\textsf{next seq}\ p, env, st, al)}$$

$$\frac{\begin{array}{c}\textsf{Label}\ G\ p = (\%x = \texttt{cmpxchg}\ ty_1{}^*\ e_1, ty_2\ e_2, ty_3\ e_3) \\ (e_1, env) \Downarrow loc \qquad (e_2, env) \Downarrow v \qquad v \in \textsf{can_read}\ m\ t\ loc \qquad (e_3, env) \Downarrow v'\end{array}}{t, G, m \vdash (p, env, st, al) \xrightarrow{\textsf{arw}\ t\ loc\ v\ v'} (\textsf{next seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\begin{array}{c}\textsf{Label}\ G\ p = (\%x = \texttt{cmpxchg}\ ty_1{}^*\ e_1, ty_2\ e_2, ty_3\ e_3) \\ (e_1, env) \Downarrow loc \qquad (e_2, env) \Downarrow v' \qquad v \in \textsf{can_read}\ m\ t\ loc \qquad v' \neq v\end{array}}{t, G, m \vdash (p, env, st, al) \xrightarrow{\textsf{arw}\ t\ loc\ v\ v} (\textsf{next seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\textsf{Label}\ G\ p = (\texttt{is_pointer}\ e) \qquad (e, env) \Downarrow loc}{t, G, m \vdash (p, env, st, al) \to (\textsf{next seq}\ p, env, st, al)}$$

Figure 4.1: The single-threaded transition semantics of MiniLLVM

4.4 Intermediate Language 2: GraphBIL

4.4.1 Baby IL

Our second target language is adapted from Gordon and Syme's Baby IL (BIL) [22]. BIL is itself a simple subset of the Common Intermediate Language (CIL) [17], which is used as an intermediate representation for the compilers of the .NET Framework. Like the LLVM IR, the CIL serves as a common intermediate representation for compilers from a range of languages, including C++, C#, Visual Basic, and F#. Because it was designed as a target language for object-oriented languages such as C#, CIL provides first-class support for classes, methods, and inheritance/dynamic dispatch.

BIL, the formalized subset of CIL, treats programs as complex expressions, with each operation postfixed to its arguments: for instance, a store operation of the value of an expression a to a memory location computed by b is written a b `stind`. A BIL program is a collection of methods, where each method consists of a *signature* and a *body*. Method signatures are of the form $type\, method(type, ..., type)$, specifying the method's return type, name, and number and types of arguments. Method bodies contain the code that is actually executed, and are constructed as follows:

$$type ::= \texttt{void} \mid \texttt{int32} \mid \texttt{class}\ class \mid \texttt{value class}\ class \mid type\&$$

$$\begin{aligned} b ::= \ & \texttt{ldc.i4}\ int \mid b\ b\ b\ \texttt{cond} \mid b\ b\ \texttt{while} \mid b\ b \mid b\ \texttt{ldind} \mid b\ b\ \texttt{stind} \mid \texttt{ldarga}\ int \mid b\ \texttt{starg}\ int \mid \\ & b\ ...\ b\ \texttt{newobj void}\ class\texttt{::}\ \texttt{.ctor}(type, ..., type) \mid \\ & b\ b\ ...\ b\ \texttt{callvirt}\ type\ class::method(type, ..., type) \mid \\ & b\ b\ ...\ b\ \texttt{call instance}\ type\ class::method(type, ..., type) \mid \\ & b\ \texttt{ldflda}\ type\ class::field \mid b\ b\ \texttt{stfld}\ type\ class::field \mid b\ \texttt{box}\ class \mid b\ \texttt{unbox}\ class \end{aligned}$$

The `ldc.i4` instruction simply returns an integer value, and `cond`, `while`, and sequencing express common control-flow concepts. Of more interest are the memory-affecting and object-oriented instructions. Classes are divided into value classes, which can be stored without class annotations and held directly in the fields of other objects, and reference classes, which can inherit from other classes, must be stored "boxed" with an annotation of their class (value objects may or may not be boxed), and cannot be held in fields of other objects (though pointers to them can). The `callvirt` instruction calls a method of a reference class; at runtime, the actual class of the calling object is determined, and the call is dispatched to the method body for that class, providing object-oriented polymorphism. The `call instance` instruction, on the other hand, applies only to value classes, and never needs to consider polymorphism. In addition to storing to heap

locations, BIL also includes stores to particular fields of heap locations, in which case the object at that heap location is extracted, the indicated field modified, and the modified object replaced in its location. We will see the semantics of BIL in detail as we develop a related but distinct language for use with PTRANS.

4.4.2 From BIL to GraphBIL: Syntax

The code representation of BIL is a level too high for PTRANS; since a and b may be complex expressions containing multiple memory operations, a b `stind` is not a suitable label for a CFG node. However, BIL is derived from CIL, a low-level intermediate language that executes as a stack machine. Beginning with the syntax and semantics of BIL, we can derive a formal description of a language that uses the CIL program model, and use this lower-level language, which we call GraphBIL, as a target language for PTRANS.

The syntax of GraphBIL is defined as follows:

instr ::= `ldc.i4` *int* | `br` | `brtrue` | `brfalse` | `ldind` | `stind` | `ldarga` *int* | `starg` *int* |
`newobj void` *class*:: `.ctor`(*type*, ..., *type*) | `callvirt` *type class*::*method*(*type*, ..., *type*) |
`call instance` *type class*::*method*(*type*, ..., *type*) | `ret` |
`ldflda` *type class* :: *field* | `stfld` *type class* :: *field* | `box` *class* | `unbox` *class* | `dup` | `pop`

Most of the instructions are straightforwardly derived from BIL instructions, with arguments to operators drawn from the evaluation stack rather than explicit in the instructions. In addition, we have broken `cond` and `while` instructions into more atomic `br` instructions (and omitted sequencing entirely), and added a `ret` instruction to signal the end of a method call. Finally, we add two instructions from CIL that are useful in optimizing stack machine programs: `dup` duplicates the top element of the evaluation stack, and `pop` removes the top element.

4.4.3 Instruction Types and CFG Edge Labels

The allowed outgoing edge types for each GraphBIL instruction are as follows:

- A node labeled with a conditional branch `brtrue` or `brfalse` has one outgoing edge labeled true and one labeled false.

- A node that calls a method of an unboxed object `call instance` $(B\ vc :: \ell(A_1, ..., A_n))$ has one outgoing edge labeled call vc (leading to the method body) and one labeled seq (indicating the location to which to return once the call is complete).

- A node that calls a method of a boxed object `callvirt` $(B\ c :: \ell(A_1, ..., A_n))$ has one outgoing edge labeled seq (indicating the location to which to return once the call is complete), and one edge labeled $\mathsf{mcall}\ c'$ for each class c' such that $c' <: c$.

- A node labeled with a return instruction `return` e can have any number of outgoing edges, all labeled seq.

- A node with any other label has one outgoing edge labeled seq.

Of particular interest is the requirement for the `callvirt` instruction, which ensures that a virtual method call can be dynamically dispatched to any subclass of the class given in the instruction.

4.4.4 From BIL to GraphBIL: Semantics

The semantics of BIL are given by a big-step evaluation relation $(h, s) \vdash b \rightsquigarrow v \cdot (h', s')$, where h and h' are heaps, s and s' are call stacks (storing the arguments passed to each currently executing method), b is a method body, and v is the resulting value (or $\mathbf{0}$ if no value is produced). We will show a few inference rules for BIL and demonstrate how we convert them into a small-step stack machine semantics for GraphBIL, following the guidelines of Section 4.2.1.

To give low-level CFG semantics to GraphBIL, we must make several elements of the CIL execution state explicit: in particular, the evaluation stack (on which operands to future instructions are placed) and the program point. We must also add a distinction between thread-local state and shared memory. Fortunately, the CIL standard [17] is quite explicit on this subject: call stack, evaluation stack, argument list, and program point are thread-local, and the heap is shared. Thus, GraphBIL execution states have the form $(s, vs, args, n)$, where vs is the evaluation stack, $args$ is the list of arguments to the currently executing method, n is the program point in the CFG, and s is the call stack, which in turn contains triples $(vs, args, n)$ of method execution states. This gives us a step relation judgment of the form $t, G, h \vdash (s, vs, args, n) \xrightarrow{a} (s', vs', args', n')$. Note that there is a fixed relationship between the call stack of a GraphBIL program and that of the corresponding BIL program: if a GraphBIL program is in a state $(s, vs, args, n)$ where $s = (vs_1, args_1, n_1) \ldots (vs_k, args_k, n_k)$ (we write stacks as growing from left to right), this corresponds to the BIL stack $\mathsf{BIL_stack}(s, args) \triangleq args_1, ..., args_k, args$. We also use a helper function $\mathsf{stack_update}(s, args, ptr, v)$ that takes a GraphBIL stack, derives the corresponding BIL stack, updates the target of ptr with the value v (assuming that ptr is at base a stack reference), and returns the resulting stack and argument list.

We derive the inference rules for GraphBIL beginning with the simplest BIL instruction:

$$\frac{}{(h,s) \vdash \texttt{ldc.i4}\ i \rightsquigarrow i \cdot (h,s)} \qquad \frac{\mathsf{Label}\ G\ n = \texttt{ldc.i4}\ i \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs, args, n) \rightarrow (s, vs\ i, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

Instead of reducing to the value i as a result, we add it to the top of the evaluation stack vs.

The BIL `ldind` and `stind` instructions take pointer values indicating the location at which to perform the load/store. These pointer values are defined as

$$ptr ::= href \mid (int, int) \mid ptr.field$$

where $href$ is a heap reference, (i, j) is a pointer to the jth argument of the ith stack frame in the call stack (where the outermost method call is number 1 and inner calls count up), and $p.f$ refers to the field of the object at ptr (which itself may be a heap reference, stack reference, or field reference). We can see that at its base, every pointer is either a heap reference or a stack reference, which points to another way to look at pointers: following Gordon and Syme, we will often write a pointer as $p.\vec{f}$, where p is the base heap or stack reference and $\vec{f}$ is the (possibly empty) list of field references attached to it.

The BIL rules for `ldind` and `stind` are:

$$\frac{(h,s) \vdash a \rightsquigarrow ptr \cdot (h', s')}{(h,s) \vdash a\ \texttt{ldind} \rightsquigarrow \mathsf{lookup}(s', h', ptr) \cdot (h', s')} \qquad \frac{(h,s) \vdash a \rightsquigarrow ptr \cdot (h', s') \quad (h', s') \vdash b \rightsquigarrow v \cdot (h'', s'')}{(h,s) \vdash a\ b\ \texttt{stind} \rightsquigarrow \mathbf{0} \cdot \mathsf{update}(h'', s'', ptr, v)}$$

where lookup and update are helper functions that break ptr into $p.\vec{f}$, determine whether p is a heap reference or a stack reference, and then retrieve or modify the value at p (if $\vec{f}$ is empty) or the (sub)field $\vec{f}$ of the object at p. These rules require closer examination before transforming them into GraphBIL. The instructions take arguments, which in GraphBIL will already have been evaluated and pushed onto the evaluation stack, but there is a subtler difference as well: in BIL, the heap and call stack are treated uniformly. When writing semantics for concurrency, however, we must consider the fact that the heap is shared memory and the stack is thread-local; while reads and writes to the stack may be executed by the step relation, reads and writes to the heap must be handed over to the memory model. This leads us to give separate stack and heap rules for each of `ldind` and `stind`:

$$\frac{\mathsf{Label}\ G\ n = \texttt{ldind} \quad n \neq \mathsf{Exit}\ G \quad \mathsf{lookup}(\mathsf{BIL_stack}(s, args), h, (i,j).\vec{f}) = v}{t, G, h \vdash (s, vs; (i,j).\vec{f}, args, n) \rightarrow (s, vs; v, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{ldc.i4}\ i \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs, args, n) \to (s, vs; i, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{br} \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs, args, n) \to (s, vs, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{brtrue} \quad n \neq \mathsf{Exit}\ G \qquad e = \text{if } v = 0 \text{ then } \mathsf{seq} \text{ else } \mathsf{branch}}{t, G, h \vdash (s, vs, args, n) \to (s, vs, args, \mathsf{next}\ G\ e\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{brfalse} \quad n \neq \mathsf{Exit}\ G \qquad e = \text{if } v = 0 \text{ then } \mathsf{branch} \text{ else } \mathsf{seq}}{t, G, h \vdash (s, vs, args, n) \to (s, vs, args, \mathsf{next}\ G\ e\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{ldind} \quad n \neq \mathsf{Exit}\ G \quad \mathsf{lookup}(\mathsf{BIL_stack}(s, args), h, (i,j).\vec{f}) = v}{t, G, h \vdash (s, vs; (i,j).\vec{f}, args, n) \to (s, vs; v, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{ldind} \quad n \neq \mathsf{Exit}\ G \quad v \in \mathsf{can_read}\ h\ t\ p.\vec{f}}{t, G, h \vdash (s, vs; p.\vec{f}, args, n) \xrightarrow{\mathsf{read}\ t\ p.\vec{f}\ v} (s, vs; v, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{stind} \quad n \neq \mathsf{Exit}\ G \quad \mathsf{stack_update}(\mathsf{BIL_stack}(s, args), h, (i,j).\vec{f}, v) = (s', args')}{t, G, h \vdash (s, vs; (i,j).\vec{f}; v, args, n) \to (s', vs, args', \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{stind} \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; p.\vec{f}; v, args, n) \xrightarrow{\mathsf{write}\ t\ p.\vec{f}\ v} (s, vs, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{ldarga}\ j \quad n \neq \mathsf{Exit}\ G \quad s = s_1\ ...\ s_i}{t, G, h \vdash (s, vs, args, n) \to (s, vs; (i+1, j), args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{starg}\ j \quad n \neq \mathsf{Exit}\ G \quad s = s_1\ ...\ s_i \quad \mathsf{stack_update}(s, args, (i+1, j), v) = (s, args')}{t, G, h \vdash (s, vs; v, args, n) \to (s, vs, args', \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{ldflda}\ A\ c{::}f \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; ptr, args, n) \to (s, vs; ptr.f, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{stfld}\ A\ c{::}f' \quad n \neq \mathsf{Exit}\ G \quad \mathsf{stack_update}(\mathsf{BIL_stack}(s, args), h, (i,j).\vec{f}.f', v) = (s', args')}{t, G, h \vdash (s, vs; (i,j).\vec{f}; v, args, n) \to (s', vs, args', \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{stfld}\ A\ c{::}f' \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; p.\vec{f}; v, args, n) \xrightarrow{\mathsf{write}\ t\ p.\vec{f}.f'\ v} (s, vs, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{newobj void}\ c{::}\ \texttt{.ctor}(A_1, ..., A_k) \quad n \neq \mathsf{Exit}\ G \qquad c \notin \mathsf{ValueClass} \quad p \in \mathsf{free_set}\ h \quad \mathsf{fields}(c) = f_i \mapsto A_i{}^{i \in 1..n}}{t, G, h \vdash (s, vs; v_1; ...; v_k, args, n) \xrightarrow{\mathsf{write}\ t\ p\ c[f_i \mapsto v_i{}^{i \in 1..n}]} (s, vs; p, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{newobj void}\ vc{::}\ \texttt{.ctor}(A_1, ..., A_k) \quad n \neq \mathsf{Exit}\ G \quad vc \in \mathsf{ValueClass} \quad \mathsf{fields}(c) = f_i \mapsto A_i{}^{i \in 1..n}}{t, G, h \vdash (s, vs; v_1; ...; v_k, args, n) \to (s, vs; (f_i \mapsto v_i{}^{i \in 1..n}), args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{callvirt}\ B\ c{::}\ell(A_1, ..., A_k) \quad n \neq \mathsf{Exit}\ G \quad c'[f_i \mapsto u_i{}^{i \in 1..n}] \in \mathsf{can_read}\ t\ p \quad c' <: c}{t, G, h \vdash (s, vs; p; v_1; ...; v_k, args, n) \xrightarrow{\mathsf{read}\ t\ p\ c'[f_i \mapsto u_i{}^{i \in 1..n}]} (s; (vs, args, n), \cdot, .args(p, v_1, ..., v_k), \mathsf{next}\ G\ (\mathsf{mcall}\ c')\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{call instance}\ B\ vc{::}\ell(A_1, ..., A_k) \quad n \neq \mathsf{Exit}\ G \quad vc \in \mathsf{ValueClass}}{t, G, h \vdash (s, vs; p; v_1; ...; v_k, args, n) \to (s; (vs, args, n), \cdot, .args(p, v_1, ..., v_k), \mathsf{next}\ G\ (\mathsf{mcall}\ vc)\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{ret} \quad n \neq \mathsf{Exit}\ G \quad (n, \mathsf{seq}, n') \in \mathsf{Edges}\ G}{t, G, h \vdash (s; (vs', args', n'), vs; v, args, n) \to (s, vs'; v, args', n')}$$

Figure 4.2: The single-threaded transition semantics of GraphBIL, part 1: control flow, loads and stores, object creation, method calls

$$\frac{\mathsf{Label}\ G\ n = \mathtt{ldind} \quad n \neq \mathsf{Exit}\ G \quad v \in \mathsf{can_read}\ h\ t\ p.\vec{f}}{t, G, h \vdash (s, vs; p.\vec{f}, args, n) \xrightarrow{\mathsf{read}\ t\ p.\vec{f}\ v} (s, vs; v, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \mathtt{stind} \quad n \neq \mathsf{Exit}\ G \quad \mathsf{stack_update}(\mathsf{BIL_stack}(s, args), h, (i, j).\vec{f}, v) = (s', args')}{t, G, h \vdash (s, vs; (i, j).\vec{f}; v, args, n) \to (s', vs, args', \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \mathtt{stind} \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; p.\vec{f}; v, args, n) \xrightarrow{\mathsf{write}\ t\ p.\vec{f}\ v} (s, vs, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

Semantics for `stfld` are derived analogously, and most of the remaining rules can be translated from BIL to GraphBIL as straightforwardly as with `ldc.i4` (note that `starg` always stores to the stack, and `ldarga` and `ldflda` create pointers to arguments/fields rather than directly reading memory). The other instructions that perform shared memory operations are `callvirt`, which must read the referenced object in order to determine its class and make it available to the called method; `newobj`, which when invoked on a reference class creates a new object on the heap (its behavior, in both BIL and GraphBIL, is entirely different when called on a value class and does not involve the heap); and `box`, which takes an instance of a value class and creates a boxed version of it on the heap. The full semantics of GraphBIL is given in Figures 4.2 and 4.3.

$$\frac{\mathsf{Label}\ G\ n = \mathtt{box}\ vc \quad n \neq \mathsf{Exit}\ G \quad vc \in \mathsf{ValueClass} \quad p \in \mathsf{free_set}\ h \quad {f_i \mapsto v_i}^{i \in 1..n} \in \mathsf{can_read}\ t\ ptr}{t, G, h \vdash (s, vs; ptr, args, n) \xrightarrow{\mathsf{read}\ t\ ptr\ v,\ \mathsf{write}\ t\ p\ vc[{f_i \mapsto v_i}^{i \in 1..n}]} (s, vs; p, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \mathtt{unbox}\ vc \quad n \neq \mathsf{Exit}\ G \quad vc \in \mathsf{ValueClass}}{t, G, h \vdash (s, vs; p, args, n) \to (s, vs; p, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \mathtt{dup} \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; v, args, n) \to (s, vs; v; v, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \mathtt{pop} \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; v, args, n) \to (s, vs, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

Figure 4.3: The single-threaded transition semantics of GraphBIL, part 2: boxing and unboxing, evaluation stack manipulation

4.5 Modeling Memory Models

One question that naturally arises when modeling concurrency in languages is that of memory model. While the term "memory model" can be used to refer to various aspects of the way data is stored in memory, in the context of shared-memory concurrency it more specifically refers to the way in which memory operations committed by a set of concurrent threads affect the portion of the memory that is visible to multiple threads. In its simplest form, a memory model is an answer to the question, "what are the values that a memory read

operation can read?" Almost every processor architecture has its own answer to this question, and many have more than one. Well-known memory models include sequential consistency, total store ordering, partial store ordering, and relaxed-memory ordering, among others [53]. Adding to the confusion, many of these models are not *operational*; they are phrased as conditions on total executions, rather than as properties that can be checked in individual steps of an operational semantics.

A common approach to specifying memory models, taken by LLVM [46] and the new C/C++11 standard [7], is to construct a partial order on memory operations appearing in a program, and determine the values that can be read by a read operation from this order. For instance, LLVM and C11 define a *happens-before* relation, produced by combining the ordering of instructions in each thread with constraints induced by explicit synchronization constructs (such as atomic instructions and locking mechanisms). They then require that a read r can see only writes that (1) do not happen after r in the order and (2) do not happen before writes that happen before r. Intuitively, if r must occur before a write w, or w is blocked from r's view by another write w', then r cannot read the value written by w; otherwise, it can. This poses a problem for operational semantics: in any particular interleaving of threads in a program, a read may be allowed to read values that have not yet been written, as long as they will be written by writes that are not later in the happens-before order. If there are runtime checks that can be performed that guarantee correctness with respect to a model, then it can be integrated into an operational semantics, but there are no known operational implementations of most happens-before-style models. However, there are other models that can be simulated by *abstract machines*, transition systems that model the architecture behaviors that originally gave rise to the memory model constraints. If there is an abstract machine that can be run in parallel with a program to produce exactly the executions allowed by a memory model, then this abstract machine can be used to make the memory model operational.

To improve the flexibility and generality of PTRANS as a verification tool, we have developed a general approach to specifying operational concurrent memory models, allowing us to write semantics for programming languages (e.g. MiniLLVM) that are memory-model-agnostic and then specialize them with memory models of our choice (we do not currently handle non-operational memory models). We do this using Isabelle's locale functionality, which allows us to parameterize definitions by certain functions and axioms, and later provide definitions that implement those functions and axioms to obtain concrete instances of the parameterized theories. Our memory model locale specifies an abstract type *memory*, supporting the following four functions:

- can_read, the workhorse of the memory model, which returns the set of values that a thread can see at a given memory location

- free_set, which returns the set of locations that are free in the memory
- start_mem, which gives a (usually empty) initial memory
- update_mem, which updates a memory with a set of memory operations performed by various threads

As a set of minimal constraints on these functions, we require that a memory updated with an alloc t ℓ operation does not have the location ℓ in its free_set, and that ℓ does not return to the free_set until a free operation is performed that includes it.

We define three instances of this locale for use in our examples: sequential consistency (SC), total store ordering (TSO), and partial store ordering (PSO). Sequential consistency is the simplest memory model, and the one usually assumed in a naive implementation of a concurrent language. Formally, it requires that every execution observed could have been produced by some total order on the memory operations in the execution. Operationally, this means that the valid executions of a program are exactly those produced by arbitrary interleavings of the threads of the program, assuming that memory operations execute atomically. In our framework, SC is implemented by a map from memory locations to values and a straightforward implementation of the four required functions. The function can_read returns the value (if any) that exists in the memory map at the given location; free_set returns the set of locations with no values in the map; start_mem is the empty map; and update_mem applies the given memory operations to the map, storing a new value on a write or arw, initializing the location with a starting value on an alloc, and clearing the location on a free.

Start: $\ell_1 \mapsto 0$ and $\ell_2 \mapsto 0$

write ℓ_1 1	write ℓ_2 1
$x :=$ read ℓ_2	$y :=$ read ℓ_1

Result: $x = 0 \wedge y = 0$

Figure 4.4: Behavior forbidden by SC but allowed in TSO

The TSO memory model is slightly more complex. Intended to model optimizations performed in real-world multicore architectures, it allows writes to be delayed past unrelated reads in the same thread, resulting in executions such as the one shown (in pseudocode) in Figure 4.4. Under SC, if one of the read instructions returned 0 in an execution, then we would be forced to conclude that the write instruction in the same thread executed before it, and so the other read would have to read a value of 1. Under TSO, however, the writes may effectively be delayed past the reads, allowing both reads to return 0. As shown by Owens et al. [61], this behavior can be modeled by associating a FIFO *write buffer* with each thread. In this model, a memory is a pair (m, b) of a (shared) map from locations to values and a collection of per-thread write buffers. When a write operation is performed, it is inserted at the head of the executing thread's write buffer; at any

point, the oldest write in any thread's write buffer may be written to the shared memory. When a thread executes a read operation, it first looks for the most recent write to the location in its write buffer, and if none exists then goes on to read from the shared memory. One final detail is that atomic arw operations serve as memory fences: they are not executed until the write buffer of the executing thread has cleared. This model can easily be expressed in terms of our four functions: can_read first searches the write buffer for a value and then checks the memory map, as described; free_set behaves exactly as in SC; start_mem is an empty memory map along with an empty buffer for each thread; and update_mem first executes all memory operations, putting any writes into the corresponding write buffers, and then allows an arbitrary number of writes to move from the write buffers to the shared memory.

Start: $\ell_1 \mapsto 0$ and $\ell_2 \mapsto 0$

write ℓ_1 1	$y :=$ read ℓ_2
write ℓ_2 1	$x :=$ read ℓ_1

Result: $x = 0 \wedge y = 1$

Figure 4.5: Behavior forbidden by SC and TSO but allowed in PSO

The PSO memory model is a further relaxation of TSO, in which writes may be delayed past unrelated writes as well. Figure 4.5 shows an example of the behavior allowed by this relaxation. As in TSO, behaviors of this form can be modeled operationally with write buffers; while in TSO it sufficed to associate a write buffer to each thread, in PSO this is naturally modeled by giving each thread a write buffer for each memory location. This allows writes to different memory locations to be executed out of order, while maintaining the order of writes to the same location. The details of the functionality of the write buffers are identical, and once again arw operations cannot be executed until all the write buffers for the executing thread have cleared.

It may be possible to verify some PTRANS optimizations, particularly those that do not involve memory in any way, without specifying the memory model. In this case, we can prove them correct using the generic memory model locale, and then automatically obtain correctness results for SC, TSO, PSO, and any other memory model specified in our framework. However, in practice, many interesting optimizations will involve memory, and their correctness will depend on the memory model being used. In fact, one of the primary purposes of relaxed memory models (i.e., non-SC models) is to allow a wider range of optimizations. We can prove correctness of an optimization with respect to a particular memory model by instantiating the semantics of the target language (e.g. MiniLLVM) with the chosen model, and this proof then provides no information about the correctness of the optimization under any other memory model. In general, some memory models are strictly more permissive than others; for instance, we have proved that every execution

produced by MiniLLVM under SC can also be produced under TSO. However, depending on our notion of correctness, it may not be immediately clear that every valid SC optimization is also a valid TSO optimization – for instance, an SC optimization may rely on the correctness of a locking mechanism that only functions properly under SC. Determining exactly when proofs of correctness under one model can be extended to other models is an interesting area for future work.

Chapter 5

Executable Semantics and Testing

5.1 From Specification Language to Design Tool

The traditional purpose of a specification is to serve as an abstract logical description of the intended behavior of a program. A specification abstracts away from implementation details, and is written for clarity over efficiency. Ideally, it is constructed in a framework that gives it precise mathematical semantics, so that its meaning is as precise as possible. Given such a specification, we can prove that a program (written in a messier, more complicated implementation language) implements the specification, often by showing that every behavior of the program is allowed by the specification, and conclude that the implementation possesses the desirable properties ensured by the specification (correctness, memory safety, security, etc.).

However, from a usability perspective, there are flaws in this approach. The specification must be taken as the definitive formulation of the program's correctness. We may expect certain properties to hold on the specification, and even prove that they hold, but we can never come up with a set of properties that completely characterizes the desired behavior of the specification – to do so would be to reproduce the specification itself. The chain of verified correctness must stop at some point, and at that point, we must rely on non-formal methods to assure us of the correctness of our specification. From this perspective, the view of a specification as an abstract formal object is worrying, because abstract formal objects are easy to write incorrectly, and often have implications that are not obvious to human intuition. PTRANS is no exception – we may complete the proof of correctness of a PTRANS optimization, but this provides no guarantee that the correct optimization is the one the designer meant to write. In the worst case, the optimization might be vacuously correct, because it never transforms any program.

Because of this, there has been a trend towards *executable specifications* [19], allowing traditional software testing methods to be applied to specifications. This both allows us to catch errors in specifications before using them as proof targets, and helps us improve our intuitive understanding of the specifications and the behaviors they describe. Applied to PTRANS, this means the ability to run PTRANS transformations on actual programs, and then either inspect the original and transformed programs to get a sense of the

behavior of the optimization or, still better, run the two programs and observe any difference in their output. In the remainder of this chapter, we will describe the executable semantics of PTRANS, and demonstrate its use in designing, testing, and correcting optimizations.

5.2 Exploration via Executable Semantics

Recall that PTRANS specifications in general break down into two components: a rewrite to be performed and a CTL side condition to be checked. The executable semantics of PTRANS must provide semantics for each of these components. The first is relatively straightforward, since the abstract semantics of atomic rewrites is already nearly executable. The second component requires a general method for determining whether and where a CTL formula holds on a given graph, which is less straightforward. In the semantic function for transformations, we quantify over all substitutions that satisfy the side conditions of a transformation, a mathematical concept that is distinctly non-executable. In this chapter we will give a more directly executable semantics for transformations, based on a method for computing the satisfying substitutions of a CTL side condition.

5.2.1 CTL Model Finding

The model checking problem for CTL and its variants is a well-studied problem with a well-known efficient algorithm [15]. Considerably less attention has been given to the closely related problem of *model finding*. The model finding problem in its general form is the following: suppose we have a first-order CTL formula φ built from a set of atomic predicates, where the predicates may contain free variables. Given a transition system $\mathcal{C}$ and an interpretation of the atomic predicates on $\mathcal{C}$, what are the possible assignments of values to the free variables appearing in φ such that φ holds on $\mathcal{C}$? When a formula contains no free variables, model finding is simply model checking; however, in the general case, model finding is considerably more complex.

In prior work in collaboration with the author [49], Griffith developed the algorithm that is used in the executable semantics of PTRANS. Bohn et al. [12] gave several algorithms for model-checking FOCTL formulae; it is their *syntactic* model-checking approach that forms the basis of the algorithm. The syntactic procedure reduces an FOCTL formula φ on a given system to a non-temporal first-order formula P such that the satisfying models of P are exactly the satisfying models of φ. We can then use a satisfiability modulo theories (SMT) solver, such as Z3 [58], to find the satisfying models of P. The main complexity of the algorithm comes in dealing with the temporal until-operators, and it relies on a family of helper functions to reduce the until-operators to non-temporal formulae. The function $\textsc{Paths}_{\leftarrow}(I, F, n, v)$ calculates a formula

characterizing the set of models such that there is a path of length n or less from v along which I holds until F holds; $\textsc{Paths}_{\wedge}(I, F, n, v)$ asserts that along all paths of length n or less from v, I holds until F holds; $\textsc{Paths}_{\rightarrow}(I, n, v)$ asserts that there exists a path of length exactly n along which I holds at every point; and $\textsc{Paths}_{\overline{\leftarrow}}(I, F, n, v)$ and $\textsc{Paths}_{\overline{\wedge}}(I, F, 0, v)$ assert respective properties on paths backwards from v. These functions are specified by a set of recursive equations, in which we assume we have fixed a set E of edges in the graph under consideration:

$$\textsc{Paths}_{\leftarrow}(I, F, 0, v) = F(v)$$

$$\textsc{Paths}_{\leftarrow}(I, F, n, v) = \textsc{Paths}_{\leftarrow}(I, F, n-1, v) \vee \left(I(v) \wedge \bigvee_{v' \in \textsc{Succ}(E,v)} \textsc{Paths}_{\leftarrow}(I, F, n-1, v') \right)$$

$$\textsc{Paths}_{\wedge}(I, F, 0, v) = F(v)$$

$$\textsc{Paths}_{\wedge}(I, F, n, v) = \textsc{Paths}_{\wedge}(I, F, n-1, v) \vee \left(I(v) \wedge \bigwedge_{v' \in \textsc{Succ}(E,v)} \textsc{Paths}_{\wedge}(I, F, n-1, v') \right)$$

$$\textsc{Paths}_{\rightarrow}(I, 0, v) = \textsc{True}$$

$$\textsc{Paths}_{\rightarrow}(I, n, v) = \bigvee_{v' \in \textsc{Succ}(E,v)} (I(v') \wedge \textsc{Paths}_{\rightarrow}(I, n-1, v'))$$

$$\textsc{Paths}_{\overline{\leftarrow}}(I, F, 0, v) = F(v)$$

$$\textsc{Paths}_{\overline{\leftarrow}}(I, F, n, v) = \textsc{Paths}_{\overline{\leftarrow}}(I, F, n-1, v) \vee \left(I(v) \wedge \bigvee_{v' \in \textsc{Pred}(E,v)} \textsc{Paths}_{\overline{\leftarrow}}(I, F, n-1, v') \right)$$

$$\textsc{Paths}_{\overline{\wedge}}(I, F, 0, v) = F(v)$$

$$\textsc{Paths}_{\overline{\wedge}}(I, F, n, v) = \textsc{Paths}_{\overline{\wedge}}(I, F, n-1, v) \vee \left(I(v) \wedge \bigwedge_{v' \in \textsc{Pred}(E,v)} \textsc{Paths}_{\overline{\wedge}}(I, F, n-1, v') \right)$$

where $\textsc{Succ}(E, n)$ is the set of nodes v' such that $(v, \ell, v') \in E$ for some ℓ, and $\textsc{Pred}(E, v)$ is the set of nodes v' such that $(v', \ell, v) \in E$ for some ℓ. Given these definitions, the algorithm itself is given by the following

set of recursive equations, in which we assume we have fixed a set N of nodes in the graph:

$$\textsc{Satis}(p(\vec{x}))(v) = p(\vec{x})$$

$$\textsc{Satis}(\varphi_1 \wedge \varphi_2)(v) = \textsc{Satis}(\varphi_1)(v) \wedge \textsc{Satis}(\varphi_2)(v)$$

$$\textsc{Satis}(\neg\varphi)(v) = \neg\textsc{Satis}(\varphi)(v)$$

$$\textsc{Satis}(\exists x.\ \varphi)(v) = \exists x.\ \textsc{Satis}(\varphi)(v)$$

$$\textsc{Satis}(E\ \varphi_1\ \mathcal{U}\ \varphi_2)(v) = \textsc{Paths}_{\leftarrow}(\textsc{Satis}(\varphi_2), \textsc{Satis}(\varphi_1), |N|, v)$$

$$\textsc{Satis}(A\ \varphi_1\ \mathcal{U}\ \varphi_2)(v) = \neg\textsc{Paths}_{\rightarrow}(\textsc{Satis}(\varphi_1 \wedge \neg\varphi_2), |N| + 1, v) \wedge\ \textsc{Paths}_{\wedge}(\textsc{Satis}(\varphi_2), \textsc{Satis}(\varphi_1), |N|, v)$$

$$\textsc{Satis}(E\ \varphi_1\ \mathcal{B}\ \varphi_2)(v) = \textsc{Paths}_{\overline{\leftarrow}}(\textsc{Satis}(\varphi_2), \textsc{Satis}(\varphi_1), |N|, v)$$

$$\textsc{Satis}(A\ \varphi_1\ \mathcal{B}\ \varphi_2)(v) = \textsc{Paths}_{\overline{\wedge}}(\textsc{Satis}(\varphi_2), \textsc{Satis}(\varphi_1), |N|, v)$$

The following theorem states the correctness of the algorithm:

Theorem 3. *Let $\mathcal{G} = (N, E, s, x, L)$ be a CFG, $v \in N$ and φ a First-Order CTL formula. Then* $\{\sigma \mid \mathcal{G}, \sigma, v \vDash \varphi\} = \{\sigma \mid \sigma \vDash \textsc{Satis}(\varphi)(v)\}$.

Proof. (due to Griffith) The proof proceeds by induction on the lexicographic ordering of (1) the number of occurrences of the $A\mathcal{U}$ operator in φ and (2) the size of φ. Since each recursive call in the definition of Satis is on a formula lower in this order that the original formula, it suffices to show that each of the equations for Satis translates its argument correctly, given the assumption that each recursive call is correct. In the cases for atomic predicates, conjunction, negation, and the existential quantifier, this is straightforward. Then for each temporal operator, we must show that the relevant Paths function(s) characterize exactly the set of substitutions that satisfy the temporal operator. These cases rely on one key observation: that if there exists a path through $\mathcal{G}$ (forward or backward) along which φ_1 holds until φ_2 holds, then there exists a cycle-free path along which φ_1 holds until φ_2 holds. The maximum length of a cycle-free path is bounded by $|N|$, so there exists a path along which φ_1 holds until φ_2 holds if and only if there exists such a path of length no more than $|N|$.

We take this observation together with correctness results for each of the Paths functions, each of which follows by straightforward induction on the path length n:

- $\textsc{Paths}_{\leftarrow}(I, F, n, v)$ characterizes the set of substitutions σ such that there is a path λ from v of length $k \leq n$ along which $\mathcal{G}, \sigma, \lambda_k \vDash F(\lambda_k)$ and $\mathcal{G}, \sigma, \lambda_i \vDash I(\lambda_i)$ for all $i < k$.

- $\textsc{Paths}_{\wedge}(I, F, n, v)$ characterizes the set of substitutions σ such that for every path λ from v that is

of length n or reaches the exit node in fewer than n steps, there is some i where $\mathcal{G}, \sigma, \lambda_i \models F(\lambda_i)$ and $\mathcal{G}, \sigma, \lambda_i \models I(\lambda_j)$ for all $j < i$.

- $\text{PATHS}_{\rightarrow}(F, n, v)$ characterizes the set of substitutions σ such that there is a path λ from v of length n along which $\mathcal{G}, \sigma, \lambda_i \models I(\lambda_i)$ for every i.

- $\text{PATHS}_{\leftharpoondown}(I, F, n, v)$ characterizes the set of substitutions σ such that there is a path λ backward from v of length $k \leq n$ along which $\mathcal{G}, \sigma, \lambda_k \models F(\lambda_k)$ and $\mathcal{G}, \sigma, \lambda_i \models I(\lambda_i)$ for all $i < k$.

- $\text{PATHS}_{\overline{\wedge}}(I, F, n, v)$ characterizes the set of substitutions σ such that for every path λ backward from v that is of length n or reaches the start node in fewer than n steps, there is some i where $\mathcal{G}, \sigma, \lambda_i \models F(\lambda_i)$ and $\mathcal{G}, \sigma, \lambda_i \models I(\lambda_j)$ for all $j < i$.

This gives us all we need to prove the correctness of the temporal cases. The $E\mathcal{U}$ case follows directly from the correctness of $\text{PATHS}_{\leftarrow}$ and our observation. In the $A\mathcal{U}$ case, $\text{PATHS}_{\wedge}$ gives us that $A\varphi_1\mathcal{U}\varphi_2$ holds on all finite paths from v, but we must also check that there are no infinite counterexamples, i.e., no cycles that can be followed to produce a path along which φ_1 always holds but the point at which φ_2 holds is never reached. We do this by using $\text{PATHS}_{\rightarrow}$ to check that there is no path of length $|N|$ along which φ_1 always holds but φ_2 never holds. The $\mathcal{B}$ cases also follow directly from the correctness of the PATHS functions and our observation; in the $A\mathcal{B}$ case, note that we do not need to check for cycles, since by definition all backward paths eventually reach the start state. We can then conclude that the satisfying substitutions σ of $\text{SATIS}(\varphi)(v)$ are exactly the substitutions such that $\mathcal{G}, \sigma, v \models \varphi$. □

By memoizing the partial results of our functions, this algorithm can be made to run in $O(|\varphi|\,|N|^2)$ time. When working on a tCFG, we can reduce it to a single graph by taking the cross product of its component graphs, so that $|N|$ is actually the product of the number of graphs in each thread. Note that this is the complexity of the reduction from FOCTL to non-temporal FOL, rather than of actually finding the relevant models; depending on the target language and the atomic predicates, finding satisfying models of an FOL formula may be undecidable, although in practice SMT solvers are often quite fast even on instances of theoretically undecidable problems.

Once $\text{SATIS}(\varphi)$ has been computed, if the basic language of atomic predicates is amenable to SMT solving, we can use such a solver to compute the set of satisfying models $\{\sigma \mid \mathcal{G}, \sigma, v \models \varphi\}$. More concretely, we can define a function $\mathsf{get_models}(\tau, \mathcal{G}, \varphi)$ that computes the satisfying models of φ by computing $\text{SATIS}(\varphi)$ starting at the Start node of each graph in the tCFG, conjoining it with a formula describing the already-known substitution τ, and then using an SMT solver to find all satisfying models of that formula. Theorem 3

then assures us that $\mathsf{get_models}(\tau, \mathcal{G}, \varphi) = \{\sigma \mid \mathcal{G}; \sigma \vDash \varphi \wedge \sigma|_{\mathrm{dom}(\tau)} = \tau\}$, and so get_models serves as an executable method for finding satisfying models of PTRANS side conditions.

5.2.2 Executable Semantics for Strategies

Using the get_models function, we can write an executable function trans_sf that computes the semantics of a transformation. First, let us review the semantics of actions and transformations. The semantic function for actions is:

$$[\![\mathsf{add_edge}(n, m, \ell)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t, E_t \cup \{(\sigma(n), \sigma(\ell), \sigma(m))\}, s_t, x_t, L_t))$$

$$[\![\mathsf{remove_edge}(n, m, \ell)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t, E_t - \{(\sigma(n), \sigma(\ell), \sigma(m))\}, s_t, x_t, L_t))$$

$$[\![\mathsf{replace}\ n\ \mathsf{with}\ ]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t - \{\sigma(n)\}, E_t - \{(a, \ell, b) \mid a = \sigma(n) \vee b = \sigma(n)\}, s_t, x_t, L_t))$$

$$[\![\mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t \cup \{n_2, ..., n_m\}, \{\mathsf{remap_succ}(\sigma(n), n_m, e) \mid e \in E\} \cup$$
$$\{(n_i, \mathsf{seq}, n_{i+1}) \mid 1 < i < m\}, s_t, x_t, L_t + (n_1 \mapsto \sigma(p_1), ..., n_m \mapsto \sigma(p_m)))$$

where $n_1 = \sigma(n)$ and $n_2, ..., n_m$ are new nodes not in $\mathcal{G}$

$$[\![\mathsf{split_edge}(n, m, \ell, p)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t \cup \{n'\},$$
$$E_t - \{(\sigma(n), \sigma(\ell), \sigma(m))\} \cup \{(\sigma(n), \sigma(\ell), n'), (n', \mathsf{seq}, \sigma(m))\},$$
$$s_t, x_t, L_t + (n' \mapsto \sigma(p))))$$ where n' is a new node not in $\mathcal{G}$

Assuming that CFGs are finite (which will be the case in any real-world program), this definition is already executable, as long as we have a suitable executable representation of sets (such as exists in most modern programming languages). The semantic function for transformations is:

$$[\![A_1, ..., A_m \text{ if } \varphi]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \exists \sigma.\ \sigma|_{\mathrm{dom}(\tau)} = \tau \wedge \mathcal{G}, \sigma \models \varphi \wedge \mathcal{G}' = [\![A_1, ..., A_m]\!](\sigma, \mathcal{G})\}$$

$$[\![\text{MATCH } \varphi \text{ IN } T]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \exists \sigma.\ \sigma|_{\mathrm{dom}(\tau)} = \tau \wedge \mathcal{G}, \sigma \models \varphi \wedge \mathcal{G}' \in [\![T]\!](\tau + \sigma|_{\mathrm{fv}(\varphi)}, \mathcal{G})\}$$

where $\mathrm{fv}(\varphi)$ is the set of free variables of φ

$$[\![T_1 \text{ THEN } T_2]\!](\tau, \mathcal{G}) = \bigcup_{\mathcal{G}' \in [\![T_1]\!](\tau, \mathcal{G})} [\![T_2]\!](\tau, \mathcal{G}')$$

$$[\![T_1 \,\square\, T_2]\!](\tau, \mathcal{G}) = [\![T_1]\!](\tau, \mathcal{G}) \cup [\![T_2]\!](\tau, \mathcal{G})$$

$$[\![\text{APPLY_ALL } T]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \mathsf{apply_some}([\![T]\!], \tau, \mathcal{G}, \mathcal{G}')\} - \{\mathcal{G}' \mid \exists \mathcal{G}'' \neq \mathcal{G}'.\ \mathcal{G}'' \in [\![T]\!](\tau, \mathcal{G}')\}$$

As mentioned above, this definition is decidedly not executable; in two equations we quantify over the satisfying models of a formula, and apply_some may not be finite. The following is an executable version of the semantic function for transformations:

$$\mathsf{trans_sf}(A_1, ..., A_k \text{ if } \varphi, \tau, \mathcal{G}) = \text{map } (\lambda\sigma.\ [\![A_1, ..., A_k]\!](\sigma, \mathcal{G}))\ (\mathsf{get_models}(\tau, \mathcal{G}, \varphi))$$

$$\mathsf{trans_sf}(\text{MATCH } \varphi \text{ IN } T, \tau, \mathcal{G}) = \text{map } (\lambda\sigma.\ \mathsf{trans_sf}(T, \sigma, \mathcal{G}))\ (\mathsf{get_models}(\tau, \mathcal{G}, \varphi))$$

$$\mathsf{trans_sf}(T_1 \text{ THEN } T_2, \tau, \mathcal{G}) = \bigcup_{\mathcal{G}' \in \mathsf{trans_sf}(T_1, \tau, \mathcal{G})} \mathsf{trans_sf}(T_2, \tau, \mathcal{G}')$$

$$\mathsf{trans_sf}(T_1 \,\Box\, T_2, \tau, \mathcal{G}) = \mathsf{trans_sf}(T_1, \tau, \mathcal{G}) \cup \mathsf{trans_sf}(T_2, \tau, \mathcal{G})$$

$$\mathsf{trans_sf}(\text{APPLY_ALL } T, \tau, \mathcal{G}) = \text{let } R = \mathsf{trans_sf}(T, \tau, \mathcal{G}) \text{ in}$$

$$\text{if } R = \{\mathcal{G}\} \text{ then } R \text{ else } \bigcup_{\mathcal{G}' \in R} \mathsf{trans_sf}(\text{APPLY_ALL } T, \tau, \mathcal{G}')$$

In order to define trans_sf as an executable function, we must give up on faithfully representing infinite results. In particular, our algorithm's treatment of the APPLY_ALL strategy does not have exactly the same semantics as $[\![\text{APPLY_ALL}]\!]$. In the abstract semantics, we used apply_some to describe the set of results produced by applying a transformation T some finite number of times, and subtracted the result graphs that could still be further transformed; if T could transform a graph $\mathcal{G}$ indefinitely, the sequence of graphs produced by this infinite chain of rewrites will not appear in $[\![\text{APPLY_ALL } T]\!](\tau, \mathcal{G})$. The trans_sf function, on the other hand, will attempt to apply T to $\mathcal{G}$ indefinitely, and so will never terminate. However, in all finite cases it can be shown that $\mathsf{trans_sf}(T, \tau, \mathcal{G}) = [\![T]\!](\tau, \mathcal{G})$, and so trans_sf is a viable executable semantics for PTRANS transformations.

This gives us an algorithm for computing the result graphs for a given transformation, which can be implemented in a functional language (we have chosen F# for its Z3 integration; more on the implementation in Section 5.4). As long as a transformation expressed in PTRANS does not require infinite computations, we can run it on a target graph and obtain all of its outputs. In the following section, we will demonstrate the use of these semantics to define, test, and refine a sample optimization.

5.3 Designing and Prototyping Optimizations in PTRANS

In this section, we will develop an optimization in PTRANS and show how its executable semantics can be of use in the process of designing optimizations. We will use as our target language the MiniLLVM intermediate language defined in Section 4.3. Our case study will be a *redundant store elimination* optimization (RSE),

which removes `store` instructions that may be overwritten before they are used, as shown in Figure 5.1. Note that the redundant store is replaced by an `is_pointer` instruction, rather than being eliminated entirely, to ensure that crashes are not delayed in bad executions in which e_2 is not a pointer-valued expression.

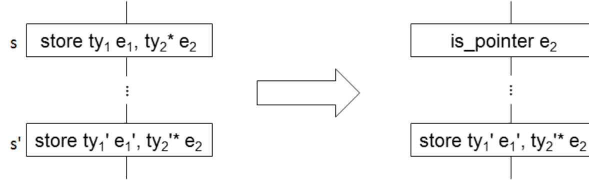


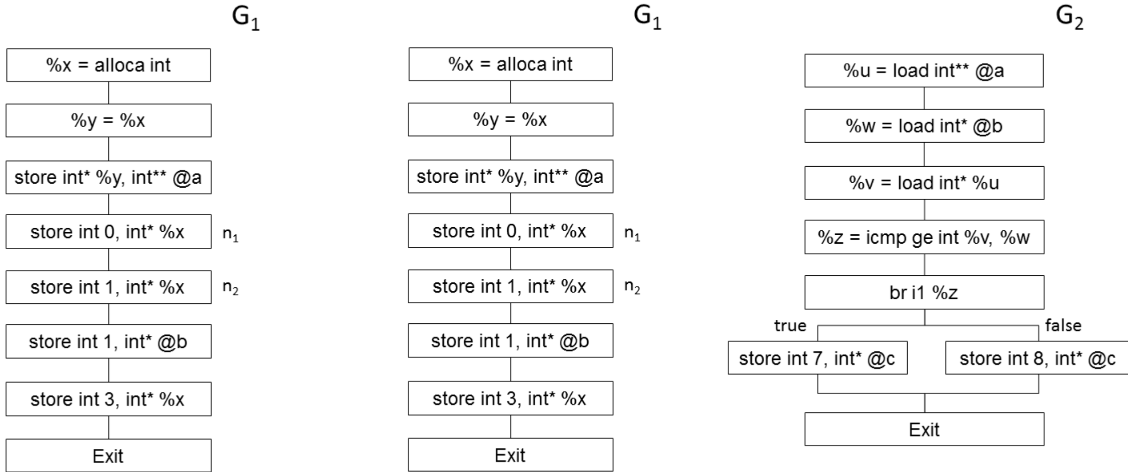
Figure 5.1: Redundant Store Elimination

In the rewrite portion, we will replace the instruction at the chosen node with the `is_pointer` instruction. The side condition should require that there is a node n containing the store to be eliminated, and that along all paths forward from n another store occurs that makes n redundant. To make the optimization safe, we must also require some property to hold on the instructions between n and the following store. For instance, if the value of e_2 is changed before the next store to e_2, removing the store at n will change the behavior of the program. Let φ be some restriction on the types of instructions that can appear between n and the stores that make it redundant; then the PTRANS specification of RSE can be written as:

$$
\begin{aligned}
RSE(\varphi) \triangleq\ & \textsf{replace } n \textsf{ with } \texttt{is_pointer}\ e_2 \textsf{ if} \\
& \text{EF } \textsf{node}_t(n) \land \textsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e_2) \land \\
& \quad A\ \varphi\ \mathcal{U}\ (\neg\textsf{node}_t(n) \land \textsf{stmt}_t(\texttt{store}\ ty_1'\ e_1', ty_2'^*\ e_2))
\end{aligned}
$$

To finish this definition, we must find a suitable value for φ. The most precise form of RSE would involve using alias analysis to determine whether memory operations may, must, or cannot refer to the location indicated by e_2 at n. For the purposes of this example, we will instead give a conservative approximation of the necessary condition, one that guarantees the safety of the transformation but may miss some redundant stores. First, we will need to require that the value of e_2 is not changed, so that we know that successive stores to e_2 do in fact overwrite the store at n; we can do this through the use of a defined def predicate describing all the instructions that might redefine a variable (recall that MiniLLVM expressions are either constants, or local or global variables). The predicate $\textsf{def}_t(e)$ is true in a state in which the instruction being executed by t will change the value of e. We will also need to place some restriction on the kinds of memory operations that can be performed between n and a following store; after all, if the value stored to e_2 is used in any way before being overwritten, the store is not redundant. In the absence of alias analysis, we must assume that any reference to a memory location could overlap with e_2, so our condition must rule

out any `load` instructions between n and a following store. We can define a predicate not_loads such that $\mathsf{not_loads}_t(e)$ is true when the instruction in t is not a load from e, and then write the remainder of our side condition as $\varphi_1 \triangleq \neg\mathsf{def}_t(e_2) \wedge \forall e.\ \mathsf{not_loads}_t(e)$.



(a) A graph with two redundant stores (b) A tCFG with redundant stores?

Figure 5.2: An RSE example

Using our executable semantics, we can run $RSE(\varphi_1)$ on a range of example CFGs, such as the graph G_1 shown in Figure 5.2a. The program in G_1 initializes a local pointer %`x`, creates an alias to it in %`y` and publicizes its location in the global variable @`a`, and then performs a series of stores to shared memory. The trans_sf function will give us two possible results for $RSE(\varphi_1)$ on G_1, one in which each of n_1 and n_2 is replaced by an `is_pointer` instruction (we could also use APPLY_ALL $RSE(\varphi_1)$ to apply the transformation repeatedly, replacing both n_1 and n_2). Furthermore, running each of the transformed programs shows that they produce the same results as the original program: 0 at the location of %`x`, the value of %`x` at @`a`, and 1 at @`b`. Thus far, φ_1 appears to be a sufficient condition to ensure the correctness of RSE, and this condition is indeed sufficient for single-threaded programs.

However, when we expand our aims to parallel programs, a potential error becomes apparent. Consider the tCFG in Figure 5.2b. Although the program is not well synchronized, we can see that the false branch in G_2 will never be taken (in a sequentially consistent execution), since if we successfully read the value at @`b` into %`w`, a value greater than or equal to 1 will have already been stored to %`x`. However, if the store at n_2 is removed, then we may reach a state in which %`w` is 1 and %`v` is 0, allowing the value 8 to be stored in @`c`. This means that $RSE(\varphi_1)$ will introduce new observable behaviors in the tCFG: in the original graph the final value of the global variable @`c` is always 7, but in the transformed graph it may be 8. Correct optimizations may rule out some executions (for instance, by optimizing away an outcome of a

race condition), but they should never introduce new behavior. Thus, this test case shows that we need to tighten the condition on our RSE optimization to make it safe on parallel programs.

The simplest refinement is to disallow any changes to shared memory between a store to be removed and its following stores. In the example above, if the store to `@b` in G_1 did not exist, then it would be impossible for G_2 to distinguish between the case in which the store at n_2 was removed and the one in which it had already been overwritten by the final store to `%x`. Since we have already ruled out `load` instructions, we need only prohibit `store` instructions as well; the appropriate side condition in PTRANS can be written as $\varphi_2 \triangleq \neg\mathsf{def}_t(e_2) \wedge ((\forall e.\ \mathsf{not_loads}_t(e) \wedge \mathsf{not_stores}_t(e)) \vee \mathsf{node}_t(n))$, where we add a special case to allow for the possibility of looping back through n before reaching the following store. Running $\mathsf{trans_sf}$ on $RSE(\varphi_2)$ will then remove the store at n_1, but leave n_2 untouched. We can run the resulting program and see that, as desired, the transformed program will never produce a value of 8 in `@c`. Through the process of iterated testing and refinement, we have produced an apparently correct form of the RSE optimization on parallel programs – although, if later tests show φ_2 to be insufficient to ensure correctness, we can repeat the process and devise a still stronger condition.

5.4 Implementation

We have implemented the executable semantics of PTRANS described above in F#, taking advantage of its integration with the Z3 SMT solver [58]. The semantic functions for actions and strategies in Section 5.2.2 can be straightforwardly translated into F# code. We use the algorithm of Section 5.2.1 to reduce side conditions to first-order formulae that can be passed to Z3, and make repeated calls to Z3 to get all the satisfying models, in each iteration adding a condition that rules out the previous model. We memoize the Satis function with a standard lookup table in order to achieve the desired running time. The examples of Section 5.3 complete in between 1 and 4 seconds, with the majority of the running time devoted to constructing the SMT queries; we believe that further optimization of the condition-generation process will allow the semantics to scale to more extensive program graphs.

Recall from Section 3.1.1 that one of the primary design goals of PTRANS was to serve as a *language-independent* framework. In order to instantiate PTRANS to a particular target language, we have seen that we must provide a type of instruction patterns for that language, as well as defining substitution of program objects for metavariables on those patterns to produce concrete instructions. If we are to use Z3 to check atomic predicates such as $\mathsf{stmt}_t(p)$, we must also define a translation from instructions/instruction patterns to Z3 terms, and vice versa. In our F# implementation, however, we have managed to completely

eliminate all of this preparatory effort, by taking advantage of reflection and the capabilities of Z3. Patterns and substitution are builtin features of Z3, so the core problem is the automated translation of F# program objects to and from Z3.

In each run of the executable semantics, we construct a *type table* mapping F# types to Z3 sorts. The table is filled in by need: each time we encode an F# object o in Z3, we look up the type of o in the type table, and if no corresponding sort exists we create one dynamically. Some types have built-in translations (integers, for instance, are translated directly to Z3 integers), but program syntax (expressions, instructions, etc.) usually requires the construction of a new sort. In our implementation we assume that program syntax is defined using one or more algebraic (disjoint union) datatypes; we can then use F# reflection to learn the constructors of the type under consideration, and define a new Z3 datatype sort with the same set of constructors. If the constructors of the object have arguments, we recursively generate Z3 sorts for those arguments as well. The F# syntax of PTRANS includes basic structures for inserting metavariables into terms, via the types `literal` and `expr_pattern`, and the constructors of these types are ignored when constructing Z3 sorts, since Z3 allows variables in its terms by default. The last type to be defined is that of program variables, which are associated with strings in F#. In general, PTRANS transformations should not introduce new program variables, so our get_models function must restrict Z3 to finding models in which program-variable-valued metavariables are associated with variables that exist in the program being transformed. We accomplish this by constructing the sort of program variables as an enumeration, with each variable appearing free in the target program as a case of the enumeration, and assigning this sort to program-variable-valued metavariables.

Once the sort for an F# type has been defined, we can begin to translate objects of that type between F# and Z3. We define two translation functions from F# from Z3: `convert_p` converts F# patterns, and `convert_i` converts concrete terms. The primary difference between the two functions is their treatment of variables: `convert_p` interprets variable strings as metavariables and converts them into Z3 variables, while `convert_i` interprets them as program variables and converts them into members of the corresponding Z3 enumeration sort. Translation from Z3 to F# proceeds similarly, though we only attempt to recover concrete terms: a function `recover_i` performs a reverse lookup in the type table to find the appropriate F# type for the Z3 sort of its argument, and then recursively translates the Z3 term constructor-by-constructor into an F# object of that type. Once these functions have been defined, substitution is trivial; given a Z3 model σ and a F# pattern p, we can define generalized substitution as $\sigma(p) = \texttt{recover_i}(\sigma(\texttt{convert_p}(p)))$. While such an approach may not be feasible for verification (for instance, reflection often introduces type safety concerns), this minimizes the effort required to use the executable semantics on a new language, making it

more readily accessible to users without detailed knowledge of substitution or Z3.

5.5 Usability

One of the greatest criticisms of past formal verification efforts and tools has been the high barrier to entry. It has been argued that traditional interactive formal tools (including proof assistants such as Isabelle) are too complex and too heavyweight to be used by non-specialists [72]. The design of PTRANS in general, and its executable semantics in particular, are intended to significantly improve the usability of the system for non-experts. While verification may still be an arduous process (and in Chapter 6, we will describe some of our efforts to reduce this burden), the design and specification of optimizations in PTRANS is quite accessible. Compiler writers are not simply forced to learn a new domain-specific language and then write correct transformations in it; they are able to plug in a target language of their choice and then play with their definitions, executing prospective optimizations on sample code and even executing that sample code to see how the optimizations affect it.

As part of the VeriF-OPT project, we asked two undergraduate students with a basic background in compiler design but no particular experience with temporal logic to write optimizations in PTRANS. They began with Kalvala et al.'s paper [28] on the original TRANS language, and worked to adapt two of their optimizations (dead assignment elimination and constant propagation) to PTRANS on MiniLLVM. After 2-3 months, they had successfully produced executable PTRANS optimizations. In the process, they produced small test suites that highlighted errors in early versions of their specifications, and used these tests to refine their conditions (as in our contrived example). We believe that the specify-test-revise-verify workflow offers usability that is uncommon in existing formal systems, particularly ones involving compiler correctness.

Chapter 6

Optimization Verification

In this chapter, we will turn our attention to the problem of verifying optimizations expressed in PTRANS. In the process, we will see the advantages of the temporal logic approach for verification, and develop some theory that can be reused in future correctness proofs.

6.1 Defining Correctness

The semantics of a PTRANS specification, and in general of a compiler transformation, can be expressed denotationally in terms of the program graphs that may be produced as a result of the transformation on a given input graph. That is, given a transformation T, a program graph (CFG or tCFG) G, and a valuation σ mapping the metavariables appearing in T to graph components, we can define the semantics of T on G and σ in terms of a set of output graphs, which we write as $[\![T]\!](G, \sigma)$. We can then call T a "correct transformation" if, for any graph G and valuation σ, every output graph in $[\![T]\!](G, \sigma)$ has some desired property relative to G. In previous work, we called two (single-threaded) programs *semantically equivalent* if they produced the same set of possible results when run to completion. However, this is already problematic in the sequential case, and becomes even more so when dealing with parallel programs, which may exhibit complicated interaction with outside processes.

A more nuanced sense of correctness is *observational refinement*: we say that a graph G' observationally refines a graph G if any observable behavior of G' is also an observable behavior of G. This approach comes out of the work of Hoare [25] and has since been used in many process algebras and programming models. Various approaches to observational refinement/equivalence have been defined for various purposes; see for instance von Glabbeek's survey of process semantics [77], or Leroy's examination of definitions of compiler correctness [41]. Among these, simulation and bisimulation [24] and their variants allow us to demonstrate observational refinement (in terms of a set of observable events) by relating corresponding states in the two systems under consideration, and subsume most other notions of equivalence.

Definition 4. *A* simulation *is a relation $\preceq$ on two labeled transition systems P and Q such that for any*

states p, p' *of* P *and* q *of* Q*, for any label* k*, if* $p \preceq q$ *and* $p \xrightarrow{k}_P p'$*, then* $\exists q'.\ q \xrightarrow{k}_Q q'$ *and* $p' \preceq q'$*. By abuse of notation, we write* $P \preceq Q$*, and say that* Q *simulates* P*.*

It can be shown that if Q simulates P (under some notion of observable behavior), then the possible observable behaviors of P are a subset of those of Q. A *bisimulation* is the symmetric analogue to a simulation, requiring that each system be able to simulate any move made by the other (under a single relation), and providing a very strong notion of observational equivalence: if P and Q are bisimilar then they have exactly the same possible observable behaviors. Our Isabelle formulation of simulation and its properties is adapted from the work of Lochbihler on JinjaThreads [47].

Depending on the optimization being verified, various notions of observable behavior and simulation may be appropriate for definitions of correctness. For imperative languages, the changes made to the program state in each transition are an obvious first choice (up to some amount of information hiding; internal variables, for instance, may not be observable from the perspective of the outside world). Some optimizations can be expected to preserve the behavior of a program exactly; others may eliminate possible (but undesirable) behaviors, for instance by reducing the nondeterminism of programs with race conditions, and so we might only expect to show that the original program simulates the transformed graph. Still others might produce programs that have the same observable behavior, but perform more or fewer internal steps in between observable transitions; this is the notion of *weak* or *stuttering* simulation. At a first approximation, we can say that the visible portion of a program's execution state is the assignment of values to globally visible variables, and call T correct if, for any input graph G and all output graphs G' produced by applying T to G, G simulates G'. In this way, we can guarantee that for each (possibly infinite) execution of the transformed program, there exists an observationally equivalent execution of the input program (and, in the case of bisimulation, vice versa); i.e., any behavior of the optimized program was a possible behavior of the original program. Thus we can obtain a strong guarantee of correctness for a transformation.

6.1.1 PTRANS and Simulation

In order to define simulation on tCFGs, we must have a way of interpreting tCFGs as labeled transition systems. In Section 4.2.2, we gave a method of deriving an unlabeled transition relation $\mathcal{G} \vdash (\mathit{states}, m) \rightarrow (\mathit{states}', m')$ from the single-threaded semantics of a target language, where *states* and *states*$'$ hold the thread-local state for each thread and m and m' are shared memories. We can extend this to a labeled transition system by labeling each transition with the observable portion of the execution state. We expect that thread-local state, being local, is not externally visible, so the observable state will be some subset of the shared memory – it may be the entire shared memory, or we may need to assume that some locations

are local to particular threads. This, then, will be our approach: we will consider pairs $(states, m)$ to be the "states" of a tCFG $\mathcal{G}$, and say that we can make a transition $(states, m) \xrightarrow{\ell} (states', m')$ in $\mathcal{G}$ if $\mathcal{G} \vdash (states, m) \rightarrow (states', m')$ and $\ell = m'|_{obs}$, where obs is the set of observable memory locations. Then if $\mathcal{G}$ simulates $\mathcal{G}'$, any trace of externally observable memory states that can be produced by $\mathcal{G}'$ can also be produced by $\mathcal{G}$, and we can consider the transformation from $\mathcal{G}$ to $\mathcal{G}'$ to be safe. We can derive a similar transition relation on a single-thread CFG G from its memory-aware unlabeled transition system, and talk about simulation between both single-thread CFGs and tCFGs.

While PTRANS is expressive enough to allow optimizations that transform multiple threads simultaneously, many optimizations (especially those that are parallel retoolings of sequential optimizations) transform the code of only one thread. We would like to provide special support for this case, so that verifiers do not have to provide a correct simulation relation on an entire tCFG state in order to prove correctness of a single-thread optimization. Fortunately, simulation is well suited to this kind of modularization.

Definition 5. *Let the execution state of a parallel program with tCFG $\mathcal{G}$ be a pair $(states, m)$, where $states$ is a vector of per-thread execution states and m is a shared memory. The* lifting *of a simulation relation $\preceq$ on single-threaded CFGs to parallel execution states relative to a thread t is defined by $(states, m) \lceil\preceq\rceil_t (states', m') \triangleq (states_t, m) \preceq (states'_t, m') \wedge \forall u \neq t.\ states_u = states'_u$.*

Theorem 4. *Fix a memory model supporting the functions* free_set*,* can_read*, and* update_mem*. Let $\mathcal{G}$ be a tCFG, t be a thread in $\mathcal{G}$, and obs be the set of observable memory locations. Suppose that $\preceq$ is a simulation relation such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, $\mathcal{G}'_u = \mathcal{G}_u$ for all $u \neq t$, and for all $(s', m') \preceq (s, m)$ the following hold:*

1. free_set m = free_set m'
2. *For any $u \neq t$, if $u, \mathcal{G}_u, m' \vdash s_1 \xrightarrow{a} s_2$, then* can_read $m\ u\ \ell$ = can_read $m'\ u\ \ell$ *for every location ℓ mentioned in an operation in a*
3. *For any $u \neq t$, if $u, \mathcal{G}_u, m \vdash s_1 \xrightarrow{a} s_2$ and* update_mem $m'\ a\ m'_2$ *holds, then there exists some m_2 such that* update_mem $m\ a\ m_2$ *holds, $m_2|_{obs} = m'_2|_{obs}$, and $(s', m'_2) \preceq (s, m_2)$*

Then $\lceil\preceq\rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil\preceq\rceil_t \mathcal{G}$.

Proof. In order to prove simulation, we assume we have some execution states $(states, m)$ and $(states', m')$ such that $(states', m') \lceil\preceq\rceil_t (states, m)$ and $\mathcal{G}' \vdash (states', m') \xrightarrow{k} (states'_2, m'_2)$, and show that there exists a state $(states_2, m_2)$ such that $\mathcal{G} \vdash (states, m) \xrightarrow{k} (states_2, m_2)$ and $(states'_2, m'_2) \lceil\preceq\rceil_t (states_2, m_2)$. Deconstructing the step relation on tCFGs, there must be some thread u and single-thread state s'_2 such that $u, \mathcal{G}_u, m' \vdash states'_u \xrightarrow{a'} s'_2$, update_mem $m'\ a'\ m'_2$ holds, $states'_2 = states'(u \mapsto s'_2)$, and $k = m'_2|_{obs}$.

First, consider the case in which $u = t$. Then since $\mathcal{G}'_t \preceq \mathcal{G}_t$, we know that there exists some (s_2, m_2) and memory operations a such that $t, \mathcal{G}_t, m \vdash \mathit{states}_t \xrightarrow{a} s_2$, $\mathsf{update_mem}\ m\ a\ m_2$ holds, $k = m_2|_{obs}$, and $(s'_2, m'_2) \preceq (s_2, m_2)$. This is sufficient for us to conclude that $\mathcal{G} \vdash (\mathit{states}, m) \xrightarrow{k} (\mathit{states}(t \mapsto s_2), m_2)$ and $(\mathit{states}'_2, m'_2) \lceil \preceq \rceil_t (\mathit{states}(t \mapsto s_2), m_2)$, as desired.

Now consider the case in which $u \neq t$. By properties 1 and 2 of $\preceq$, we know that $\mathsf{free_set}\ m = \mathsf{free_set}\ m'$ and that $\mathsf{can_read}\ m\ u\ \ell = \mathsf{can_read}\ m'\ u\ \ell$ for all ℓ mentioned in operations in a'. This encapsulates exactly the information about the memory model used in the step: since free_set and can_read are the interface through which threads can learn about the memory state, and a' includes all read operations performed in the current step, we effectively know that m and m' provide the same data for the purpose of the step from s to s', implying that $u, \mathcal{G}_u, m \vdash \mathit{states}_u \xrightarrow{a'} s'_2$ also holds. Then from the third property of $\preceq$, we have that there exists an m_2 such that $\mathsf{update_mem}\ m\ a'\ m_2$ holds, $m_2|_{obs} = m'_2|_{obs} = k$, and $(\mathit{states}'_t, m'_2) \preceq (\mathit{states}_t, m_2)$. Using the definition of the step relation on tCFGs, this gives us that $\mathcal{G} \vdash (\mathit{states}, m) \xrightarrow{k} (\mathit{states}(u \mapsto s'_2), m_2)$. Furthermore, since $\mathit{states}_u = \mathit{states}'_u$ for all $u \neq t$ and we stepped from states_u to s'_2 in both graphs, we once again have $(\mathit{states}'(u \mapsto s'_2), m'_2) \lceil \preceq \rceil_t (\mathit{states}(u \mapsto s'_2), m_2)$. □

While the exact conditions of the theorem are complicated, the intuition is straightforward: if $\preceq$ is a simulation relation for $\mathcal{G}_t$ and $\mathcal{G}'_t$ such that $(s', m') \preceq (s, m)$ implies that m and m' look the same to all threads $u \neq t$, and $\preceq$ is preserved by steps of threads other than t, then $\lceil \preceq \rceil_t$ is a simulation relation for $\mathcal{G}$ and $\mathcal{G}'$. This theorem allows us to break proofs of correctness for transformations on multithreaded programs into two parts: correctness of the simulation on the transformed thread, and validity of the relation with respect to the remaining threads. In the case in which the simulation relation requires that $m = m'$, i.e., in which the optimization does not change the effects of $\mathcal{G}_t$ on shared memory, most of these conditions are trivial, and we can further simplify the side conditions:

Corollary 1. *Suppose that $\preceq$ is a simulation relation such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, $\mathcal{G}'_u = \mathcal{G}_u$ for all $u \neq t$, and for all $(s', m') \preceq (s, m)$ the following hold:*

1. $m = m'$
2. *If* update_mem *$m\ a\ m_2$ holds and none of the operations in a are performed by thread t, then* $(s', m_2) \preceq (s, m_2)$

Then $\lceil \preceq \rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil \preceq \rceil_t \mathcal{G}$.

In this case, a single-thread simulation relation can be lifted to the entire tCFG under almost any circumstances; we need only ensure that operations performed by other threads cannot change the memory

so that the simulation relation no longer holds. In optimizations that produce noticeable effects on the shared memory, on the other hand, the proof of the premises of Theorem 4 will involve some effort.

6.2 Verifying a MiniLLVM Optimization

In this section, we will attack the problem of proving the correctness of an optimization expressed in the PTRANS framework, with the MiniLLVM of Section 4.3 as our target language. The candidate optimization is Redundant Store Elimination (RSE), which eliminates stores that will always be overwritten before they are used. The basic shape of the program fragment to be rewritten is as in Section 5.3, and is shown in Figure 6.1. (Note that s is replaced by an `is_pointer` instruction, rather than being eliminated entirely, to preserve failures: in the original program, if e_2 is not pointer-valued at s, the program will crash immediately, while eliminating s would allow the program to run until reaching s', thus potentially introducing new behavior.)

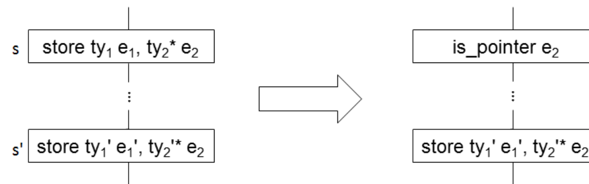


Figure 6.1: Redundant Store Elimination

In sequential code, the optimization is safe if, between the eliminated store s and the following store s', there is no store to or read from the location referred to by e_2 and the value of e_2 is not changed. In the parallel case, the correctness condition is more complex, due to the fact that changes to a memory location can be observed by other threads. While correctness can be ensured by using mutual exclusion analysis to ensure that no other threads use the memory at e_2 before s', under relaxed memory models, more flexible versions of the optimization are possible. This brings up an important point in optimization design: while writing correct optimizations is relatively simple – in the worst case, an optimization with an unsatisfiable side condition is always correct – writing correct optimizations with maximally permissive side conditions can be quite complex. The process of verifying an optimization also provides insight into the reasons for its correctness, and may (as in this example) lead to ideas for more permissive but still correct conditions.

6.2.1 Specifying the Optimization

The "rewrite plus condition" style of PTRANS lends itself well to stating multiple versions of an optimization for different memory models. We begin with this generalization of the statement of RSE from Section 5.3:

$$\begin{aligned} RSE(\varphi) \triangleq\ & \mathsf{replace}\ n\ \mathsf{with}\ \mathtt{is_pointer}\ e_2\ \mathsf{if} \\ & EF\ \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\mathtt{store}\ ty_1\ e_1, ty_2^*\ e_2) \wedge \varphi \end{aligned}$$

Now, for each memory model, we need only provide a condition φ that ensures that the optimization is safe to perform. In general, this will be an "until"-property stating necessary conditions on the nodes between n and the next store to e_2, possibly along with some general properties on the tCFG being transformed. In the following section, we will show this property can vary depending on the memory model under which the optimization is performed by stating it under each of the memory models described in Section 4.5.

RSE under Sequential Consistency

Sequential consistency, the most restrictive of our three memory models, has therefore the most restrictive side condition. Since the removal of the `store` instruction at node n is instantly visible to the other threads in the program, the side condition must ensure that no other thread's behavior depends on the value stored at e_2. One way to ensure this would be to require that no memory operations are performed between n and a following store to e_2; a second approach, which we take here, is to take advantage of mutual exclusion analysis. If other threads may be affected by the value stored at e_2, but cannot do so while t is in the region from n to the following store to e_2, then the optimization can still be performed safely. Using the mutex predicates described in Sections 3.3 and 3.5, the condition can be written as:

$$\begin{aligned} \varphi_{SC} \triangleq\ & \mathsf{protected}(x, e_2) \wedge \neg\mathsf{is}(x, e_2) \wedge A\ \mathsf{in_critical}_t(x, e_2) \wedge (\mathsf{node}_t(n) \vee \mathsf{not_touches}_t(e_2)) \\ & \mathcal{U}\ (\mathsf{in_critical}_t(x, e_2) \wedge \neg\mathsf{node}_t(n) \wedge \exists ty_1'\ e_1'\ ty_2'.\ \mathsf{stmt}_t(\mathtt{store}\ ty_1'\ e_1', ty_2'\ e_2)) \end{aligned}$$

RSE under Total Store Ordering

The relaxation of ordering constraints provided by the TSO memory model has an interesting interaction with the RSE optimization. As described in Section 4.5, TSO allows store instructions to be delayed past other instructions, so certain types of redundant store occur, there are executions of the original program in which the redundant store is delayed until immediately before the store that makes it redundant. In other words, if there are no instructions between n and the following store that prevent its being delayed, then there are in fact executions of the original program in which the store at n was never visible to any other

thread. In this case, there is no need to use mutual exclusion to hide the value at e_2 from other threads; the observable behaviors of the program with the eliminated store will be a strict subset of those of the original program.

Based on this observation, we can provide a side condition for RSE under TSO that, while it does not cover every case covered by φ_{SC}, allows the optimization to be applied in considerably more cases and does not rely on potentially expensive or ineffective mutex analysis. In TSO, a write can be delayed past reads to different locations, but not past writes or atomic read-writes. To construct the necessary side condition, we must first define a predicate not_loads that is analogous to not_touches, but checks only for `load` instructions that might alias, rather than all memory operations:

$$\mathsf{not_loads}_t(e) \triangleq \forall e'\ x\ ty.\ \mathsf{stmt}_t(\%x = \mathtt{load}\ ty^*\ e') \Rightarrow \mathsf{cannot_alias}_t(e', e)$$

We also make use of a predicate not_mods that uses the MiniLLVM-specific atomic predicate def to check that the value of an expression e is not changed by the current instruction (i.e., the local variables that appear in e are not modified):

$$\mathsf{not_mods}_t(e) \triangleq \neg\mathsf{def}_t(e) \land \neg\exists x\ ty\ args.\ \mathsf{stmt}_t(\%x = \mathtt{call}\ ty\ args) \land \neg\exists e'.\ \mathsf{stmt}_t(\mathtt{return}\ e')$$

We can then construct the following condition:

$$\begin{aligned}\varphi_{TSO} \triangleq\ & AX_t(A\ \mathsf{not_mods}_t(e_2) \land \mathsf{not_loads}_t(e_2) \land \neg(\exists x\ ty_1\ e_1\ ty_2\ e_2'\ ty_3\ e_3.\ \mathsf{stmt}_t(\mathtt{store}\ ty_1\ e_1, ty_2^*\ e_2') \lor \\ & \qquad \mathsf{stmt}_t(\%x = \mathtt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e_2', ty_3\ e_3)) \\ & \quad \mathcal{U}\ (\neg\mathsf{node}_t(n)\ \land \exists ty_1'\ e_1'\ ty_2'.\ \mathsf{stmt}_t(\mathtt{store}\ ty_1'\ e_1', ty_2'\ e_2)))\end{aligned}$$

The body of this condition (inside the AX_t operator) provides a characterization of the nodes between n and the following store to e_2 that will be useful in proving the optimization's correctness; we will call it φ'_{TSO}, where $\varphi_{TSO} = AX_t\ \varphi'_{TSO}$. Note that φ_{SC} also still serves as a reasonable side condition under TSO, and we could form a still more general optimization by using $\varphi_{SC} \lor \varphi_{TSO}$ as our side condition.

RSE under Partial Store Ordering

The relaxation of the PSO memory model is simply a more permissive version of that of TSO, and so we can obtain a side condition for it by simply relaxing the constraints of φ_{TSO}. A write in PSO can be delayed past reads and writes to different locations, but not past operations on the same location or atomic read-writes.

As such, a reasonable side condition for it is:

$$\varphi_{PSO} \triangleq AX_t(A\ \mathsf{not_mods}_t(e_2) \wedge \mathsf{not_touches}_t(e_2) \wedge \neg(\exists x\ ty_1\ e_1\ ty_2\ e_2'\ ty_3\ e_3.$$
$$\mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e_2', ty_3\ e_3))$$
$$\mathcal{U}\ (\neg\mathsf{node}_t(n)\ \wedge \exists ty_1'\ e_1'\ ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1' e_1', ty_2'\ e_2)))$$

This condition is strictly weaker than φ_{TSO}, allowing the optimization to be applied to a wider range of programs. As above, we define φ'_{PSO} such that $\varphi_{PSO} = AX_t\ \varphi'_{PSO}$ for use in verification.

6.2.2 Verification

We are now ready to demonstrate the correctness of our several versions of MiniLLVM RSE in PTRANS. As laid out in Section 6.1.1, we will prove the correctness of the reordering transformation by showing that for any transformed tCFG $\mathcal{G}'$ produced by applying the optimization to a graph $\mathcal{G}$, there exists a simulation relation $\preceq$ such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, states related by $\preceq$ make the same values visible to threads other than t, and steps by threads other than t preserve $\preceq$. For each version of RSE, we will present a candidate relation and prove that it is in fact a simulation relation.

Theorem 5. *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{SC}) on a tCFG $\mathcal{G}$, and ℓ be the location targeted by the redundant store removed in $\mathcal{G}'$. Let $\preceq_{SC}$ be the relation such that $(s', m') \preceq_{SC} (s, m)$ iff*

- $s = s'$
- *either $\ell \in \mathsf{free_set}\ m$ and $\ell \in \mathsf{free_set}\ m'$, or $\ell \notin \mathsf{free_set}\ m$ and $\ell \notin \mathsf{free_set}\ m'$*
- *either $m = m'$, or else φ_{SC} holds at the program point of s in $\mathcal{G}$ and $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$.*

Then $\lceil\preceq_{SC}\rceil_t$ is a simulation relation such that $\mathcal{G}'\ \lceil\preceq_{SC}\rceil_t\,\mathcal{G}$ with all locations other than ℓ observable.

Proof. By Theorem 4. First, we must show that $\mathcal{G}'_t \preceq_{SC} \mathcal{G}_t$, which is true iff for every configuration (s', m') of $\mathcal{G}'_t$, if $\mathcal{G}'_t \vdash (s', m') \xrightarrow{k} (s'_2, m'_2)$ and $(s', m') \preceq_{SC} (s, m)$, then there exist s_2, m_2 such that $\mathcal{G}_t \vdash (s, m) \xrightarrow{k} (s_2, m_2)$ and $(s'_2, m'_2) \preceq_{SC} (s_2, m_2)$. By the definition of $\preceq_{SC}$, $s = s'$ and either $m = m'$ or φ_{SC} holds at the program point of s in $\mathcal{G}$ and $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$. If $m = m'$ and the program point of s is not the node where the redundant store was eliminated, then $\mathcal{G}_t$ executes the same instruction as $\mathcal{G}'_t$ in the same configuration (s, m), and so $\mathcal{G}_t \vdash (s, m) \xrightarrow{k} (s'_2, m'_2)$ and $(s'_2, m'_2) \preceq_{SC} (s'_2, m'_2)$. If $m = m'$ and the program point of s is the transformed node, then $\mathcal{G}'_t$ executed the `is_pointer` instruction to go from (s, m) to (s'_2, m'_2), and $\mathcal{G}_t$ executes the `store` instruction at s. By the semantics of the `is_pointer` and `store` instructions we can

infer that $\mathcal{G}_t \vdash (s, m) \xrightarrow{k'} (s'_2, m'_2(\ell \mapsto v))$ for some v, and since $m'_2(\ell \mapsto v)|_{\overline{\ell}} = m'_2|_{\overline{\ell}}$ we know that $k' = k$. Furthermore, since the side condition of the RSE transformation is true on $\mathcal{G}$, we know that φ_{SC} holds at the transformed node, and so $(s'_2, m'_2) \preceq_{SC} (s'_2, m'_2(\ell \mapsto v))$.

If $m \neq m'$, then we know that φ_{SC} holds at the program point of s. Recall that the condition $\mathsf{not_touches}_t(e_2)$ guarantees us that unless we have reached a store to e_2 in $\mathcal{G}$, the instruction at s will not use or modify the value at ℓ. We can break this situation down into three cases: either we are at the store in $\mathcal{G}$ that was removed, we are at some instruction that does not use or affect the value of e_2, or we are at a store to ℓ in both $\mathcal{G}$ and $\mathcal{G}'$. In the first case, we once again have that $\mathcal{G}_t \vdash (s, m) \xrightarrow{k} (s'_2, m'_2(\ell \mapsto v))$ for some v and $(s'_2, m'_2) \preceq_{SC} (s'_2, m'_2(\ell \mapsto v))$. In the second case, we know that $\mathcal{G}_t$ executes the same instruction as $\mathcal{G}'_t$ in effectively equivalent configurations, and the value of ℓ is preserved by the step: then $\mathcal{G}_t \vdash (s, m) \xrightarrow{k} (s'_2, m'_2(\ell \mapsto m(\ell)))$ and $(s'_2, m'_2) \preceq_{SC} (s'_2, m'_2(\ell \mapsto m(\ell)))$. In the third case, by the semantics of the `store` instruction, we have that $\mathcal{G}'_t \vdash (s, m') \xrightarrow{k} (s, m'(\ell \mapsto v))$ and $\mathcal{G}_t \vdash (s, m) \xrightarrow{k'} (s, m(\ell \mapsto v))$ for some value v. But since $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$, we know that $m(\ell \mapsto v) = m'(\ell \mapsto v)$, $k' = k$, and $(s, m(\ell \mapsto v)) \preceq (s, m(\ell \mapsto v))$.

It remains to show that the conditions of Theorem 4 are satisfied for any states such that $(s', m') \preceq_{SC} (s, m)$. We know that $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$ and ℓ is either in the $\mathsf{free_set}$s of both m and m' in in neither, so $\mathsf{free_set}\ m = \mathsf{free_set}\ m'$. Conditions 2 and 3 are trivial when $m = m'$, and when $m \neq m'$, φ_SC guarantees that thread t is in a critical section for ℓ. By correctness of mutex analysis, we know that a thread $u \neq t$ cannot perform any memory operations on ℓ while t is in a critical section for ℓ. Condition 2 follows immediately from this fact, and for condition 3, we need only note that when $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$ and we update m' to m'_2 using operations that do not involve ℓ, we can perform the same operations on m and reach a memory m_2 such that once again $m_2|_{\overline{\ell}} = m'_2|_{\overline{\ell}}$. Since the satisfaction of φ_{SC} depends only on the state of t, φ_{SC} still holds after a step by any other thread u, and we can conclude that $(s', m'_2) \preceq (s, m_2)$. With these conditions proved, we have shown that $\lceil \preceq_{SC} \rceil_t$ is a simulation relation showing the correctness of $RSE(\varphi_{SC})$. □

Recall that, while in SC the memory is simply a map m from locations to values, in TSO and PSO it is a pair (m, b) of a shared memory and a collection of per-thread write buffers. Since the correctness of our conditions under these models depends on our ability to delay stores until they become redundant, we must have a notion of one buffer being a "redundant expansion" of another.

Definition 6. *A* write buffer *is a queue of writes expressed as location-value pairs. A write buffer b' is a* redundant expansion *of b if b' can be constructed from b by adding, in front of each pair (ℓ, v) in b, zero or more writes of other values to ℓ. We will say that a collection of write buffers c' is a redundant expansion of a collection c when each write buffer c'_t is a redundant expansion of the corresponding write buffer c_t.*

Because the added writes appear immediately in front of other writes to the same location, they can be immediately overwritten when the buffers are cleared, and are never read when looking for the latest write to a location. This allows a redundant expansion of b to simulate the behavior of b with regard to the memory-model functions.

Lemma 1. *In TSO and PSO, if b' is a redundant expansion of b, then* $\mathsf{can_read}\ (m, b)\ t = \mathsf{can_read}\ (m, b')\ t$.

Proof. In both TSO and PSO, $\mathsf{can_read}\ (m, b)\ t\ \ell$ begins by searching b_t for the most recently inserted (i.e., backmost) write to ℓ, and if none is found, returns $m(\ell)$ instead. Since any added writes to ℓ in b' appear immediately in front of other writes to ℓ, none of them is the most recently inserted write to ℓ, so $\mathsf{can_read}\ (m, b)\ t\ \ell = \mathsf{can_read}\ (m, b')\ t\ \ell$ for any ℓ. □

Lemma 2. *In TSO and PSO, if b' is a redundant expansion of b and* $\mathsf{update_mem}\ (m, b)\ a\ (m_2, b_2)$ *holds, then there exists a buffer b'_2 such that* $\mathsf{update_mem}\ (m, b')\ a\ (m_2, b'_2)$ *holds and b'_2 is a redundant expansion of b_2.*

Proof. In both TSO and PSO, $\mathsf{update_mem}\ (m, b)\ a\ (m_2, b_2)$ holds iff (m_2, b_2) can be produced from (m, b) by first adding all writes in a to the appropriate write buffers (and executing all non-write operations) to produce an intermediate memory (m_i, b_i), and then removing some number of writes from the fronts of buffers in b_i in any order and updating m_i with the values written. If b' is a redundant expansion of b, then applying a to (m, b') can produce the intermediate memory (m_i, b'_i) where b'_i is a redundant expansion of b_i. Now consider each write moved from b_i to the shared memory. Starting with some buffer with head (ℓ_0, v_0) and tail b_j, we removed (ℓ_0, v_0) from the buffer and updated the shared memory m_j to $m_j(\ell_0 \mapsto v_0)$. The redundant expansion of this buffer has a prefix b_0 consisting of (ℓ_0, v_0) preceded by zero or more other writes to ℓ_0, and a suffix b'_j that is a redundant expansion of b_j. Then we can move all of the writes in b_0 to m_j, once again producing $m_j(\ell_0 \mapsto v_0)$, and leaving the buffer b'_j that is a redundant expansion of b_j. By induction on the number of writes moved, we can use this process to produce a memory state (m'_2, b'_2) such that $\mathsf{update_mem}\ (m, b')\ a\ (m'_2, b'_2)$ holds, $m'_2 = m_2$, and b'_2 is a redundant expansion of b_2. □

When a redundant store instruction is eliminated from a graph $\mathcal{G}$ to produce a graph $\mathcal{G}'$, each execution of $\mathcal{G}$ can be matched with an execution of $\mathcal{G}$ in which the write buffer for the transformed thread t in $\mathcal{G}$ is a redundant expansion of the write buffer for t in $\mathcal{G}'$ (where the redundant writes are those produced by the redundant store), and this fact lies at the heart of the proof of correctness of RSE under these models.

Theorem 6. *Let $\mathcal{G}'$ be a tCFG in the output of $RSE(\varphi_{TSO})$ on a tCFG $\mathcal{G}$. Let $\preceq_{TSO}$ be the relation such that $(s', (m', b')) \preceq_{TSO} (s, (m, b))$ iff*

- $s = s'$, $m = m'$, *and* $b_u = b'_u$ *for all* $u \neq t$, *and*

- *either (1)* b_t *is a redundant expansion of* b'_t, *or else (2)* φ'_{TSO} *holds at the program point of* s *in* $\mathcal{G}$, *the store eliminated in* $\mathcal{G}'$ *was to some expression* e_2, *and there is a location* ℓ *such that* e_2 *evaluates to* ℓ *in* s, *the last write in* b_t *is a write to* ℓ, *and the rest of* b_t *is a redundant expansion of* b'_t.

Then $\lceil \preceq_{TSO} \rceil_t$ *is a simulation relation such that* $\mathcal{G}' \lceil \preceq_{TSO} \rceil_t \mathcal{G}$ *with all locations observable.*

Proof. By Theorem 4. First, we must show that $\mathcal{G}'_t \preceq_{TSO} \mathcal{G}_t$, which is true iff for every configuration $(s', (m', b'))$ of $\mathcal{G}'_t$, if $\mathcal{G}' \vdash (s', (m', b')) \xrightarrow{k} (s'_2, (m'_2, b'_2))$ and $(s', (m', b')) \preceq_{SC} (s, (m, b))$, then there exist s_2, m_2, b_2 such that $\mathcal{G}_t \vdash (s, (m, b)) \xrightarrow{k} (s_2, (m_2, b_2))$ and $(s'_2, (m'_2, b'_2)) \preceq_{TSO} (s_2, (m_2, b_2))$. Note that the definition of $\preceq_{TSO}$ implies that $s = s'$ and $m = m'$. Let us consider the two cases of the simulation relation. If b_t is a redundant expansion of b'_t and the program point of s is not the node where the redundant store was eliminated, then by Lemma 1, $\mathcal{G}_t$ has exactly the same view of the shared memory as $\mathcal{G}'_t$. In conjunction with Lemma 2, this implies that there exists some b_2 such that $\mathcal{G}_t \vdash (s, (m, b)) \xrightarrow{k} (s_2, (m_2, b_2))$, where $b_{2,t}$ is a redundant expansion of $b'_{2,t}$ and $b_{2,u} = b'_{2,u}$ for all $u \neq t$. This is sufficient to conclude that $(s_2, (m_2, b'_2)) \preceq_{TSO} (s_2, (m_2, b_2))$. If b_t is a redundant expansion of b'_t and the program point of s is the transformed node, then $\mathcal{G}'_t$ executed the `is_pointer` instruction to go from $(s, (m, b'))$ to $(s'_2, (m'_2, b'_2))$, and $\mathcal{G}_t$ executes the `store` instruction at s. By the definition of the step relation and the semantics of the `is_pointer` instruction, we have that $t, \mathcal{G}'_t, (m, b') \vdash s \to s'_2$ and $\mathsf{update_mem}\ (m, b')\ \emptyset\ (m'_2, b'_2)$. Then by Lemma 2, there exists a redundant expansion b_2 of b'_2 such that $\mathsf{update_mem}\ (m, b)\ \emptyset\ (m'_2, b_2)$. Since we can always choose to hold a write in a write buffer rather than moving it to shared memory, this implies that $\mathsf{update_mem}\ (m, b)\ \{\mathsf{write}\ t\ \ell\ v\}\ (m'_2, c)$, where $c_t = b_{2,t}; (\ell, v)$ and $c_u = b_{2,u}$ for $u \neq t$. Using this fact and the semantics of the `store` instruction, we can conclude that $\mathcal{G}_t \vdash (s, (m, b)) \xrightarrow{k} (s'_2, (m'_2, c))$. Furthermore, since the side condition of the RSE transformation is true on $\mathcal{G}$, we know that φ'_{TSO} holds at the transformed node, and so $(s'_2, (m'_2, b'_2)) \preceq_{TSO} (s'_2, (m'_2, c))$.

If, on the other hand, we are in case (2), then we know that the eliminated store was to some expression e_2 that evaluates to a location ℓ and that φ'_{TSO} holds at the program point of s. Thus, we are in one of two situations: either we are at a store to e_2 other than the eliminated store, or we are at an instruction that does not modify e_2 or load from ℓ and is not a `store` or `cmpxchg` instruction. In the former case, by the definition of the step relation and the semantics of the `store` instruction, we have that $t, \mathcal{G}'_t, (m, b') \vdash s \xrightarrow{\mathsf{write}\ t\ \ell\ v} s'_2$ for some v, and that $\mathsf{update_mem}\ (m, b')\ \{\mathsf{write}\ t\ \ell\ v\}\ (m'_2, b'_2)$. But because writes are always processed by first storing them in write buffers, we know that $\mathsf{update_mem}\ (m, b')\ \{\mathsf{write}\ t\ \ell\ v\}\ (m'_2, b'_2)$ iff $\mathsf{update_mem}\ (m, c')\ \emptyset\ (m'_2, b'_2)$, where $c'_t = b'_t; (\ell, v)$ and $c'_u = b'_u$ for $u \neq t$. Let $b_t = c_t; (\ell, v')$ for some c_t and

v, as required by the conditions of case (2); then $c_t; (\ell, v'); (\ell, v)$ is a redundant expansion of c'_t. By Lemma 2, this means that there exists a redundant expansion b_2 of b'_2 such that $\mathsf{update_mem}\ (m, d)\ \emptyset\ (m'_2, b_2)$, where $d_t = c_t; (\ell, v'); (\ell, v)$ and $d_u = b_u = b'_u$ for $u \neq t$. Because writes are processed by first storing them in write buffers, this implies that $\mathsf{update_mem}\ (m, b)\ \{\mathsf{write}\ t\ \ell\ v\}\ (m'_2, b_2)$, which in combination with the semantics of $\mathtt{store}$ gives us that $\mathcal{G}_t \vdash (s, (m, b)) \xrightarrow{k} (s'_2, (m'_2, b_2))$ and $(s'_2, (m'_2, b'_2)) \preceq_{TSO} (s'_2, (m'_2, b_2))$. In the case in which we have not yet reached a store to e_2, $\mathcal{G}_t$ and $\mathcal{G}'_t$ execute the same instruction and, by Lemma 1, see the same values in shared memory (since they do not load from ℓ), and do not write any values to memory or change the value of e_2. By the definition of the step relation, $t, \mathcal{G}'_t, (m, b') \vdash s \xrightarrow{a} s'_2$ and $\mathsf{update_mem}\ (m, b')\ a\ (m'_2, b'_2)$, and furthermore the only operations in a are allocs and reads. This means that the only differences between b' and b'_2 are caused by writes moving from the fronts of write buffers to memory, and building on Lemma 2, the same writes can be moved from b, leaving a buffer b_2 such that $b_{2,t}$ is a redundant expansion of b'_2 followed by a write to ℓ and $b_{2,u} = b'_{2,u}$ for $u \neq t$. This allows us to conclude that $\mathcal{G}_t \vdash (s, (m, b)) \xrightarrow{k} (s'_2, (m'_2, b_2))$ and $(s'_2, (m'_2, b'_2)) \preceq_{TSO} (s'_2, (m'_2, b_2))$.

It remains to show that the conditions of Theorem 4 are satisfied for any pair of states such that $(s', (m', b')) \preceq_{TSO} (s, (m, b))$. The $\mathsf{free_set}$ function depends only on m, and the values that can be read by threads other than t are not affected by t's write buffer, so the first two conditions follow immediately. Finally, if we have $\mathsf{update_mem}\ (m, b')\ a\ (m'_2, b'_2)$ where the operations in a do not involve t's write buffer, we know that b_t is either a redundant expansion of b'_t or a redundant expansion of b'_t followed by a store to ℓ; in the first case the fact that $\mathsf{update_mem}\ (m, b)\ a\ (m'_2, b_2)$ for a redundant expansion b_2 of b'_2 follows directly from Lemma 2, and in the second, by an argument analogous to the one above, we can conclude that $\mathsf{update_mem}\ (m, b)\ a\ (m'_2, b_2)$ such that $b_{2,t}$ is a redundant expansion of b'_2 followed by a write to ℓ. In either case, we have $(s, (m'_2, b'_2)) \preceq_{TSO} (s, (m'_2, b_2))$, and we can conclude that $\lceil \preceq_{TSO} \rceil_t$ is a simulation relation showing the correctness of $RSE(\varphi_{TSO})$. □

Theorem 7. *Let $\mathcal{G}'$ be a tCFG in the output of $RSE(\varphi_{PSO})$ on a tCFG $\mathcal{G}$. Let $\preceq_{PSO}$ be the relation such that $(s', (m', b')) \preceq_{PSO} (s, (m, b))$ iff*

- *$s = s'$, $m = m'$, $b_{u,\ell} = b'_{u,\ell}$ for all ℓ and all $u \neq t$, and*
- *either (1) $b_{t,\ell}$ is a redundant expansion of $b'_{t,\ell}$ for all ℓ, or else (2) φ'_{PSO} holds at the program point of s in $\mathcal{G}$, the store eliminated in $\mathcal{G}'$ was to some expression e_2, and there is a location ℓ such that e_2 evaluates to ℓ in s, $b_{t,\ell}$ is a redundant expansion of $b'_{t,\ell}$ followed by a write to ℓ, and $b_{t,\ell'}$ is a redundant expansion of $b'_{t,\ell'}$ for all other locations ℓ'.*

Then $\lceil \preceq_{PSO} \rceil_t$ is a simulation relation such that $\mathcal{G}'\ \lceil \preceq_{PSO} \rceil_t\ \mathcal{G}$ with all locations observable.

Proof. By Theorem 4. The proof is nearly identical to that of the TSO case. Since write buffers are per-location, `store` instructions to locations other than ℓ may be executed between n and the following write to ℓ without changing the relationship between $b(t, \ell)$ and $b'(t, \ell)$, justifying the weaker side condition; otherwise, the proof proceeds entirely analogously. □

In this manner, the VeriF-OPT framework allows us to express and verify optimizations under several memory models, sharing as much information as possible between specifications and proofs of similar transformations. All of these proofs have been carried out in full formal detail in the Isabelle proof assistant.

6.3 Verifying a GraphBIL Optimization

Optimizing stack-machine code is a very different problem from optimizing register-based code. In a register machine, an assignment, for instance, appears as a single instruction, and if the assignment is found to be redundant the instruction can easily be removed as a unit. In a stack machine, on the other hand, the pieces of an assignment may appear scattered throughout a program. Saabas and Uustalu [70] have argued that the correct approach for a stack machine is not to try to find and eliminate entire dead expressions, but instead to replace individual dead operations with `pop` instructions, and then in a second pass eliminate redundant push/pop pairs. In this section, we will present an optimization of this sort on GraphBIL, and demonstrate its verification process.

GraphBIL has three store-type instructions: `stind`, `starg`, and `stfld`. As we saw in Section 4.4, at runtime store operations further break down into stores to the call stack (local storage) and stores to the heap (shared memory). For simplicity, and to draw a contrast with the technique of the previous section, we will choose `starg` as our redundant store to eliminate in this section. The instruction `starg` j stores a value (drawn from the top of the evaluation stack) to the jth argument slot of the currently executing method; that argument slot can later be read by first using `ldarga` j to load the address of the argument slot, and then `ldind` to load the value from that address. Since `starg` removes its argument from the top of the stack, we must replace it with a `pop` instruction to ensure that later instructions see the same stack as they would in the original program.

Since `starg` can only store to the thread's local state, the definition of this RSE optimization will not require different conditions under different memory models, and a correct single-threaded formulation will also be correct under a concurrent model. This means that we can develop our proof of correctness independent of the memory model, and obtain a proof that can be specialized to any memory model under which we might run our GraphBIL programs.

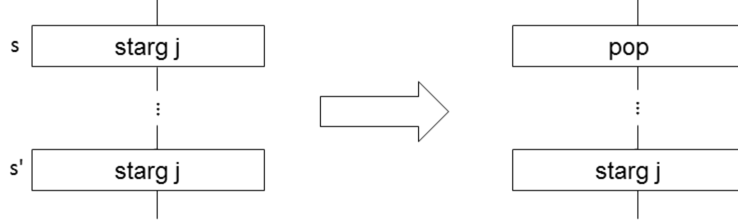


Figure 6.2: Redundant `starg` Elimination

6.3.1 Specifying the Optimization

As before, a `starg` instruction is redundant if the location to which the store occurs is not read or written until the following `starg` to the same location. Once a method call occurs it becomes difficult to track all the references to argument locations, and a `ret` instruction will remove the stored argument from the call stack, so we will rule out those instructions as well. Without using alias analysis, we can write

$$\begin{aligned}\mathsf{safe}_t(j) \triangleq \neg\, (&\mathsf{stmt}_t(\texttt{ldind}) \vee \mathsf{stmt}_t(\texttt{stind}) \vee \mathsf{stmt}_t(\texttt{starg}\ j) \vee \mathsf{stmt}_t(\texttt{ret}) \vee \\ &(\exists A\, c\, \ell\, A_1 \ldots A_k.\ \mathsf{stmt}_t(\texttt{callvirt}\ A\ c{::}\ell(A_1, \ldots, A_k)) \vee \\ &\qquad \mathsf{stmt}_t(\texttt{call instance}\ A\ c{::}\ell(A_1, \ldots, A_k))) \vee \\ &(\exists A\, c\, f.\ \mathsf{stmt}_t(\texttt{stfld}\ A\ c{::}f)))\end{aligned}$$

to characterize the instructions that may safely occur between the original and the following store. We can then use a structure similar to that of our previous RSE optimization:

$$\begin{aligned}RSE_B \triangleq\ &\mathsf{replace}\ n\ \mathsf{with}\ \mathsf{pop}\ \mathsf{if} \\ &EF\mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\texttt{starg}\ j) \wedge AX_t\ (A\ \mathsf{safe}_t(j)\ \mathcal{U}\ \neg\mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\texttt{starg}\ j))\end{aligned}$$

As in the previous section, we will want to refer to the condition that holds between the redundant store and the following store, $\varphi_{BIL} \triangleq A\ \mathsf{safe}_t(j)\ \mathcal{U}\ \neg\mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\texttt{starg}\ j)$, in our proof.

6.3.2 Verification

Theorem 8. *Let $\mathcal{G}'$ be a tCFG in the output of RSE_B on a tCFG $\mathcal{G}$. Let $\preceq_B$ be the relation such that $(s', m') \preceq_B (s, m)$ iff*

- *$m = m'$, and*
- *either (1) $s = s'$, or else (2) $s = (st, vs, args, n)$, $s' = (st, vs, args[j \mapsto v], n)$ for some v, and φ_{BIL} holds at s in $\mathcal{G}$, where `starg` j is the instruction removed in $\mathcal{G}'$.*

Then $\lceil \preceq \rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil \preceq_B \rceil_t \mathcal{G}$ with all locations observable.

Proof. By Theorem 1. First, we must show that $\mathcal{G}'_t \preceq_B \mathcal{G}_t$, which is true iff for every configuration (s', m) of $\mathcal{G}'_t$, if $\mathcal{G}'_t \vdash (s', m) \xrightarrow{k} (s'_2, m_2)$ and $(s', m) \preceq_B (s, m)$, then there exists a state s_2 such that $\mathcal{G}_t \vdash (s, m) \xrightarrow{k} (s_2, m_2)$ and $(s'_2, m_2) \preceq_B (s_2, m_2)$. If $s = s'$ and the program point of s is not the transformed node, then $\mathcal{G}_t$ executes the same instruction as $\mathcal{G}'_t$ in the same configuration (s, m), and so $\mathcal{G}_t \vdash (s, m) \xrightarrow{k} (s'_2, m_2)$ and $(s'_2, m_2) \preceq_B (s'_2, m_2)$. If $s = s'$ and the program point of s is the transformed node, then $\mathcal{G}'_t$ executed the `pop` instruction to go from (s', m) to (s'_2, m_2), while $\mathcal{G}_t$ executes the `starg` j instruction at s. By the semantics of the `pop` and `starg` instructions, we can conclude that $s'_2 = (st, vs, args, n)$ such that $\mathcal{G}_t \vdash s \xrightarrow{k} ((st, vs, args[j \mapsto v], n), m_2)$ for some v, as desired. Furthermore, since the side condition of the transformation is true on $\mathcal{G}$, we know that φ_{BIL} holds at the transformed node, and so $((st, vs, args, n), m_2) \preceq_B ((st, vs, args[j \mapsto v], n), m_2)$.

If $s \neq s'$, then we know that φ_{BIL} holds at the program point of s. Then either we are at an instruction such that $\mathsf{safe}_t(j)$ holds, or we have reached a subsequent `starg` j command. In either case, we are not at the transformed node, so $\mathcal{G}_t$ and $\mathcal{G}'_t$ execute the same instruction. In the former case, $\mathsf{safe}_t(j)$ guarantees that the behavior of the instruction executed is not affected by and does not affect the difference in arguments, and so $\mathcal{G}_t$ can step to a state s_2 that again differs from s'_2 only at argument j. In the latter case, by the semantics of the `starg` instruction, we have that $\mathcal{G}'_t \vdash ((st, vs; v', args[j \mapsto v], n), m) \xrightarrow{k} (((st, vs, args[j \mapsto v'], n'), m))$ and $\mathcal{G}_t \vdash ((st, vs; v', args, n), m) \xrightarrow{k} (((st, vs, args[j \mapsto v'], n'), m))$, and clearly $((st, vs, args[j \mapsto v'], n'), m) \preceq_B ((st, vs, args[j \mapsto v'], n'), m)$.

The conditions of Theorem 1 on the simulation relation follow directly from the definition of $\preceq_B$, and we can conclude that $\lceil \preceq_B \rceil_t$ is a simulation relation showing the correctness of RSE_B. □

6.4 Factoring Out Common Elements

One of the goals of VeriF-OPT is to simplify the process of verifying optimizations by making proofs as modular as possible. In particular, we aim to leverage the language-independence of the framework to state and prove general compiler verification principles, which can then be instantiated for any target language. Our Isabelle development is implemented as a hierarchy of *locales*, in which theorems are parameterized by assumptions and specification elements. For instance, in Section 4.2.1, we gave rules for deriving memory-model-aware single-thread and concurrent step relations from a memory-model-agnostic single-thread step relation for a language; this is implemented as a `step_rel` locale that assumes the existence of a step relation, and then defines and proves properties of the derived relations.

The proof of correctness of each of the RSE optimizations in MiniLLVM relied on about 345 lemmas (excluding facts about infinite lists, general bisimulation, and other library functions/proofs). Of these:

- 55 were basic facts about the semantics of PTRANS.
- 90 were about graphs, CFGs, tCFGs, and paths through tCFGs.
- 30 were about step relations and simulations between graphs given step relations, including the theorems of Section 6.1.1 that let us lift single-thread simulations to concurrent programs.

These lemmas were reused for the GraphBIL verification, and could be reused again in the verification of any optimization on any language under any memory model.

- 25 were about specific memory models, but were independent of the target language.
- 120 were about MiniLLVM, but were independent of the memory model.
- 10 were about MiniLLVM under particular memory models, but could be reused for any MiniLLVM optimization.
- 5 were specific to RSE in MiniLLVM, but were independent of the memory model.
- 10 were specific to the particular version of RSE under the particular memory model being verified.

In all, about half of the work was completely generalizable to other systems; of what remained, the majority was specific to the language being analyzed, but could be reused for any optimization in that language under any memory model. This provides some evidence of the advantages of the language-independent approach, and we expect that as more target languages and optimizations are incorporated into the system, the library of reusable facts will continue to grow.

Chapter 7

Conclusions

We have now outlined and demonstrated the various components of a framework for designing, testing, and verifying compiler optimizations and transformations on parallel programs. The core of the framework is the PTRANS language described in Chapter 3, a domain-specific specification language in which optimizations are expressed as rewrites on control flow graphs with temporal logic (specifically, First-Order CTL) side conditions. We have seen both the abstract mathematical semantics of PTRANS (Section 3.4), which in combination with formal CFG-based semantics for a target language can be used to prove the correctness of optimizations on that language, and the executable semantics (Section 5.2), which forms the basis of an interpreter that turns PTRANS specifications into prototype optimizations. To overcome potential limitations of temporal logic for expressing optimization conditions, we demonstrated in Section 3.5 a method for incorporating external analyses into the CTL side conditions of PTRANS, and stated and proved the correctness of a CTL implementation of mutual exclusion analysis.

While most modern compilers use CFG-based intermediate representations at some point in the compilation process, formal semantics are not usually given at the CFG level. In Chapter 4 we stated a set of guidelines for transforming more conventional small-step or big-step operational semantics rules into small-step CFG-based rules for use with PTRANS, and used them to develop semantics for two intermediate representations, a register-machine language based on the LLVM IR (Section 4.3) and a stack-machine language based on Microsoft's Common Intermediate Language (Section 4.4). Our semantics are memory-model-agnostic by design, and can be specialized to any memory model that can be represented operationally; in Section 4.5 we demonstrated instantiations with sequential consistency, total store ordering, and partial store ordering models.

One of the greatest criticisms levelled at any formal verification work is the high barrier to entry: compiler designers will not necessarily want to learn to prove theorems in a proof assistant in order to write their compilers. The executable semantics of PTRANS, described in Chapter 5, allows us to separate the task of designing and testing optimizations from the heavy machinery of verification. PTRANS specifications can be interpreted directly as optimization prototypes, taking advantage of the Z3 SMT solver to efficiently

find locations that can be optimized. Iterative testing helps designers refine optimizations, and in particular their possibly complex temporal logic side conditions, before handing them over for the time-consuming task of verification. Using the executable semantics, relative novices were able to quickly learn to use PTRANS and state and test complex optimizations.

The original rationale for specifying optimizations in terms of temporal logic side conditions was the relative ease of verification. In Chapter 6 we outlined a method for proving the correctness of PTRANS optimizations that transform a single thread in a multithreaded program, and used it to obtain strong guarantees of correctness for redundant store elimination optimizations on our two target languages under several memory models, showing that the behaviors of every program output by each version of the transformation are a subset of the behaviors of the input program. We observed that the temporal logic side conditions of correct optimizations led directly to correct simulation relations between input and output programs, implying that the side conditions express both the conditions under which the optimization applies and the rationale for its correctness. Overall, we believe that the PTRANS methodology considerably simplifies the process of stating and verifying optimizations on parallel code, by allowing for clean, proof-amenable statements of program transformations, incorporating testing and refinement of specifications as a core part of the verification workflow, and emphasizing modular, reusable formalizations and proofs.

7.1 Discussion

Several other formalisms were tried and shelved or discarded in the process of constructing the framework presented. As one example, we initially believed that the best tool for writing side conditions for transformations on parallel programs would be a parallelism-aware logic such as alternating-time temporal logic (ATL) [3], which expresses formulae over concurrent systems with multiple agents, called concurrent game structures (CGSs). In each step of a CGS, each agent chooses from a range of available actions in the current state, and the transition function takes these choices into account when determining the next state. There is a clear correspondence between CGSs and tCFGs; threads can be thought of as agents, and in each step a thread has a choice between remaining in its current state and executing its next instruction. In previous work, we developed a rely-guarantee-based extension of ATL, called RGTL [51], which seemed particularly well suited to expressing the sorts of properties we would use in PTRANS: for instance, that a thread would be safe to transform as long as other threads guaranteed that they would respect critical sections for some resource.

The agent-based approach turned out to be less useful than we had hoped, for two main reasons. First,

the properties that we wanted to check for the thread to be transformed were often thread-local properties that did not depend on any guarantees provided by other threads. While the correctness of (say) a redundant store elimination may rely on the fact that other threads respect some property, the actual temporal property we require of the thread to be transformed is simply that it has a store followed by another store to the same location (plus some thread-local requirement on the intervening instructions). More fundamentally, agent-based temporal logics rely on an idea of agency; their primary use is in describing properties that multiple agents can cooperate to ensure by coordinating their choices of actions. In the case of multithreaded programs, on the other hand, properties that depend on the "choices" made by other threads (essentially, either to stall or to advance) are rarely useful, since these actions are not actually under the control of the threads. While the rely-guarantee approach may be helpful in proving that conditions across multiple threads are satisfied – for instance, we can imagine a mutual exclusion protocol that depends on an interlocking set of guarantees that one thread will use one resource correctly as long as another uses another resource correctly – we have not yet found any cases in which we benefit from adding explicit rely-guarantee structure to our side conditions. On a related note, Theorem 4 of Section 6.1.1, which allows us to lift a simulation relation from a single thread to a multithreaded program by proving some non-interference conditions on the other threads, seems to encapsulate the beginnings of a rely-guarantee method of verifying our optimizations, and a framework such as RGSim [45] might push this further by, for instance, allowing us to verify a transformation on two threads simultaneously by combining single-thread simulation relations for each thread. We have not yet tested our approach on optimizations that transform multiple threads simultaneously or spawn new threads at runtime, and in doing so may find reasons to extend both our specification language and our verification methods.

Another of our goals was to integrate the K Framework for programming language specification [69] into our platform. Semantics written in K are immediately executable, and rewriting is one of the framework's fundamental principles, so we hoped to use it to give executable semantics to PTRANS in a more formal style. K has been used to give semantics for a variety of languages which might serve as targets for PTRANS, including a subset of C [18]; while these languages are specified in the usual AST style rather than in terms of control flow graphs, it may be possible to give a general automatic translation from one format to the other for suitable classes of intermediate languages. K also includes tools for searching the execution space of a program, which would be useful in developing and testing PTRANS optimizations. Ultimately, we used F# instead of K for our executable semantics because our executable algorithm relied on SMT solving and benefited from F#'s tight integration with the Z3 SMT solver. K does provide some support for SMT solver integration, but crucially, F# allows us to construct Z3 algebraic datatypes, so that we can define

a language-independent translation from instruction syntax to Z3 expressions. The tight interface between F# and Z3 proved valuable in constructing executable semantics for PTRANS, although we would still like to either construct K semantics as well or re-develop the search tools available in K to make it easier to design and test PTRANS optimizations.

7.1.1 Comparison with Related Work

Our work builds on the work of Kalvala et al. on TRANS [28] in several dimensions. We have extended the program model and atomic predicates to handle parallel programs; we have made the target language a parameter and instantiated it with two intermediate languages that are considerably more complex than the original target language of TRANS; we have given our language formal semantics in a proof assistant and used it to verify an optimization under several models; we have given executable semantics that take advantage of SMT solving to quickly find satisfying substitutions for side conditions. In comparison to Cobalt [37], the most prominent existing system that uses the temporal logic approach, we can express a much wider range of transformations and side conditions, and our interactive verification approach means that any correct optimization that can be stated can be verified. On the other hand, Cobalt's limited expressiveness buys it mostly-automatic verification; if the user can provide a suitable witness property, the rest of the proof proceeds completely automatically, whereas interactive proofs require specialist knowledge and are generally quite labor-intensive, even with a library of supporting lemmas as provided by our framework.

Vellvm [80] is a comparable system for optimization verification, containing a formalized subset of the LLVM intermediate language and an established method of verifying optimizations supported by pre-proven lemmas. The semantics of Vellvm's verified LLVM is closer to the actual semantics of LLVM than is MiniLLVM, and includes details of memory layout that are glossed over in our presentation. The verification technique is based on extending the static single assignment property (which holds of all LLVM code, though not of MiniLLVM) to prove other invariants, and has been used to verify a version of the `mem2reg` optimization that lies at the heart of SSA in LLVM (a more complex transformation than our redundant store elimination). The code for its verified optimization can also be extracted from the theorem prover Coq, and runs in time comparable to that of real compilers, while our executable semantics is at present only pencil-and-paper verified and may not scale to real-world programs. The main advantage of our work relative to Vellvm is that it is not tied to a specific target language, and as shown by our experiments with GraphBIL, much of the work done for one target language can be repurposed for another; Vellvm has also not yet addressed the problem of concurrency.

The most comprehensive compiler correctness effort to date is CompCertTSO [71], the extension of

CompCert [41] to the TSO memory model. CompCertTSO includes a range of verified optimizations on intermediate languages at various levels, including one, RTL, which is roughly analogous to our MiniLLVM. The project also includes verified translations between languages, and ultimately provides an end-to-end proof of correctness for a compiler with realistic source and target languages, while we have thus far only verified same-language transformations. We believe that the most significant advantage of our approach is its language- and memory-model independence; the development of tools such as PTRANS will eventually make it easier to construct the next CompCert, and the one after that.

7.2 Ongoing and Future Work

The framework as described leaves plenty of room for future work. We have so far verified only one simple optimization in PTRANS (albeit in several variants); the literature is full of further possibilities. Following in the footsteps of the Vellvm project [80], we have begun the process of specifying and verifying a version of the `mem2reg` optimization, the core transformation used by LLVM to convert programs into static single assignment form. In previous work, we verified a translation into SSA form written in the original TRANS [50]; it would be interesting to see how much of that work could be carried over to a more realistic target language and transformation. Specifying the `mem2reg` optimization may also require recovering information about basic blocks, which is obscured in our control flow graph model. Fortunately, we can fairly easily express conditions involving dominance and basic block structure in CTL, and use this to replicate the analyses performed in the LLVM system.

The redundant store elimination optimization that we have verified in GraphBIL does not affect references to shared memory, and so can be verified independently of the memory model. However, if we wish to verify optimizations that use the shared memory, we must adjust our memory models to take into account the complexities of memory references in an object-oriented system. In CIL, we can load from and store to fields of an object in the shared memory; even if we make the simplifying assumption that each object is held in a single memory location, our memory model must allow actions to affect the sublocations specified by a field access as well as top-level locations. We can generalize this by adding to our concept of locations a separate concept of *pointers*: reads and writes are performed on pointers, while alloc and free operations are performed on entire locations. Our memory models as formalized inhabit the sub-case in which pointers and locations are identical. We have begun to extend our memory model axiomatizations to handle this generalization, and intend to use it to improve the range of GraphBIL and other optimizations that we can verify. This or similar generalizations may also be of use in handling arrays, structs, and other complex

memory layouts, which are so far ignored by our framework.

Since we have made it one of our primary goals to be able to state and verify optimizations involving concurrency, another clear path forward is to test our framework on a wider range of concurrency paradigms. For instance, in the parallelism model used by C and Java, a program begins with a single thread and may fork off more threads (which execute some fragment of the original thread's code) during its execution. The dynamic semantics of this model can be expressed straightforwardly on tCFGs – nothing prevents a program's semantics from adding new CFGs to its tCFG at runtime – but this would introduce new complexities to our static analyses, since we have so far assumed that all threads are present and executing from the beginning of the program. In order to state correct optimizations in PTRANS on programs with fork and join operations, we would need to write conditions that identified (perhaps approximately) which instructions could be guaranteed not to execute in more than one thread, and otherwise construct side conditions that, when checked on a single thread, ensured desirable runtime properties on all threads spawned by that thread – for instance, determining from the code of a single thread that all threads spawned will preserve mutual exclusion for a resource. In general, when working with a variable number of threads, we may be forced to overapproximate the potential for concurrency to ensure correctness, so that our optimizations do not apply in some cases in which they are safe. There are no theoretical limitations barring us from expressing such properties in our framework, and trying to verify optimizations on, for instance, MiniLLVM with fork and join instructions would be an interesting case study and bring our target languages a step closer to the real world.

7.2.1 The Big Picture

The PTRANS language, and the semantics and infrastructure we have built around it, form just one piece of the intended scope of the VeriF-OPT project. The ultimate goal of VeriF-OPT is to provide a unified platform and tool suite for designing, testing, and verifying compiler optimizations. Building on the theoretical underpinnings described in this paper, we intend to develop tools for designing optimizations in PTRANS. CFGs and temporal formulae over them lend themselves well to a visual approach, and tools such as Graphviz [20] are already in common use for displaying program CFGs. We envision a PTRANS design tool that displays PTRANS specifications consistently with their semantics: as graph transformations, rather than written code. By illustrating the transformations and result graphs produced by specifications on sample program fragments (taking advantage of the executable semantics we have already developed), we hope to make PTRANS more concrete and accessible to compiler designers and further reduce the amount of formal-methods knowledge needed to use the system. Once a PTRANS optimization has been developed,

it can be handed off as-is to a verification specialist, who can use the Isabelle semantics and library of shared lemmas to prove its correctness – or, more realistically, turn up details and edge-case behaviors that must be modified by the designer before the optimization can be proved correct, leading to an iterative process of refinement and verification.

In order to use PTRANS as a unified platform for testing and verification, we must also relate our executable semantics to our formal semantics, so that we can be assured that the optimizations we run are semantically equivalent to the optimizations we verify. In the current PTRANS development, we consider the verification semantics in Isabelle to be the reference semantics, since it is the basis for our proofs of correctness, and provide a pencil-and-paper proof that the executable semantics respects the verification semantics. We could provide a stronger guarantee of the correctness of our prototype optimizations by mechanizing this proof in Isabelle. Once we have a proof of correctness for the executable algorithm, it remains to show that the actual implementation of the executable semantics is faithful to the algorithm. One approach would be to prove the correctness of the implementation directly, which would require formal semantics for implementation language; another approach would be to extract code for the executable semantics directly from verifiable Isabelle definitions, which can be done with Isabelle's code generation facility, although some effort is required to formulate the definitions so that code can be extracted, and useful features of F# such as reflection and SMT solver integration cannot easily be emulated by extracted code.

Given that the executable semantics of PTRANS allows us to use PTRANS specifications as optimization prototypes, a reasonable question to ask is how we can use optimizations specified in PTRANS in actual production compilers. The simplest approach would be to integrate the execution algorithm directly into a compiler architecture. For real-world programs, we would need to make considerable improvements to the algorithm so that compilation could be accomplished in a reasonable timeframe. Utimately, though, our specification and execution methods are designed for simplicity, human-readability, and ease of verification, and are unlikely to be able to match the running time of performance-optimized C code. This leaves us with several options. We could try to find a method of compiling PTRANS specifications into a lower-level form closer to the code used in real-world compilers, although we would need to verify this translation in order to preserve any correctness guarantees. We could attempt to prove the faithfulness of existing or new compiler implementations to PTRANS specifications, for instance by finding a method of relating the semantics of traditional dataflow analyses to our CTL side conditions. We could also take a translation-validation approach, checking in each optimization step of some compiler that the transformation performed was an instance of some proved-correct PTRANS rewrite; this approach would require extra checks at compile time,

but would impose a considerably smaller proof burden than would attempting to verify realistic C code with respect to abstract PTRANS specifications.

Finally, optimization is only one phase of the full compiler stack. In general, compilation phases can be divided into two types: those that modify code in a particular representation, and those that translate code from one representation to another. We have shown PTRANS to be a useful tool for designing and verifying transformations on a single language (as long as that language can be expressed in terms of control flow graphs), but we have yet to address the issue of language-to-language translation. Outside of the TRANS approach, the problem has been well studied: for instance, CompCertTSO [71] verifies every translation in a stack of about 13 languages, from a subset of C all the way down to x86 machine code. We may be able to write translations as rewrites on control flow graphs, where we replace and insert instructions in a language other than the language of the existing nodes in the graph – although this would require semantics for mixed-language CFGs. Alternatively, PTRANS transformations may serve as steps in a larger compilation chain, by coupling them with the techniques established by CompCert and other projects. At present, VeriF-OPT is a framework supporting the construction of correct same-language transformations; there is much work to be done to turn it into a framework for constructing correct compilers.

7.3 Summary

In this thesis we have seen the use of PTRANS, a specification language for transformations on parallel programs, as a tool for specifying, testing, and verifying compiler optimizations. PTRANS allows clean and verification-amenable specification of optimizations and transformations, by expressing them as rewrites on control flow graphs with temporal logic side conditions. Its executable semantics can be used to test and validate optimizations before verifying, and its formal semantics in Isabelle can be used to derive strong guarantees of correctness. By generalizing over target languages and over memory models, we can prove a wide range of facts once and for all and then use them as lemmas to reduce the effort involved in verifying any particular optimization. Overall, PTRANS presents an accessible and powerful approach to designing correct transformations for parallel programs, and should allow compiler designers and verification experts to cooperate to more easily develop optimizations with strong guarantees of correctness.

Appendix A

Code Listing

A.1 Isabelle Proofs

The following proofs were developed in Isabelle 2013-2 on Windows 8, using the theories `JinjaThreads` [47] and `List – Infinite` [75] from the Archive of Formal Proofs, version 2013-2.

```
(* trans_syntax.thy *)
(* William Mansky *)
(* The syntax of a transformation specification language. *)

theory trans_syntax
imports "~~/src/AFP/List-Infinite/ListInfinite"
begin

(* General utility type for defining syntax with variables. *)
datatype ('data, 'mvar) literal = Inj 'data | MVar 'mvar

(* Basic syntax bits: node and edge literals. *)
(* Note that these arguments are thread patterns (mvars), not concrete threads. *)
datatype 'mvar node_const = NStart 'mvar | NExit 'mvar

type_synonym 'mvar node_lit = "('mvar node_const, 'mvar) literal"

(* Expression pattern e<e'> matches "an expression e with e' somewhere in it",
   allowing basic pattern-matching within a transformation. *)
datatype ('mvar, 'expr) expr_pattern = EPInj 'expr | EPSubst 'mvar 'expr ("_<_>" 102)

(* Collecting free variables. *)
primrec lit_fv where
"lit_fv fv (Inj data) = fv data" |
"lit_fv _ (MVar mv) = {mv}"
```

```
fun node_fv where
"node_fv (MVar v) = {v}" |
"node_fv (Inj (NStart t)) = {t}" |
"node_fv (Inj (NExit t)) = {t}"

abbreviation "type_fv t \<equiv> lit_fv (\<lambda>x. {}) t"
lemma node_fv_finite [simp]: "finite (node_fv n)"
apply (case_tac n, auto)
apply (case_tac data, auto)
done

lemma type_fv_finite [simp]: "finite (type_fv t)"
by (case_tac t, auto)

primrec expr_pattern_fv where
"expr_pattern_fv expr_fv (EPInj e) = expr_fv e" |
"expr_pattern_fv expr_fv (x<e>) = insert x (expr_fv e)"


(* Actions are the atomic rewrites. *)
datatype ('mvar, 'edge_type, 'pattern) action = AReplace "'mvar node_lit" "'pattern list"
 | ARemoveEdge "'mvar node_lit" "'mvar node_lit" 'edge_type
 | AAddEdge "'mvar node_lit" "'mvar node_lit" 'edge_type
 | ASplitEdge "'mvar node_lit" "'mvar node_lit" 'edge_type 'pattern

(* Side conditions are CTL formulae on program graphs. *)
datatype ('mvar, 'pred) side_cond = SCTrue | SCPred 'pred
 | SCAnd "('mvar, 'pred) side_cond" "('mvar, 'pred) side_cond" (infixl "\<and>sc" 120)
 | SCNot "('mvar, 'pred) side_cond" ("\<not>sc")
 | SCAU "('mvar, 'pred) side_cond" "('mvar, 'pred) side_cond" ("A _ \<U> _" 125)
 | SCEU "('mvar, 'pred) side_cond" "('mvar, 'pred) side_cond" ("E _ \<U> _" 125)
 | SCAB "('mvar, 'pred) side_cond" "('mvar, 'pred) side_cond" ("A _ \<B> _" 125)
 | SCEB "('mvar, 'pred) side_cond" "('mvar, 'pred) side_cond" ("E _ \<B> _" 125)
 | SCEx 'mvar "('mvar, 'pred) side_cond"

primrec cond_fv where
"cond_fv _ SCTrue = {}" |
"cond_fv pred_fv (SCPred p) = pred_fv p" |
"cond_fv pred_fv (\<phi>1 \<and>sc \<phi>2) = cond_fv pred_fv \<phi>1 \<union> cond_fv
    pred_fv \<phi>2" |
```

```
"cond_fv pred_fv (\<not>sc \<phi>) = cond_fv pred_fv \<phi>" |
"cond_fv pred_fv (A \<phi>1 \<U> \<phi>2) = cond_fv pred_fv \<phi>1 \<union> cond_fv
    pred_fv \<phi>2" |
"cond_fv pred_fv (E \<phi>1 \<U> \<phi>2) = cond_fv pred_fv \<phi>1 \<union> cond_fv
    pred_fv \<phi>2" |
"cond_fv pred_fv (A \<phi>1 \<B> \<phi>2) = cond_fv pred_fv \<phi>1 \<union> cond_fv
    pred_fv \<phi>2" |
"cond_fv pred_fv (E \<phi>1 \<B> \<phi>2) = cond_fv pred_fv \<phi>1 \<union> cond_fv
    pred_fv \<phi>2" |
"cond_fv pred_fv (SCEx x \<phi>) = cond_fv pred_fv \<phi> - {x}"

lemma cond_fv_finite [simp, intro]: "\<forall>p. finite (pred_fv p) \<Longrightarrow>
    finite (cond_fv pred_fv \<phi>)"
by (induct \<phi>, auto)

abbreviation SCFalse where "SCFalse \<equiv> \<not>sc SCTrue"
abbreviation SCOr (infixl "\<or>sc" 110) where "\<phi>1 \<or>sc \<phi>2 \<equiv> \<not>sc
    (\<not>sc \<phi>1 \<and>sc \<not>sc \<phi>2)"
definition SCImp (infixl "\<longrightarrow>sc" 105) where "\<phi>1 \<longrightarrow>sc \<
    phi>2 \<equiv> \<not>sc (\<phi>1 \<and>sc \<not>sc \<phi>2)"
declare SCImp_def [simp]
abbreviation SCEF ("EF") where "EF \<phi> \<equiv> E SCTrue \<U> \<phi>"
abbreviation SCAF ("AF") where "AF \<phi> \<equiv> A SCTrue \<U> \<phi>"
abbreviation SCEG ("EG") where "EG \<phi> \<equiv> \<not>sc (AF (\<not>sc \<phi>))"
abbreviation SCAG ("AG") where "AG \<phi> \<equiv> \<not>sc (EF (\<not>sc \<phi>))"
abbreviation SCAll where "SCAll x \<phi> \<equiv> \<not>sc (SCEx x (\<not>sc \<phi>))"
abbreviation SCAW ("A _ \<W> _" 125) where "A \<phi> \<W> \<psi> \<equiv> \<not>sc (E \<not
    >sc \<psi> \<U> (\<not>sc \<phi> \<and>sc \<not>sc \<psi>))"
abbreviation SCEW ("E _ \<W> _" 125) where "E \<phi> \<W> \<psi> \<equiv> (E \<phi> \<U> \<
    psi>) \<or>sc EG \<phi>"
primrec SCAnds where
"SCAnds [] = SCTrue" |
"SCAnds (p#ps) = p \<and>sc SCAnds ps"
primrec SCOrs where
"SCOrs [] = SCFalse" |
"SCOrs (p#ps) = p \<or>sc SCOrs ps"

(* Transformations are the top-level specs. *)
(* State conditions are paired with metavars indicating the node at which they're evaluated
    . *)
```

```
datatype ('mvar, 'edge_type, 'pattern, 'pred) transform =
  TIf "('mvar, 'edge_type, 'pattern) action list" "('mvar, 'pred) side_cond"
| TMatch "('mvar, 'pred) side_cond" "('mvar, 'edge_type, 'pattern, 'pred) transform"
| TApplyAll "('mvar, 'edge_type, 'pattern, 'pred) transform"
| TChoice "('mvar, 'edge_type, 'pattern, 'pred) transform" "('mvar, 'edge_type, 'pattern, '
    pred) transform"
| TThen "('mvar, 'edge_type, 'pattern, 'pred) transform" "('mvar, 'edge_type, 'pattern, '
    pred) transform"

end

(* trans_flowgraph.thy *)
(* William Mansky *)
(* Threaded CFGs for transformation. *)

theory trans_flowgraph
imports trans_syntax
begin

type_synonym ('node, 'edge_type) edge = "'node \<times> 'node \<times> 'edge_type"

definition in_edges where "in_edges e m \<equiv> {(u,t)| u t. (u,m,t) \<in> e}"
definition out_edges where "out_edges e m \<equiv> {(u,t)| u t. (m,u,t) \<in> e}"
definition out_by_t where "out_by_t e t m \<equiv> {u. (m,u,t) \<in> e}"

lemma out_edges_by_t: "out_by_t e t m = {u. (u,t) \<in> out_edges e m}"
by (simp add: out_by_t_def out_edges_def)

lemma out_edges_Un1: "(\<And>u t. (m, u, t) \<notin> e') \<Longrightarrow> out_edges (e \<
    union> e') m = out_edges e m"
by (simp add: out_edges_def)

lemma out_edges_Un2: "(\<And>u t. (m, u, t) \<notin> e) \<Longrightarrow> out_edges (e \<
    union> e') m = out_edges e' m"
by (simp add: out_edges_def)

lemma out_edges_Un [simp]: "out_edges (e \<union> e') m = out_edges e m \<union> out_edges
    e' m"
by (force simp add: out_edges_def)

lemma out_edges_insert [simp]: "out_edges (insert (a, b, c) e) m =
```

```
  (if m = a then insert (b, c) (out_edges e m) else out_edges e m)"
by (force simp add: out_edges_def)

(* Unlabeled graphs with start and exit nodes. *)
(* Call them "pointed graphs". *)
locale pointed_graph =
 fixes Nodes::"'node set"
   and Edges::"('node, 'edge_type) edge set"
   and Start::'node
   and Exit::'node
 assumes finite_nodes [simp]: "finite Nodes"
     and finite_edge_types [simp]: "finite (UNIV::'edge_type set)"
     and edges_ok [simp]: "(u,v,t) \<in> Edges \<Longrightarrow> u \<in> Nodes \<and> v \<
         in> Nodes"
     and has_start [simp]: "Start \<in> Nodes"
     and has_exit [simp]: "Exit \<in> Nodes"
     and start_not_exit [simp]: "Start \<noteq> Exit"
     and start_first [simp]: "in_edges Edges Start = {}"
     and exit_last [simp]: "out_edges Edges Exit = {}"
begin

definition CPaths where "CPaths n \<equiv>
  {path. (\<exists>r. path = n#r) \<and> (\<forall>i<(length path - 1). \<exists>t. (path!i
      , path!(i+1), t) \<in> Edges) \<and>
         \<not>(\<exists>n\<in>Nodes. \<exists>t. (path!(length path - 1), n, t) \<in>
             Edges)}"

(* infinite paths *)
definition Paths where "Paths n \<equiv>
  {path. path 0 = n \<and> (\<forall>i. \<exists>t. (path i, path (i+1), t) \<in> Edges)}"

lemma finite_edges [simp]: "finite Edges"
by (rule_tac B="Nodes \<times> Nodes \<times> UNIV" in finite_subset, auto)

lemma finite_out_edges [simp]: "finite (out_edges Edges n)"
apply (simp add: out_edges_def)
apply (rule_tac A="{n}" in finite_cartesian_productD2, auto)
apply (rule_tac B=Edges in finite_subset, auto)
done
```

```
end

definition "next_node Edges t p \<equiv> SOME p'. (p, p', t) \<in> Edges"
lemma next_in [intro]: "(p, p', t) \<in> Edges \<Longrightarrow> (p, next_node Edges t p, t
    ) \<in> Edges"
apply (simp add: next_node_def)
apply (erule someI)
done

lemma next_only [simp]: "out_edges Edges p = {(p', t)} \<Longrightarrow> next_node Edges t
    p = p'"
by (auto simp add: out_edges_def next_node_def)

lemma next_only_gen: "\<lbrakk>(p, p', t) \<in> Edges; \<forall>(m, s) \<in> out_edges
    Edges p. s = t \<longrightarrow> m = p'\<rbrakk> \<Longrightarrow>
 next_node Edges t p = p'"
by (smt mem_Collect_eq next_in next_node_def out_edges_def splitD)

lemma next_only2 [simp]: "\<lbrakk>out_edges Edges p = {(p', t), (m, s)}; s \<noteq> t\<
    rbrakk> \<Longrightarrow> next_node Edges t p = p'"
by (rule next_only_gen, auto simp add: out_edges_def)

lemma next_only2_2 [simp]: "\<lbrakk>out_edges Edges p = {(m, s), (p', t)}; s \<noteq> t\<
    rbrakk> \<Longrightarrow> next_node Edges t p = p'"
by (rule next_only_gen, auto simp add: out_edges_def)

lemma next_depends: "out_edges Edges p = out_edges Edges' p \<Longrightarrow>
  next_node Edges t p = next_node Edges' t p"
apply (clarsimp simp add: next_node_def out_edges_def)
by (metis (lifting) mem_Collect_eq prod_caseI splitD)

record ('node, 'edge_type) doubly_pointed_graph =
 Nodes::"'node set"
 Edges::"('node, 'edge_type) edge set"
 Start::"'node"
 Exit::"'node"

definition is_doubly_pointed_graph
 :: "('node,'edge_type,'a) doubly_pointed_graph_scheme \<Rightarrow> bool" where
"is_doubly_pointed_graph g \<equiv> pointed_graph (Nodes g) (Edges g) (Start g) (Exit g)"
```

```
declare is_doubly_pointed_graph_def [simp]

(* Count the edge types in a (finite) set of edges. *)
definition edge_types where "edge_types e t \<equiv> card {v. (v,t) \<in> e}"

lemma has_edge: "\<lbrakk>finite e; edge_types e t > 0\<rbrakk> \<Longrightarrow> \<exists>
    v. (v,t) \<in> e"
apply (simp add: edge_types_def)
apply (case_tac "card {v. (v, t) \<in> e}", auto simp add: card_Suc_eq)
done

abbreviation "no_edges \<equiv> \<lambda>e. 0"

lemma out_one [simp]: "finite (Edges G) \<Longrightarrow> (edge_types (out_edges (Edges G)
    n) = no_edges(ty := Suc 0)) =
 (\<exists>m. out_edges (Edges G) n = {(m, ty)})"
apply (rule iffI, frule_tac x=ty in cong, simp, simp (no_asm_use) add: edge_types_def)
apply (clarsimp simp add: card_Suc_eq)
apply (rule_tac x=b in exI, clarsimp simp add: set_eq_iff out_edges_def)
apply (subgoal_tac "finite {v. (n, v, ba) \<in> Edges G}")
apply (drule_tac x=ba in cong, simp, clarsimp simp add: edge_types_def split: if_splits)
apply (simp, erule finite_vimageI, simp add: inj_on_def)
apply (rule ext, clarsimp simp add: edge_types_def)
done

corollary out_one' [simp]: "finite (Edges G) \<Longrightarrow> (edge_types {(u, t). (n, u,
    t) \<in> Edges G} = no_edges(seq := Suc 0)) =
 (\<exists>m. (n, m, seq) \<in> Edges G \<and> (\<forall>u t. (n, u, t) \<in> Edges G \<
     longrightarrow> u = m \<and> t = seq))"
by (drule_tac n=n and ty=seq in out_one, auto simp add: out_edges_def)

lemma out_two [simp]: "\<lbrakk>finite (Edges G); t1 \<noteq> t2\<rbrakk> \<Longrightarrow>
     (edge_types (out_edges (Edges G) n) =
 no_edges(t1 := Suc 0, t2 := Suc 0)) =
 (\<exists>m1 m2. out_edges (Edges G) n = {(m1, t1), (m2, t2)})"
apply (rule iffI, frule_tac x=t1 in cong, simp, simp (no_asm_use) add: edge_types_def)
apply (frule_tac x=t2 in cong, simp, simp (no_asm_use) add: edge_types_def)
apply (clarsimp simp add: card_Suc_eq)
apply (rule_tac x=b in exI, rule_tac x=ba in exI, clarsimp simp add: set_eq_iff
    out_edges_def)
```

```
apply (subgoal_tac "finite {v. (n, v, bb) \<in> Edges G}")
apply (drule_tac x=bb in cong, simp, clarsimp simp add: edge_types_def split: if_splits)
apply (simp, erule finite_vimageI, simp add: inj_on_def)
apply (rule ext, clarsimp simp add: edge_types_def)
done

(* Flowgraphs label nodes with instructions. *)
locale flowgraph = pointed_graph where Edges="Edges::('node, 'edge_type) edge set" for
    Edges +
 fixes L::"'node \<Rightarrow> 'instr" and Seq::'edge_type
(* The relationship between instruction and out-edges of a node, defined per language. *)
   and instr_edges::"'instr \<Rightarrow> ('edge_type \<Rightarrow> nat) set"
 assumes instr_edges_ok: "\<lbrakk>u \<in> Nodes; u \<noteq> Exit\<rbrakk> \<Longrightarrow
     > edge_types (out_edges Edges u) \<in> instr_edges (L u)"
     and no_loop: "(u, u, Seq) \<notin> Edges"

record ('node, 'edge_type, 'instr) flowgraph =
 "('node, 'edge_type) doubly_pointed_graph" +
 Label :: "'node \<Rightarrow> 'instr"

thm flowgraph_def

definition is_flowgraph where
"is_flowgraph g seq InstrEdges \<equiv> flowgraph (Nodes g) (Start g) (Exit g) (Edges g) (
    Label g) seq InstrEdges"

(* Extraction functions for tCFGs; don't need to be in the locale. *)
definition nodes where "nodes CFGs \<equiv> \<Union>(Nodes ` ran CFGs)"
definition edges where "edges CFGs \<equiv> \<Union>(Edges ` ran CFGs)"
definition starts where "starts CFGs \<equiv> Start ` ran CFGs"
definition exits where "exits CFGs \<equiv> Exit ` ran CFGs"
definition thread_of where "thread_of n CFGs \<equiv> THE t. \<exists>g. CFGs t = Some g \<
    and> n \<in> Nodes g"
definition label where "label n CFGs \<equiv> case CFGs (thread_of n CFGs) of Some g \<
    Rightarrow> Label g n"
definition start_of where "start_of n CFGs \<equiv> case CFGs (thread_of n CFGs) of Some g
    \<Rightarrow> Start g"
definition exit_of where "exit_of n CFGs \<equiv> case CFGs (thread_of n CFGs) of Some g \<
    Rightarrow> Exit g"
```

```
lemma ran_dom: "f ` dom f = Some ` ran f"
apply (auto simp add: dom_def ran_def)
apply force
done

lemma finite_ran_dom: "finite (dom f) \<Longrightarrow> finite (ran f)"
apply (drule_tac h=f in finite_imageI)
apply (simp add: ran_dom)
apply (drule finite_imageD, simp+)
done

lemma nodes_Union: "nodes CFGs = (\<Union>t\<in>dom CFGs. case CFGs t of Some g \<
    Rightarrow> Nodes g)"
by (auto simp add: nodes_def ran_def)

lemma nodes_by_graph [intro]: "\<lbrakk>CFGs t = Some g; n \<in> Nodes g\<rbrakk> \<
    Longrightarrow> n \<in> nodes CFGs"
by (auto simp add: nodes_def ran_def)

lemma starts_by_graph [intro]: "CFGs t = Some g \<Longrightarrow> Start g \<in> starts CFGs
    "
by (auto simp add: starts_def ran_def)

lemma exits_by_graph [intro]: "CFGs t = Some g \<Longrightarrow> Exit g \<in> exits CFGs"
by (auto simp add: exits_def ran_def)

definition safe_points where "safe_points CFGs ps \<equiv>
 \<forall>t G n. CFGs t = Some G \<and> ps t = Some n \<longrightarrow> n \<in> Nodes G"
definition start_points where "start_points CFGs t \<equiv> case CFGs t of Some G \<
    Rightarrow> Some (Start G)
 | _ \<Rightarrow> None"
definition end_points where "end_points CFGs t \<equiv> case CFGs t of Some G \<Rightarrow>
     Some (Exit G)
 | _ \<Rightarrow> None"

lemma add_no_nodes [simp]: "\<lbrakk>CFGs t = Some G; Nodes G' = Nodes G\<rbrakk> \<
    Longrightarrow>
 nodes (CFGs(t \<mapsto> G')) = nodes CFGs"
apply (auto simp add: nodes_def ran_def)
apply (case_tac "a = t", auto)
```

```
done

definition "default_state CFGs n \<equiv> (start_points CFGs)(thread_of n CFGs \<mapsto> n)
    "



(* A tCFG has one CFG per thread name. *)
locale tCFG =
 fixes CFGs::"('thread, ('node, 'edge_type, 'instr) flowgraph) map"
   and instr_edges::"'instr \<Rightarrow> ('edge_type \<Rightarrow> nat) set"
   and Seq::'edge_type
 assumes CFGs: "CFGs t = Some g \<Longrightarrow> is_flowgraph g Seq instr_edges"
     and disjoint: "\<lbrakk>CFGs t = Some g; CFGs t' = Some g'; t \<noteq> t'\<rbrakk> \<
         Longrightarrow>
                     Nodes g \<inter> Nodes g' = {}"
     and finite_threads [simp]: "finite (dom CFGs)"
begin

lemma thread_of_correct: "\<lbrakk>CFGs t = Some g; n \<in> Nodes g\<rbrakk> \<
    Longrightarrow> thread_of n CFGs = t"
apply (simp add: thread_of_def, rule the_equality, auto)
apply (rule ccontr, drule disjoint, simp+, blast)
done

lemma node_of_graph: "\<lbrakk>n \<in> nodes CFGs; CFGs (thread_of n CFGs) = Some G\<rbrakk
    > \<Longrightarrow>
 n \<in> Nodes G"
by (clarsimp simp add: nodes_def ran_def thread_of_correct)

lemma label_correct: "\<lbrakk>CFGs t = Some g; n \<in> Nodes g\<rbrakk> \<Longrightarrow>
    label n CFGs = Label g n"
by (simp add: label_def thread_of_correct)

lemma start_of_correct: "\<lbrakk>CFGs t = Some g; n \<in> Nodes g\<rbrakk> \<
    Longrightarrow> start_of n CFGs = Start g"
by (simp add: start_of_def thread_of_correct)

lemma exit_of_correct: "\<lbrakk>CFGs t = Some g; n \<in> Nodes g\<rbrakk> \<Longrightarrow
    > exit_of n CFGs = Exit g"
by (simp add: exit_of_def thread_of_correct)
```

```
lemma nodes_finite: "finite (nodes CFGs)"
apply (simp add: nodes_Union, rule) (* rule? *)
apply (clarsimp simp add: dom_def)
apply (drule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def pointed_graph_def)
done

lemma edges_disjoint: "\<lbrakk>CFGs t = Some g; CFGs t' = Some g'; t \<noteq> t'\<rbrakk>
    \<Longrightarrow>
                      Edges g \<inter> Edges g' = {}"
apply (frule CFGs, frule_tac t=t' in CFGs)
apply (drule disjoint, simp+)
apply (clarsimp simp add: is_flowgraph_def flowgraph_def pointed_graph_def)
apply force
done

lemma starts_and_exits_disjoint: "\<lbrakk>CFGs t = Some g; CFGs t' = Some g'; t \<noteq> t
    '\<rbrakk> \<Longrightarrow>
                      Start g \<noteq> Start g' \<and> Start g \<noteq> Exit g' \<and> Exit
                          g \<noteq> Start g' \<and> Exit g \<noteq> Exit g'"
apply (frule CFGs, frule_tac t=t' in CFGs)
apply (drule disjoint, simp+)
apply (auto simp add: is_flowgraph_def flowgraph_def pointed_graph_def)
done

lemma safe_start [simp]: "safe_points CFGs (start_points CFGs)"
apply (clarsimp simp add: safe_points_def start_points_def)
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def pointed_graph_def)
done

lemma safe_end [simp]: "safe_points CFGs (end_points CFGs)"
apply (clarsimp simp add: safe_points_def end_points_def)
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def pointed_graph_def)
done

(* Infinite paths through a tCFG. *)
definition Paths where "Paths q \<equiv>
{path. path 0 = q \<and> (\<forall>i. \<forall>t. path (Suc i) t = path i t \<or>
 (\<exists>n G n' e. path i t = Some n \<and> CFGs t = Some G \<and> path (Suc i) t = Some
     n' \<and> (n, n', e) \<in> Edges G))}"
```

```
lemma Path_first [simp]: "l \<in> Paths q \<Longrightarrow> l 0 = q"
by (clarsimp simp add: Paths_def)

lemma Path_domain1: "l \<in> Paths q \<Longrightarrow> dom (l i) = dom q"
apply (induct i, simp+)
apply (clarsimp simp only: Paths_def dom_def, rule set_eqI, clarsimp)
apply (erule_tac x=i in allE)
apply ((erule_tac x=x in allE)+, erule disjE, clarsimp, blast, clarsimp, blast)
done

lemma Path_domain: "\<lbrakk>l \<in> Paths q; dom q = dom CFGs\<rbrakk> \<Longrightarrow>
    dom (l i) = dom CFGs"
by (simp add: Path_domain1)

lemma Path_next [simp]: "\<lbrakk>l \<in> Paths q; l i t = Some n\<rbrakk> \<Longrightarrow
    >
 \<exists>n'. l (Suc i) t = Some n' \<and> (n' = n \<or> (\<exists>G e. CFGs t = Some G \<
     and> (n, n', e) \<in> Edges G))"
apply (clarsimp simp only: Paths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
apply (erule disjE, clarsimp+)
done

lemma Path_next_None: "\<lbrakk>l \<in> Paths q; l i t = None\<rbrakk> \<Longrightarrow> l
    (Suc i) t = None"
apply (clarsimp simp only: Paths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
done

lemma Path_prev [simp]: "\<lbrakk>l \<in> Paths q; l (Suc i) t = Some n\<rbrakk> \<
    Longrightarrow>
 \<exists>n'. l i t = Some n' \<and> (n' = n \<or> (\<exists>G e. CFGs t = Some G \<and> (n
     ', n, e) \<in> Edges G))"
apply (clarsimp simp only: Paths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
apply (erule disjE, force, clarsimp)
done
```

```
lemma Path_prev_None: "\<lbrakk>l \<in> Paths q; l (Suc i) t = None\<rbrakk> \<
    Longrightarrow> l i t = None"
apply (clarsimp simp only: Paths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
done

lemma Path_safe: "\<lbrakk>l \<in> Paths q; safe_points CFGs q\<rbrakk> \<Longrightarrow>
    safe_points CFGs (l i)"
apply (induct i, auto)
apply (clarsimp simp add: safe_points_def)
apply (erule_tac x=t in allE, clarsimp)
apply (drule Path_prev, auto)
apply (drule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def)
apply (drule pointed_graph.edges_ok, simp+)
done

lemma exists_path: "\<exists>l. l \<in> Paths q"
by (rule_tac x="\<lambda>i. q" in exI, simp only: Paths_def, simp)

lemma specify_path: "\<forall>l\<in>Paths q. P l \<Longrightarrow> \<exists>l\<in>Paths q.
    P l"
by (auto simp add: exists_path)

lemma path_incremental: "\<lbrakk>CFGs t = Some G; \<exists>e. (n, n', e) \<in> Edges G; l
    \<in> Paths q;
 q t = Some n'\<rbrakk> \<Longrightarrow> [q(t \<mapsto> n)] \<frown> l \<in> Paths (q(t \<
     mapsto> n))"
apply (clarsimp simp only: Paths_def, clarsimp simp add: i_append_def)
apply (rule conjI, force, force)
done

lemma path_incremental_gen: "\<lbrakk>l \<in> Paths q';
 \<forall>t. q t = q' t \<or> (\<exists>G n n' e. CFGs t = Some G \<and> q t = Some n \<and
     > q' t = Some n' \<and> (n, n', e) \<in> Edges G)\<rbrakk> \<Longrightarrow>
 [q] \<frown> l \<in> Paths q"
apply (clarsimp simp only: Paths_def, clarsimp simp add: i_append_def)
apply (rule conjI, clarsimp)
apply (erule_tac x=t in allE, force)
apply clarsimp
apply (erule_tac x="i - 1" in allE, (erule_tac x=t in allE)+, clarsimp)
```

```
done

lemma path_incremental_segment: "\<lbrakk>l \<in> Paths q;
 \<forall>i<length l'. \<forall>t. (l' ! i) t = ((l' @ [q]) ! Suc i) t \<or>
 (\<exists>G n n' e. CFGs t = Some G \<and> (l' ! i) t = Some n \<and> ((l' @ [q]) ! Suc i)
       t = Some n' \<and> (n, n', e) \<in> Edges G)\<rbrakk> \<Longrightarrow>
 l' \<frown> l \<in> Paths ((l' @ [q]) ! 0)"
apply (induct l', auto)
apply (subgoal_tac "l' \<frown> l \<in> Paths ((l' @ [q]) ! 0)", rule_tac P="\<lambda>x. x
    \<in> Paths a" in i_append_Cons [THEN sym [THEN subst]])
apply (rule path_incremental_gen, simp, force)
apply (subgoal_tac "\<forall>i<length l'. \<forall>t. (l' ! i) t = ((l' @ [q]) ! Suc i) t
    \<or>
 (\<exists>G. CFGs t = Some G \<and> (\<exists>n. (l' ! i) t = Some n \<and> (\<exists>n'.
     ((l' @ [q]) ! Suc i) t = Some n' \<and> (\<exists>e. (n, n', e) \<in> Edges G))))",
 simp, force)
done

lemma path_suffix: "l \<in> Paths q \<Longrightarrow> l \<Up> i \<in> Paths (l i)"
by (simp only: Paths_def, simp)

lemma combine_paths: "\<lbrakk>l \<in> Paths q; l' \<in> Paths (l i)\<rbrakk> \<
    Longrightarrow>
 (l \<Down> i) \<frown> l' \<in> Paths q"
apply (frule_tac l=l' and l'="l \<Down> i" in path_incremental_segment, simp_all, clarsimp)
apply (case_tac "l ia t", drule Path_next_None, simp+)
apply (clarsimp simp add: nth_append)
apply (case_tac "Suc ia = i", simp+)
apply (drule Path_next, simp+)
apply (clarsimp simp add: nth_append)
apply (case_tac "Suc ia = i", simp+)
apply (erule disjE, clarsimp+)+
apply (simp add: nth_append)
apply (case_tac i, simp+)
done

lemma path_plus_edge: "\<lbrakk>l \<in> Paths q; l i t = Some n; CFGs t = Some G; (n, n', e
    ) \<in> Edges G\<rbrakk> \<Longrightarrow>
 \<exists>l'. (l \<Down> i) \<frown> [l i] \<frown> l' \<in> Paths q \<and> l' 0 t = Some n
     '"
```

```
apply (cut_tac q="l i(t \<mapsto> n')" in exists_path, clarsimp)
apply (frule_tac l=la in path_incremental, force, simp+)
apply (simp add: map_upd_triv, drule combine_paths, simp, force)
done

definition RPaths where "RPaths q \<equiv>
{path. path 0 = q \<and> (\<forall>i. \<forall>t. path (Suc i) t = path i t \<or>
 (\<exists>n G n' e. path i t = Some n \<and> CFGs t = Some G \<and> path (Suc i) t = Some
     n' \<and> (n', n, e) \<in> Edges G)) \<and>
 (\<forall>t. (\<exists>p. q t = Some p) \<longrightarrow> (\<exists>i. path i t = Some (
     Start (the (CFGs t)))))}"

lemma RPath_first [simp]: "l \<in> RPaths q \<Longrightarrow> l 0 = q"
by (clarsimp simp add: RPaths_def)

lemma RPath_domain1 [simp]: "l \<in> RPaths q \<Longrightarrow> dom (l i) = dom q"
apply (induct i, simp+)
apply (clarsimp simp only: RPaths_def dom_def, rule set_eqI, clarsimp)
apply (erule_tac x=i in allE)
apply ((erule_tac x=x in allE)+, erule disjE, clarsimp, blast, clarsimp)
apply blast
done

lemma RPath_domain [simp]: "\<lbrakk>l \<in> RPaths q; dom q = dom CFGs\<rbrakk> \<
    Longrightarrow> dom (l i) = dom CFGs"
by simp

lemma RPath_next [simp]: "\<lbrakk>l \<in> RPaths q; l i t = Some n\<rbrakk> \<
    Longrightarrow>
 \<exists>n'. l (Suc i) t = Some n' \<and> (n' = n \<or> (\<exists>G e. CFGs t = Some G \<
     and> (n', n, e) \<in> Edges G))"
apply (clarsimp simp only: RPaths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
apply (erule disjE, clarsimp+)
done

lemma RPath_next_None: "\<lbrakk>l \<in> RPaths q; l i t = None\<rbrakk> \<Longrightarrow>
    l (Suc i) t = None"
apply (clarsimp simp only: RPaths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
```

```
done

lemma RPath_prev [simp]: "\<lbrakk>l \<in> RPaths q; l (Suc i) t = Some n\<rbrakk> \<
    Longrightarrow>
 \<exists>n'. l i t = Some n' \<and> (n' = n \<or> (\<exists>G e. CFGs t = Some G \<and> (n
     , n', e) \<in> Edges G))"
apply (clarsimp simp only: RPaths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
apply (erule disjE, force, clarsimp)
done

lemma RPath_prev_None: "\<lbrakk>l \<in> RPaths q; l (Suc i) t = None\<rbrakk> \<
    Longrightarrow>
 l i t = None"
apply (clarsimp simp only: RPaths_def)
apply (erule_tac x=i in allE, clarsimp, (erule_tac x=t in allE)+, clarsimp)
done

lemma RPath_safe: "\<lbrakk>l \<in> RPaths q; safe_points CFGs q\<rbrakk> \<Longrightarrow>
     safe_points CFGs (l i)"
apply (induct i, auto)
apply (clarsimp simp add: safe_points_def)
apply (erule_tac x=t in allE, clarsimp)
apply (drule RPath_prev, auto)
apply (drule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def)
apply (drule pointed_graph.edges_ok, simp+)
done

lemma rpath_incremental: "\<lbrakk>CFGs t = Some G; \<exists>e. (n', n, e) \<in> Edges G; l
     \<in> RPaths q;
 q t = Some n'\<rbrakk> \<Longrightarrow> [q(t \<mapsto> n)] \<frown> l \<in> RPaths (q(t
     \<mapsto> n))"
apply (clarsimp simp only: RPaths_def, clarsimp simp add: i_append_def)
apply (rule conjI, clarsimp, rule conjI, force, clarsimp)
apply (metis diff_Suc_Suc gr0_conv_Suc minus_nat.diff_0)
by (metis Suc_neq_Zero diff_Suc_Suc minus_nat.diff_0 the.simps)

lemma rpath_incremental_gen: "\<lbrakk>l \<in> RPaths q';
 \<forall>t. q t = q' t \<or> (\<exists>G n n' e. CFGs t = Some G \<and> q t = Some n \<and
     > q' t = Some n' \<and> (n', n, e) \<in> Edges G)\<rbrakk> \<Longrightarrow>
```

```
 [q] \<frown> l \<in> RPaths q"
apply (clarsimp simp only: RPaths_def, clarsimp simp add: i_append_def)
apply (rule conjI, clarsimp, rule conjI, clarsimp)
apply (erule_tac x=t in allE, force)
apply (metis diff_Suc_Suc gr0_conv_Suc minus_nat.diff_0)
apply clarsimp
apply (erule_tac x="ia - 1" in allE, (erule_tac x=t in allE)+, clarsimp)
apply force
done

lemma rpath_incremental_segment: "\<lbrakk>l \<in> RPaths q;
 \<forall>i<length l'. \<forall>t. (l' ! i) t = ((l' @ [q]) ! Suc i) t \<or>
 (\<exists>G n n' e. CFGs t = Some G \<and> (l' ! i) t = Some n \<and> ((l' @ [q]) ! Suc i)
      t = Some n' \<and> (n', n, e) \<in> Edges G)\<rbrakk> \<Longrightarrow>
 l' \<frown> l \<in> RPaths ((l' @ [q]) ! 0)"
apply (induct l', auto)
apply (subgoal_tac "l' \<frown> l \<in> RPaths ((l' @ [q]) ! 0)", rule_tac P="\<lambda>x. x
    \<in> RPaths a" in i_append_Cons [THEN sym [THEN subst]])
apply (rule rpath_incremental_gen, simp, force)
apply (subgoal_tac "\<forall>i<length l'. \<forall>t. (l' ! i) t = ((l' @ [q]) ! Suc i) t
   \<or>
 (\<exists>G. CFGs t = Some G \<and> (\<exists>n. (l' ! i) t = Some n \<and> (\<exists>n'.
    ((l' @ [q]) ! Suc i) t = Some n' \<and> (\<exists>e. (n', n, e) \<in> Edges G))))",
 simp, force)
done

lemma combine_rpaths: "\<lbrakk>l \<in> RPaths q; l' \<in> RPaths (l i)\<rbrakk> \<
   Longrightarrow>
 (l \<Down> i) \<frown> l' \<in> RPaths q"
apply (frule_tac l=l' and l'="l \<Down> i" in rpath_incremental_segment, simp_all, clarsimp
   )
apply (case_tac "l ia t", drule RPath_next_None, simp+)
apply (clarsimp simp add: nth_append)
apply (case_tac "Suc ia = i", simp+)
apply (drule RPath_next, simp+)
apply (clarsimp simp add: nth_append)
apply (case_tac "Suc ia = i", simp+)
apply (erule disjE, clarsimp+)+
apply (simp add: nth_append)
apply (case_tac i, simp+)
```

```
done

lemma start_rpath: "(\<lambda>i. start_points CFGs) \<in> RPaths (start_points CFGs)"
by (simp only: RPaths_def, simp add: start_points_def split: option.splits)

lemma start_one_rpath: "CFGs t = Some G \<Longrightarrow> (\<lambda>i. [t \<mapsto> Start G
    ]) \<in> RPaths [t \<mapsto> Start G]"
by (simp only: RPaths_def, clarsimp)

lemma start_some_rpath: "(\<lambda>i t. if t \<in> T then Some (Start (the (CFGs t))) else
    None) \<in>
 RPaths (\<lambda>t. if t \<in> T then Some (Start (the (CFGs t))) else None)"
by (simp only: RPaths_def, clarsimp)

lemma specify_start_rpath: "\<forall>l\<in>RPaths (start_points CFGs). P l \<Longrightarrow
    > P (\<lambda>i. start_points CFGs)"
by (simp add: start_rpath)

lemma reverse_path: "l \<in> Paths (start_points CFGs) \<Longrightarrow> \<exists>l'\<in>
    RPaths (l i). \<forall>j\<le>i. l' j = l (i - j)"
apply (cut_tac start_rpath)
apply (drule_tac l'="rev (i_take (Suc i) l)" in rpath_incremental_segment, clarsimp)
apply (clarsimp simp add: rev_nth nth_append)
apply (case_tac i, simp+)
apply (cut_tac n=ia and m=nat in Suc_diff_le, simp+)
apply (case_tac "l (Suc nat - ia) t", simp, drule Path_prev_None, simp+)
apply (drule Path_prev, simp+, force)
apply (rule_tac x="rev (l \<Down> Suc i) \<frown> (\<lambda>i. start_points CFGs)" in bexI,
     auto simp add: nth_append rev_nth)
done

lemma reverse_path_gen: "l \<in> Paths (\<lambda>t. if t \<in> T then Some (Start (the (
    CFGs t))) else None) \<Longrightarrow>
 \<exists>l'\<in>RPaths (l i). \<forall>j\<le>i. l' j = l (i - j)"
apply (cut_tac T=T in start_some_rpath)
apply (drule_tac l'="rev (i_take (Suc i) l)" in rpath_incremental_segment, clarsimp)
apply (clarsimp simp add: rev_nth nth_append)
apply (case_tac i, simp+)
apply (cut_tac n=ia and m=nat in Suc_diff_le, simp+)
apply (case_tac "l (Suc nat - ia) t", simp, drule Path_prev_None, simp+)
```

```
apply (drule Path_prev, simp+, force)
apply (rule_tac x="rev (l \<Down> Suc i) \<frown> (\<lambda>i t. if t \<in> T then Some (
    Start (the (CFGs t))) else None)"
 in bexI, auto simp add: nth_append rev_nth)
done

lemma path_rpath: "\<lbrakk>l \<in> Paths q; l' \<in> RPaths q\<rbrakk> \<Longrightarrow>
    \<exists>l''. l'' \<in> RPaths (l i)"
apply (subgoal_tac "rev (l \<Up> 1 \<Down> i) @ [q] = rev (l \<Down> Suc i)")
apply (drule_tac l'="rev (l \<Up> 1 \<Down> i)" in rpath_incremental_segment, clarsimp simp
     add: rev_nth)
apply (case_tac "l (i - Suc ia) t", drule Path_next_None, simp+, drule Path_next, auto)
apply (clarsimp simp add: rev_nth nth_append, force)
apply (metis Path_first i_take_Suc rev.simps(2))
done

lemma specify_rpath: "\<lbrakk>\<forall>l\<in>RPaths q. P l; l \<in> Paths (start_points
    CFGs); l i = q\<rbrakk> \<Longrightarrow>
 \<exists>l'\<in>RPaths q. (\<forall>j\<le>i. l' j = l (i - j)) \<and> P l'"
by (drule_tac i=i in reverse_path, simp+)

lemma reverse_rpath: "l \<in> RPaths q \<Longrightarrow> \<exists>l'\<in>Paths (l i). \<
    forall>j\<le>i. l' j = l (i - j)"
apply (cut_tac q=q in exists_path, clarsimp)
apply (drule_tac l'="rev (i_take (Suc i) l)" in path_incremental_segment, clarsimp)
apply (clarsimp simp add: rev_nth nth_append)
apply (case_tac i, simp+)
apply (cut_tac n=ia and m=nat in Suc_diff_le, simp+)
apply (case_tac "l (Suc nat - ia) t", simp, drule RPath_prev_None, simp+)
apply (drule RPath_prev, simp+, force)
apply (rule_tac x="rev (l \<Down> Suc i) \<frown> la" in bexI, auto simp add: nth_append
    rev_nth)
done

lemma after_exit_graph: "\<lbrakk>l \<in> Paths q; CFGs t = Some G; l i t = Some (Exit G);
    j \<ge> i\<rbrakk> \<Longrightarrow>
 l j t = Some (Exit G)"
apply (induct j, auto)
apply (case_tac "i = Suc j", auto)
apply (drule_tac i=j in Path_next, simp+, clarsimp)
```

```
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def, drule pointed_graph.
    exit_last,
 simp add: out_edges_def)
done

lemma after_exit: "\<lbrakk>l \<in> Paths q; l i = end_points CFGs; j \<ge> i\<rbrakk> \<
    Longrightarrow> l j = end_points CFGs"
apply (induct j, auto)
apply (case_tac "i = Suc j", auto)
apply (subgoal_tac "l j = end_points CFGs", rule ext, case_tac "l j x")
apply (drule_tac i=j in Path_next_None, simp, simp)
apply (drule_tac i=j in Path_next, simp+, clarsimp simp add: end_points_def)
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def, drule pointed_graph.
    exit_last,
 simp add: out_edges_def)
apply (case_tac "i = j", auto)
done

lemma rpath_end: "\<lbrakk>l \<in> RPaths q; q t \<noteq> None\<rbrakk> \<Longrightarrow>
    \<exists>i. l i t = Some (Start (the (CFGs t)))"
by (simp only: RPaths_def, clarsimp)

lemma before_start_thread: "\<lbrakk>l \<in> RPaths q; l i t = Some (Start (the (CFGs t)));
     j \<ge> i\<rbrakk> \<Longrightarrow>
 l j t = Some (Start (the (CFGs t)))"
apply (induct j, auto)
apply (case_tac "i = Suc j", auto)
apply (drule_tac i=j in RPath_next, simp+, clarsimp)
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def, drule pointed_graph.
    start_first,
 simp add: in_edges_def)
done

lemma before_start_graph: "\<lbrakk>l \<in> RPaths q; CFGs t = Some G; l i t = Some (Start
    G); j \<ge> i\<rbrakk> \<Longrightarrow>
 l j t = Some (Start G)"
apply (induct j, auto)
apply (case_tac "i = Suc j", auto)
apply (drule_tac i=j in RPath_next, simp+, clarsimp)
```

```
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def, drule pointed_graph.
    start_first,
 simp add: in_edges_def)
done

lemma before_start: "\<lbrakk>l \<in> RPaths q; l i = start_points CFGs; j \<ge> i\<rbrakk>
     \<Longrightarrow> l j = start_points CFGs"
apply (induct j, auto)
apply (case_tac "i = Suc j", auto)
apply (subgoal_tac "l j = start_points CFGs", rule ext, case_tac "l j x")
apply (drule_tac i=j in RPath_next_None, simp, simp)
apply (drule_tac i=j in RPath_next, simp+, clarsimp simp add: start_points_def)
apply (frule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def, drule pointed_graph.
    start_first,
 simp add: in_edges_def)
apply (case_tac "i = j", auto)
done

lemma rpath_suffix: "l \<in> RPaths q \<Longrightarrow> l \<Up> i \<in> RPaths (l i)"
apply (frule_tac i=i in RPath_domain1)
apply (simp only: RPaths_def, clarsimp)
apply (case_tac "l 0 t")
apply (metis domIff option.distinct(1))
apply (rule_tac x=t in allE, simp, erule impE, simp, clarify)
apply (rule_tac x="ia - i" in exI)
apply (case_tac "ia \<ge> i", simp+)
apply (cut_tac l=l and q="l 0" and j=i in before_start_thread, simp only: RPaths_def, simp
    +)
done

lemma path_by_thread: "\<lbrakk>l \<in> Paths q; q' t = q t\<rbrakk> \<Longrightarrow> \<
    exists>l'\<in>Paths q'. \<forall>i. l' i t = l i t"
apply (cut_tac q=q' in exists_path, clarsimp)
apply (rule_tac x="\<lambda>i t'. if t' = t then l i t else la i t'" in bexI, simp)
apply (simp only: Paths_def, clarsimp)
apply (rule ext, simp)
done

lemma path_one_thread: "\<lbrakk>l \<in> Paths q; q t = Some n\<rbrakk> \<Longrightarrow>
 (\<lambda>i t'. if t = t' then l i t else None) \<in> Paths [t \<mapsto> n]"
```

```
by (simp only: Paths_def, clarsimp intro!: ext)

lemma rpath_by_thread: "\<lbrakk>l \<in> RPaths q; l' \<in> RPaths q'; q' t = q t\<rbrakk>
    \<Longrightarrow> \<exists>l'\<in>RPaths q'. \<forall>i. l' i t = l i t"
apply (rule_tac x="\<lambda>i t'. if t' = t then l i t else l' i t'" in bexI, simp)
apply (simp only: RPaths_def, clarsimp)
apply (rule conjI, rule ext, simp, clarsimp)
by metis

lemma rpath_one_thread: "\<lbrakk>l \<in> RPaths q; q t = Some n\<rbrakk> \<Longrightarrow>
 (\<lambda>i t'. if t = t' then l i t else None) \<in> RPaths [t \<mapsto> n]"
by (simp only: RPaths_def, clarsimp intro!: ext)

lemma rpath_start: "\<lbrakk>l \<in> RPaths q; finite (dom q)\<rbrakk> \<Longrightarrow>
 \<exists>l'\<in>Paths (\<lambda>t. if t \<in> dom q then Some (Start (the (CFGs t))) else
     None). \<exists>i'. l i = l' i'"
apply (frule_tac i="SOME i. l i = (\<lambda>t. if t \<in> dom q then Some (Start (the (CFGs
     t))) else None)"
 in reverse_rpath, clarsimp)
apply (cut_tac P="\<lambda>i. l i = (\<lambda>t. if t \<in> dom q then Some (Start (the (
    CFGs t))) else None)"
 in someI_ex)
apply (rule_tac x="Max {LEAST i. l i t = Some (Start (the (CFGs t))) | t. t \<in> dom q}"
    in exI,
 rule ext, clarsimp)
apply (rule conjI, clarsimp)
apply (frule_tac t=t in rpath_end, simp+, clarsimp)
apply (drule_tac P="\<lambda>i. l i t = Some (Start (the (CFGs t)))" in LeastI)
apply (rule before_start_thread, simp+)
apply (cut_tac A="{LEAST i. l i t = Some (Start (the (CFGs t))) |t. t \<in> dom q}" and
 k="LEAST x. l x t = Some (Start (the (CFGs t)))" in wellorder_Max_lemma, force, simp+)
apply (drule_tac i="Max {LEAST i. l i t = Some (Start (the (CFGs t))) |t. t \<in> dom q}"
    in
 RPath_domain1,force simp add: domIff)
apply clarsimp
apply (rule bexI, simp_all, rule_tac x="(SOME i. l i = (\<lambda>t. if t \<in> dom q then
 Some (Start (the (CFGs t))) else None)) - i" in exI, clarsimp)
apply (case_tac "i < (SOME i. l i = (\<lambda>t. if t \<in> dom q then Some (Start (the (
    CFGs t))) else None))",
 simp+)
```

```
apply (rule ext, simp, rule conjI, clarsimp)
apply (rule_tac i="SOME x. l x = (\<lambda>t. if t \<in> dom q then Some (Start (the (CFGs
    t))) else None)"
 in before_start_thread, simp+, simp_all)
apply (simp add: dom_def)
apply (drule_tac i=i in RPath_domain1, force simp add: domIff)
done

(* Transferring paths to other CFGs with the same structure *)
lemma path_by_thread_gen: "\<lbrakk>l \<in> Paths q; tCFG CFGs' instr_edges seq; CFGs' t =
    CFGs t; q' t = q t\<rbrakk> \<Longrightarrow>
 \<exists>l'\<in>tCFG.Paths CFGs' q'. \<forall>i. l' i t = l i t"
apply (frule_tac q=q' in tCFG.exists_path, clarsimp)
apply (rule_tac x="\<lambda>i t'. if t' = t then l i t else la i t'" in bexI, simp)
apply (simp only: tCFG.Paths_def, clarsimp)
apply (rule conjI, rule ext, simp)
apply clarsimp
apply (case_tac "l i t", auto)
apply (drule Path_next_None, simp+)
apply (drule Path_next, simp+, clarsimp)
done

lemma rpath_by_thread_gen: "\<lbrakk>l \<in> RPaths q; tCFG CFGs' instr_edges seq; CFGs' t
    = CFGs t;
 l' \<in> tCFG.RPaths CFGs' q'; q' t = q t; start_points CFGs' t = start_points CFGs t\<
     rbrakk> \<Longrightarrow>
 \<exists>l'\<in>tCFG.RPaths CFGs' q'. \<forall>i. l' i t = l i t"
apply (rule_tac x="\<lambda>i t'. if t' = t then l i t else l' i t'" in bexI, simp)
apply (simp only: tCFG.RPaths_def, clarsimp)
apply (rule conjI, rule ext, simp)
apply (rule conjI, clarsimp)
apply (case_tac "l i t", auto)
apply (drule RPath_next_None, simp+)
apply (drule RPath_next, simp+, clarsimp)
apply (clarsimp simp only: RPaths_def, clarsimp)
by metis

lemma path_G': "\<lbrakk>CFGs t = Some G; CFGs' t = Some (G\<lparr>Label := L'\<rparr>); l
    \<in> Paths q; q' t = q t;
```

```
 tCFG CFGs' instr_edges seq\<rbrakk> \<Longrightarrow> \<exists>l'\<in>tCFG.Paths CFGs' q'.
      \<forall>i. l' i t = l i t"
apply (frule_tac q=q' in tCFG.exists_path, clarsimp)
apply (rule_tac x="\<lambda>i t'. if t' = t then l i t else la i t'" in bexI, simp)
apply (simp only: tCFG.Paths_def, clarsimp)
apply (rule conjI, rule ext, simp)
apply clarsimp
apply (case_tac "l i t", auto)
apply (drule Path_next_None, simp+)
apply (drule Path_next, simp+, clarsimp)
done

lemma rpath_G': "\<lbrakk>CFGs t = Some G; CFGs' t = Some (G\<lparr>Label := L'\<rparr>); l
     \<in> Paths q; q' t = q t;
 tCFG CFGs' instr_edges seq\<rbrakk> \<Longrightarrow> \<exists>l'\<in>tCFG.Paths CFGs' q'.
      \<forall>i. l' i t = l i t"
apply (frule_tac q=q' in tCFG.exists_path, clarsimp)
apply (rule_tac x="\<lambda>i t'. if t' = t then l i t else la i t'" in bexI, simp)
apply (simp only: tCFG.Paths_def, clarsimp)
apply (rule conjI, rule ext, simp)
apply clarsimp
apply (case_tac "l i t", auto)
apply (drule Path_next_None, simp+)
apply (drule Path_next, simp+, clarsimp)
done

lemma path_CFGs': "\<lbrakk>\<forall>t G. CFGs t = Some G \<longrightarrow> (\<exists>L'.
    CFGs' t = Some (G\<lparr>Label := L'\<rparr>)); l \<in> Paths q;
 tCFG CFGs' instr_edges seq\<rbrakk> \<Longrightarrow> l \<in> tCFG.Paths CFGs' q"
by (simp only: tCFG.Paths_def Paths_def, force)

lemma rpath_CFGs': "\<lbrakk>\<forall>t G. CFGs t = Some G \<longrightarrow> (\<exists>L'.
    CFGs' t = Some (G\<lparr>Label := L'\<rparr>)); l \<in> RPaths q;
 tCFG CFGs' instr_edges seq; dom q \<subseteq> dom CFGs\<rbrakk> \<Longrightarrow> l \<in>
     tCFG.RPaths CFGs' q"
apply (simp only: tCFG.RPaths_def RPaths_def, auto)
apply force
apply (case_tac "CFGs t", auto)
apply (erule_tac x=t in allE, force)
done
```

```
lemma thread_of_path: "\<lbrakk>CFGs t = Some G; l i t = Some n; l \<in> Paths (
    start_points CFGs)\<rbrakk> \<Longrightarrow>
  thread_of n CFGs = t"
apply (frule_tac i=i in Path_safe, simp)
apply (force simp add: safe_points_def thread_of_correct)
done

end

end

(* trans_semantics.thy *)
(* William Mansky *)
(* Semantics for the revised specification language PTRANS. *)

theory trans_semantics
imports trans_flowgraph
begin

(* matches a total valuation of metavariables with a partial one *)
definition part_matches where "part_matches \<sigma> t \<equiv> \<forall>x y. t x = Some y
    \<longrightarrow> \<sigma> x = y"

(* extends a partial valuation with another valuation over a given domain *)
definition part_extend where "part_extend t d \<sigma> \<equiv> \<lambda>x. if x\<in>d then
     Some (\<sigma> x) else t x"

(* Infinite types and fresh objects. *)
primrec new::"'a set \<Rightarrow> nat \<Rightarrow> 'a list" where
"new S 0 = []" |
"new S (Suc n) = (let n' = SOME n. n \<notin> S in n' # new (S \<union> {n'}) n)"

lemmas fresh_new = ex_new_if_finite [THEN someI_ex]
lemma new_nodes_are_new: "\<lbrakk>\<not>finite (UNIV::'a set); finite (S::'a set); m \<in>
     set (new S n)\<rbrakk> \<Longrightarrow> m \<notin> S"
apply (induct n arbitrary: S, simp)
apply (simp add: Let_def)
apply (erule disjE, simp add: fresh_new)
apply (subgoal_tac "m \<notin> insert (SOME n. n \<notin> S) S", blast+)
done
```

```
lemma new_nodes_are_new2: "\<lbrakk>\<not>finite (UNIV::'a set); finite (S::'a set); m \<in
    > S\<rbrakk> \<Longrightarrow> m \<notin> set (new S n)"
by (clarsimp simp add: new_nodes_are_new)

lemma new_nodes_count [simp]: "length (new S n) = n"
by (induct n arbitrary: S, auto simp add: Let_def)

lemma new_nodes_diff: "\<lbrakk>\<not>finite (UNIV::'a set); finite (S::'a set)\<rbrakk> \<
    Longrightarrow> distinct (new S n)"
apply (induct n arbitrary: S, simp)
apply (clarsimp simp add: Let_def)
apply (cut_tac S="insert (SOME n. n \<notin> S) S" in new_nodes_are_new, simp+)
done

(* The pieces that must be provided by the target language to give semantics to
    transformations. *)
locale TRANS_basics = (* give SCEx a type annotation, parameterize by 'valuation? *)
  fixes subst::"(('thread, ('node, 'edge_type, 'instr) flowgraph) map) \<Rightarrow> ('
      metavar \<Rightarrow> 'object) \<Rightarrow> 'pattern \<rightharpoonup> 'instr"
    and node_subst::"(('thread, ('node, 'edge_type, 'instr) flowgraph) map) \<Rightarrow>
        ('metavar \<Rightarrow> 'object) \<Rightarrow>
                     'metavar node_lit \<rightharpoonup> 'node"
    and type_subst::"('metavar \<Rightarrow> 'object) \<Rightarrow> 'edge_type_expr \<
        rightharpoonup> 'edge_type"
    and pred_fv::"'pred \<Rightarrow> 'metavar set"
    and eval_pred::"(('thread, ('node, 'edge_type, 'instr) flowgraph) map) \<Rightarrow> ('
        thread, 'node) map \<Rightarrow>
                    ('metavar \<Rightarrow> 'object) \<Rightarrow> 'pred \<Rightarrow> bool
                        "
    and instr_edges
    and Seq::'edge_type
  assumes infinite_nodes [simp]: "\<not>finite (UNIV::'node set)"
      and infinite_mvars [simp]: "\<not>finite (UNIV::'metavar set)"
      and pred_same_subst: "\<lbrakk>\<forall>x\<in>pred_fv p. \<sigma> x = \<sigma>' x;
          tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
 eval_pred CFGs q \<sigma> p = eval_pred CFGs q \<sigma>' p"
begin

lemmas fresh_nodes = infinite_nodes [THEN fresh_new]
```

```
lemmas fresh_mvars = infinite_mvars [THEN fresh_new]

primrec subst_list where
"subst_list G \<sigma> [] = Some []" |
"subst_list G \<sigma> (p # rest) = (case (subst G \<sigma> p, subst_list G \<sigma> rest)
    of
   (Some i, Some irest) \<Rightarrow> Some (i # irest) | _ \<Rightarrow> None)"

primrec models where
"models _ _ _ SCTrue = True" |
"models CFGs \<sigma> q (SCPred p) = eval_pred CFGs q \<sigma> p" |
"models CFGs \<sigma> q (\<phi>1 \<and>sc \<phi>2) = (models CFGs \<sigma> q \<phi>1 \<and>
     models CFGs \<sigma> q \<phi>2)" |
"models CFGs \<sigma> q (\<not>sc \<phi>) = (\<not>models CFGs \<sigma> q \<phi>)" |
"models CFGs \<sigma> q (A \<phi>1 \<U> \<phi>2) = (\<forall>l \<in> tCFG.Paths CFGs q. \<
    exists>i. models CFGs \<sigma> (l i) \<phi>2 \<and>
 (\<forall>j<i. models CFGs \<sigma> (l j) \<phi>1))" |
"models CFGs \<sigma> q (E \<phi>1 \<U> \<phi>2) = (\<exists>l \<in> tCFG.Paths CFGs q. \<
    exists>i. models CFGs \<sigma> (l i) \<phi>2 \<and>
 (\<forall>j<i. models CFGs \<sigma> (l j) \<phi>1))" |
"models CFGs \<sigma> q (A \<phi>1 \<B> \<phi>2) = (\<forall>l \<in> tCFG.RPaths CFGs q. \<
    exists>i. models CFGs \<sigma> (l i) \<phi>2 \<and>
 (\<forall>j<i. models CFGs \<sigma> (l j) \<phi>1))" |
"models CFGs \<sigma> q (E \<phi>1 \<B> \<phi>2) = (\<exists>l \<in> tCFG.RPaths CFGs q. \<
    exists>i. models CFGs \<sigma> (l i) \<phi>2 \<and>
 (\<forall>j<i. models CFGs \<sigma> (l j) \<phi>1))" |
"models CFGs \<sigma> q (SCEx x \<phi>) = (\<exists>obj. models CFGs (\<sigma>(x := obj)) q
     \<phi>)"

abbreviation "side_cond_sf \<psi> \<sigma> CFGs \<equiv> models CFGs \<sigma> (start_points
     CFGs) \<psi>"

lemma cond_same_subst: "\<lbrakk>\<forall>x\<in>cond_fv pred_fv P. \<sigma> x = \<sigma>' x
    ; tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q P = models CFGs \<sigma>' q P"
apply (induct P arbitrary: \<sigma> \<sigma>' q, simp+)
apply (erule pred_same_subst, simp)
apply (metis (lifting, mono_tags) UnI1 UnI2 cond_fv.simps(3) models.simps(3))
apply simp
apply (smt UnI1 UnI2 cond_fv.simps(5) models.simps(5))
```

```
apply (smt UnI1 UnI2 cond_fv.simps(6) models.simps(6))
apply (smt UnI1 UnI2 cond_fv.simps(7) models.simps(7))
apply (smt UnI1 UnI2 cond_fv.simps(8) models.simps(8))
apply auto
apply (rule_tac x=obj in exI, smt DiffI fun_upd_def singletonE)+
done

lemma cond_gen: "\<lbrakk>models CFGs \<sigma> q P; \<forall>x\<in>cond_fv pred_fv P. \<
    sigma> x = \<sigma>' x; tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q P"
by (drule cond_same_subst, simp, force)

end

(* Utility functions for actions. *)
fun update_list where
"update_list f [] = f" |
"update_list f ((x,y) # rest) = update_list (f(x := y)) rest"

definition "remap_succ x y e \<equiv> let (n,s,t) = e in if x = n then (y,s,t) else e"

context TRANS_basics begin

definition "rep_edges es ll \<equiv> if ll = [] then es else
  {remap_succ (hd ll) (last ll) e | e. e\<in>es} \<union>
  {(u,v,Seq) | u v. (\<exists>a. ll!a = u \<and> ll!(a+1) = v \<and> (a+1) < length ll)}"

lemma rep_one_edge [simp]: "rep_edges es [n] = es"
by (auto simp add: rep_edges_def remap_succ_def)

lemma rep_out_edges [simp]: "n \<notin> set ll \<Longrightarrow> out_edges (rep_edges es ll
    ) n = out_edges es n"
apply (auto simp add: rep_edges_def out_edges_def remap_succ_def)
apply force+
done

lemma rep_out_by_t [simp]: "n \<notin> set ll \<Longrightarrow> out_by_t (rep_edges es ll)
    t n = out_by_t es t n"
by (simp add: out_edges_by_t)
```

```
(* Actions are defined by their effects on one of the CFGs in a tCFG. *)
fun action_sf where
"action_sf (AAddEdge n m e) \<sigma> CFGs = (case (node_subst CFGs \<sigma> n, node_subst
    CFGs \<sigma> m, type_subst \<sigma> e) of
   (Some u, Some v, Some ty) \<Rightarrow> if u \<notin> nodes CFGs then None else
   (case CFGs (thread_of u CFGs) of Some G \<Rightarrow>
    if v \<in> Nodes G then Some (CFGs(thread_of u CFGs \<mapsto> G\<lparr>Edges := Edges G
         \<union> {(u,v,ty)}\<rparr>)) else None
 | _ \<Rightarrow> None) | _ \<Rightarrow> None)" |
"action_sf (ARemoveEdge n m e) \<sigma> CFGs = (case (node_subst CFGs \<sigma> n,
    node_subst CFGs \<sigma> m, type_subst \<sigma> e) of
   (Some u, Some v, Some ty) \<Rightarrow> if u \<notin> nodes CFGs then None else
   (case CFGs (thread_of u CFGs) of Some G \<Rightarrow>
   Some (CFGs(thread_of u CFGs \<mapsto> G\<lparr>Edges := Edges G - {(u,v,ty)}\<rparr>)) |
        _ \<Rightarrow> None)
 | _ \<Rightarrow> None)" |
"action_sf (AReplace m pl) \<sigma> CFGs = (if pl = [] then (* remove node *)
 case node_subst CFGs \<sigma> m of Some l \<Rightarrow> if l \<notin> nodes CFGs \<or> l
     \<in> starts CFGs \<or> l \<in> exits CFGs then None
   (* What should we do with the in- and out-edges of a removed node? *)
   else (case CFGs (thread_of l CFGs) of Some G \<Rightarrow>
   Some (CFGs(thread_of l CFGs \<mapsto> G\<lparr>Nodes := Nodes G - {l}, Edges := {(u,v,t)
       . (u,v,t) \<in> Edges G \<and> u \<noteq> l \<and> v \<noteq> l}\<rparr>)) | _ \<
       Rightarrow> None) | _ \<Rightarrow> None
 else case (node_subst CFGs \<sigma> m, subst_list CFGs \<sigma> pl) of
   (Some l, Some il) \<Rightarrow> if l \<notin> nodes CFGs then None else
   (case CFGs (thread_of l CFGs) of Some G \<Rightarrow>
   let ll = new (nodes CFGs) (length il - 1)
    in Some (CFGs(thread_of l CFGs \<mapsto> G\<lparr>Nodes := Nodes G \<union> set ll,
        Edges := rep_edges (Edges G) (l#ll),
                        Label := update_list (Label G) (zip (l#ll) il)\<rparr>)) | _ \<
                            Rightarrow> None)
   | _ \<Rightarrow> None)" |
"action_sf (ASplitEdge n m e i) \<sigma> CFGs = (case (node_subst CFGs \<sigma> n,
    node_subst CFGs \<sigma> m,type_subst \<sigma> e,subst CFGs \<sigma> i) of
   (Some u, Some v, Some ty, Some j) \<Rightarrow>
   (if (u,v,ty) \<in> edges CFGs then (case CFGs (thread_of u CFGs) of Some G \<Rightarrow>
    (let q = hd (new (nodes CFGs) 1)
    in Some (CFGs(thread_of u CFGs \<mapsto> G\<lparr>Nodes := Nodes G \<union> {q}, Edges
        := (Edges G - {(u,v,ty)}) \<union> {(u,q,ty),(q,v,Seq)},
```

```
                        Label := (Label G)(q := j)\<rparr>))) | _ \<Rightarrow> None)
    else None) | _ \<Rightarrow> None)"

(* semantics for lists of actions *)
primrec part_comp :: "('a \<rightharpoonup> 'a) list \<Rightarrow> 'a \<rightharpoonup> 'a"
where
"part_comp [] x = Some x" |
"part_comp (f#rest) x = (case f x of None \<Rightarrow> None | Some y \<Rightarrow>
    part_comp rest y)"

definition action_list_sf
where "action_list_sf Al \<sigma> CFGs \<equiv> part_comp (map (\<lambda>a G. action_sf a
    \<sigma> G) Al) CFGs"

(* Recursive (fixpoint) application of a transformation *)
inductive apply_some::"(('metavar \<rightharpoonup> 'object) \<Rightarrow> 'a \<Rightarrow>
     'a set) \<Rightarrow>
 ('metavar \<rightharpoonup> 'object) \<Rightarrow> 'a \<Rightarrow> 'a \<Rightarrow> bool"
      where
apply_none: "apply_some T \<tau> G G" |
apply_more: "\<lbrakk>G' \<in> T \<tau> G; apply_some T \<tau> G' G''\<rbrakk> \<
    Longrightarrow> apply_some T \<tau> G G''"

(* Transformations apply actions when their conditions are met. *)
primrec trans_sf where
"trans_sf (TIf al p) \<tau> CFGs = {CFGs' | CFGs' \<sigma>. Some CFGs' = action_list_sf al
    \<sigma> CFGs \<and>
    side_cond_sf p \<sigma> CFGs \<and> part_matches \<sigma> \<tau>}" |
"trans_sf (TMatch p T) \<tau> CFGs = {CFGs' | CFGs' \<sigma>. side_cond_sf p \<sigma> CFGs
    \<and> part_matches \<sigma> \<tau> \<and>
      CFGs' \<in> trans_sf T (part_extend \<tau> (cond_fv pred_fv p) \<sigma>) CFGs}" |
"trans_sf (TChoice T1 T2) \<tau> CFGs = (trans_sf T1 \<tau> CFGs) \<union> (trans_sf T2 \<
    tau> CFGs)" |
"trans_sf (TThen T1 T2) \<tau> CFGs = {CFGs'' | CFGs' CFGs''. CFGs' \<in> trans_sf T1 \<tau
    > CFGs \<and> CFGs'' \<in> trans_sf T2 \<tau> CFGs'}" |
"trans_sf (TApplyAll T) \<tau> CFGs = {CFGs'. apply_some (trans_sf T) \<tau> CFGs CFGs'} -
 {CFGs' | CFGs' CFGs''. CFGs' \<noteq> CFGs'' \<and> CFGs'' \<in> trans_sf T \<tau> CFGs'}"

end
```

```
end

(* trans_preds.thy *)
(* William Mansky *)
(* Generic state predicates for PTRANS side conditions. *)

theory trans_preds
imports trans_semantics
begin

datatype ('var) int_expr =
Num int |
Num_var 'var |
Num_expr_add "('var) int_expr" "('var) int_expr"

datatype ('mvar, 'edge_type_expr, 'instr, 'other) pred =
 Node 'mvar "'mvar node_lit" | Stmt 'mvar 'instr
 | Out 'mvar 'edge_type_expr "'mvar node_lit"
 | In 'mvar 'edge_type_expr "'mvar node_lit"
 | Freevar 'mvar 'instr | Is 'mvar 'mvar | Eq "'mvar int_expr" "'mvar int_expr"
 | Fresh 'mvar | OnlyIn 'mvar 'mvar
 | Other 'other

abbreviation "stmt t i \<equiv> SCPred (Stmt t (Inj i))"
abbreviation "node t i \<equiv> SCPred (Node t (MVar i))"
primrec SCExs where
"SCExs [] P = P" |
"SCExs (x # rest) P = SCEx x (SCExs rest P)"
lemma exs_fv [simp]: "cond_fv p (SCExs l P) = cond_fv p P - set l"
by (induct l, auto)

primrec SCAlls where
"SCAlls [] P = P" |
"SCAlls (x # rest) P = SCAll x (SCAlls rest P)"
lemma alls_fv [simp]: "cond_fv p (SCAlls l P) = cond_fv p P - set l"
by (induct l, auto)

(* Supporting lemmas for translating sequences of SCEx's to quantifiers over lists of
    objects. *)
lemma some_nth_distinct [simp]: "\<lbrakk>distinct l; i < length l\<rbrakk> \<
    Longrightarrow>
```

```
 (SOME ia. ia < length l \<and> l ! ia = l ! i) = i"
by (cut_tac P="\<lambda>ia. ia < length l \<and> l ! ia = l ! i" in someI, auto simp add:
    nth_eq_iff_index_eq)

lemma some_nth_distinct2 [simp]: "\<lbrakk>distinct l; i < length l; length l = k\<rbrakk>
    \<Longrightarrow>
 (SOME ia. ia < k \<and> l ! ia = l ! i) = i"
by (cut_tac P="\<lambda>ia. ia < length l \<and> l ! ia = l ! i" in someI, auto simp add:
    nth_eq_iff_index_eq)

lemma update_list_of [simp]: "\<lbrakk>distinct l; length objs \<ge> length l\<rbrakk> \<
    Longrightarrow> update_list \<sigma> (zip l objs) x =
 (if x \<in> set l then objs ! (SOME i. i < length l \<and> l ! i = x) else \<sigma> x)"
apply (induct l arbitrary: \<sigma> objs, simp_all)
apply (case_tac objs, auto)
apply (cut_tac P="\<lambda>i. i < Suc (length l) \<and> (x # l) ! i = x" in someI, auto)
apply (case_tac "SOME i. i < Suc (length l) \<and> (x # l) ! i = x", auto)
apply (clarsimp simp add: set_conv_nth)
apply (cut_tac l="a # l" and i="Suc i" in some_nth_distinct, auto simp add: set_conv_nth)
done

lemma update_list_of2 [simp]: "\<lbrakk>distinct l; length objs \<ge> length l; i < length
    l\<rbrakk> \<Longrightarrow>
 update_list \<sigma> (zip l objs) (l ! i) = objs ! i"
by simp

primrec int_expr_fv where
"int_expr_fv (Num i) = {}" |
"int_expr_fv (Num_var v) = {v}" |
"int_expr_fv (Num_expr_add ie ie2) = (int_expr_fv ie) \<union> (int_expr_fv ie2)"

lemma int_expr_fv_finite [simp]: "finite (int_expr_fv i)"
by (induct i, auto)

locale TRANS_preds = fixes subst::"(('thread, ('node, 'edge_type, 'instr) flowgraph) map)
    \<Rightarrow>
                                   ('mvar \<Rightarrow> 'object) \<Rightarrow> ('pattern, '
                                       mvar) literal \<rightharpoonup> 'instr"
  and node_subst::"(('thread, ('node, 'edge_type, 'instr) flowgraph) map) \<Rightarrow> ('
      mvar \<Rightarrow> 'object) \<Rightarrow>
```

```
                  'mvar node_lit \<rightharpoonup> 'node"
  and type_subst::"('mvar \<Rightarrow> 'object) \<Rightarrow> 'edge_type_expr \<
      rightharpoonup> 'edge_type"
  and thread_subst::"('mvar \<Rightarrow> 'object) \<Rightarrow> 'mvar \<rightharpoonup> '
      thread"
  and var_subst::"('mvar \<Rightarrow> 'object) \<Rightarrow> 'mvar \<rightharpoonup> 'var"
  and eval_other::"(('thread, ('node, 'edge_type, 'instr) flowgraph) map) \<Rightarrow> ('
      thread \<rightharpoonup> 'node) \<Rightarrow>
    ('mvar \<Rightarrow> 'object) \<Rightarrow> 'other \<Rightarrow> bool"
  and type_fv::"'edge_type_expr \<Rightarrow> 'mvar set" and pat_fv::"'pattern \<Rightarrow
      > 'mvar set"
  and other_fv::"'other \<Rightarrow> 'mvar set" and instr_fv::"'instr \<Rightarrow> 'var
      set"
  and int_of::"('mvar \<Rightarrow> 'object) \<Rightarrow> 'mvar \<rightharpoonup> int"
  and instr_edges::"'instr \<Rightarrow> ('edge_type \<Rightarrow> nat) set" and Seq::"'
      edge_type"
  assumes same_subst: "\<lbrakk>\<forall>x\<in>lit_fv pat_fv p. \<sigma> x = \<sigma>' x;
      tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
    subst CFGs \<sigma> p = subst CFGs \<sigma>' p"
      and node_same_subst: "\<lbrakk>\<forall>x\<in>node_fv n. \<sigma> x = \<sigma>' x;
          tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
    node_subst CFGs \<sigma> n = node_subst CFGs \<sigma>' n"
      and type_same_subst: "\<forall>x\<in>type_fv ty. \<sigma> x = \<sigma>' x \<
          Longrightarrow> type_subst \<sigma> ty = type_subst \<sigma>' ty"
      and thread_same_subst: "\<sigma> t = \<sigma>' t \<Longrightarrow> thread_subst \<
          sigma> t = thread_subst \<sigma>' t"
      and var_same_subst: "\<sigma> v = \<sigma>' v \<Longrightarrow> var_subst \<sigma> v
          = var_subst \<sigma>' v"
      and int_of_same: "\<sigma> i = \<sigma>' i \<Longrightarrow> int_of \<sigma> i =
          int_of \<sigma>' i"
      and other_same_subst: "\<forall>x\<in>other_fv r. \<sigma> x = \<sigma>' x \<
          Longrightarrow>
        eval_other CFGs q \<sigma> r = eval_other CFGs q \<sigma>' r"
      and type_fv_finite [simp]: "finite (type_fv ty)"
      and pat_fv_finite [simp]: "finite (pat_fv pt)"
      and other_fv_finite [simp]: "finite (other_fv r)"
      and infinite_nodes [simp]: "\<not>finite (UNIV::'node set)"
      and infinite_mvars [simp]: "\<not>finite (UNIV::'mvar set)"
begin
```

```
lemma pat_lit_fv_finite [simp]: "finite (lit_fv pat_fv p)"
by (case_tac p, simp+)

primrec pred_fv where
"pred_fv (Node t n) = insert t (node_fv n)" |
"pred_fv (Out t ty n) = insert t (type_fv ty \<union> node_fv n)" |
"pred_fv (In t ty n) = insert t (type_fv ty \<union> node_fv n)" |
"pred_fv (Stmt t i) = insert t (lit_fv pat_fv i)" |
"pred_fv (Freevar x i) = insert x (lit_fv pat_fv i)" |
"pred_fv (Is x1 x2) = {x1, x2}" |
"pred_fv (Eq e1 e2) = int_expr_fv e1 \<union> int_expr_fv e2" |
"pred_fv (Fresh x) = {x}" |
"pred_fv (OnlyIn t x) = {t, x}" |
"pred_fv (Other x) = other_fv x"

lemma pred_fv_finite [simp]: "finite (pred_fv p)"
by (case_tac p, auto)

lemma cond_fv_finite [simp]: "finite (cond_fv pred_fv P)"
by (induct P, auto)

primrec eval_int where
"eval_int \<sigma> (Num i) = Some i" |
"eval_int \<sigma> (Num_var v) = int_of \<sigma> v" |
"eval_int \<sigma> (Num_expr_add e1 e2) = (case (eval_int \<sigma> e1, eval_int \<sigma> e2
    ) of
  (Some i1, Some i2) \<Rightarrow> Some (i1 + i2))"

lemma int_expr_same_subst: "\<forall>x\<in>int_expr_fv e. \<sigma> x = \<sigma>' x \<
    Longrightarrow> eval_int \<sigma> e = eval_int \<sigma>' e"
by (induct e, auto intro: int_of_same)

primrec eval_pred where
"eval_pred CFGs q \<sigma> (Node t n) = (case (thread_subst \<sigma> t, node_subst CFGs \<
    sigma> n) of (Some t', Some n') \<Rightarrow>
 q t' = Some n' | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (Out t ty n) = (case (thread_subst \<sigma> t, type_subst \<
    sigma> ty, node_subst CFGs \<sigma> n) of
 (Some t', Some ty', Some n') \<Rightarrow> (case q t' of Some n0 \<Rightarrow> (n0, n', ty
     ') \<in> Edges (the (CFGs t'))
```

```
 | _ \<Rightarrow> False) | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (In t ty n) = (case (thread_subst \<sigma> t, type_subst \<sigma
    > ty, node_subst CFGs \<sigma> n) of
 (Some t', Some ty', Some n') \<Rightarrow> (case q t' of Some n0 \<Rightarrow> (n', n0, ty
     ') \<in> Edges (the (CFGs t'))
 | _ \<Rightarrow> False) | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (Stmt t i) = (case (thread_subst \<sigma> t, subst CFGs \<sigma>
     i) of (Some t', Some i') \<Rightarrow>
 \<exists>n G. q t' = Some n \<and> CFGs t' = Some G \<and> n \<noteq> Exit G \<and> Label
     G n = i' | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (Freevar x i) = (case (var_subst \<sigma> x, subst CFGs \<sigma>
     i) of (Some x', Some i') \<Rightarrow>
 x' \<in> instr_fv i' | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (Is x1 x2) = (\<sigma> x1 = \<sigma> x2)" |
"eval_pred CFGs q \<sigma> (Eq e1 e2) = (case (eval_int \<sigma> e1, eval_int \<sigma> e2)
    of
 (Some i1, Some i2) \<Rightarrow> i1 = i2 | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (Fresh x) = (case var_subst \<sigma> x of Some x' \<Rightarrow>
 \<forall>n \<in> nodes CFGs. x' \<notin> instr_fv (label n CFGs) | _ \<Rightarrow> False)"
      |
"eval_pred CFGs q \<sigma> (OnlyIn t x) = (case (thread_subst \<sigma> t, var_subst \<sigma
    > x) of (Some t', Some x') \<Rightarrow>
 \<forall>n \<in> nodes CFGs. x' \<in> instr_fv (label n CFGs) \<longrightarrow> thread_of
     n CFGs = t' | _ \<Rightarrow> False)" |
"eval_pred CFGs q \<sigma> (Other x) = eval_other CFGs q \<sigma> x"

lemma pred_same_subst: "\<lbrakk>\<forall>x\<in>pred_fv p. \<sigma> x = \<sigma>' x; tCFG
    CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
 eval_pred CFGs q \<sigma> p = eval_pred CFGs q \<sigma>' p"
apply (case_tac p, simp_all)
apply (clarsimp, drule thread_same_subst, drule node_same_subst, simp, simp split: option.
    splits)
apply (clarsimp, drule thread_same_subst, drule same_subst, simp, simp split: option.splits
    )
apply (clarsimp, drule thread_same_subst)
apply (cut_tac ty=edge_type_expr and \<sigma>=\<sigma> and \<sigma>'=\<sigma>' in
    type_same_subst, simp)
apply (cut_tac n=literal and \<sigma>=\<sigma> and \<sigma>'=\<sigma>' in node_same_subst,
    simp+, simp split: option.splits)
apply (clarsimp, drule thread_same_subst)
```

```
apply (cut_tac ty=edge_type_expr and \<sigma>=\<sigma> and \<sigma>'=\<sigma>' in
    type_same_subst, simp)
apply (cut_tac n=literal and \<sigma>=\<sigma> and \<sigma>'=\<sigma>' in node_same_subst,
    simp+, simp split: option.splits)
apply (clarsimp, drule var_same_subst, drule same_subst, simp, simp split: option.splits)
apply (cut_tac e=int_expr1 and \<sigma>=\<sigma> and \<sigma>'=\<sigma>' in
    int_expr_same_subst, simp)
apply (cut_tac e=int_expr2 and \<sigma>=\<sigma> and \<sigma>'=\<sigma>' in
    int_expr_same_subst, simp+, simp split: option.splits)
apply (clarsimp, drule var_same_subst, simp split: option.splits)
apply (clarsimp, drule thread_same_subst, drule var_same_subst, simp split: option.splits)
apply (erule other_same_subst)
done

end

sublocale TRANS_preds \<subseteq> TRANS_basics where pred_fv=pred_fv and eval_pred=
    eval_pred
apply (unfold_locales, simp+)
apply (rule pred_same_subst, simp+)
done

context TRANS_preds begin

lemma side_cond_gen: "\<lbrakk>models CFGs \<sigma> q P; \<forall>x\<in>cond_fv pred_fv P.
    \<sigma> x = \<sigma>' x;
  tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow> models CFGs \<sigma>' q P"
by (erule cond_gen, auto)

(* The algebra of single-thread predicates. *)
definition "by_thread P t \<equiv> \<forall>CFGs \<sigma> q q' l l'. (case thread_subst \<
    sigma> t of Some t' \<Rightarrow> q t' = q' t' | _ \<Rightarrow> True) \<and>
 l\<in>tCFG.RPaths CFGs q \<and> l'\<in>tCFG.RPaths CFGs q' \<and> tCFG CFGs instr_edges
     Seq \<and>
 models CFGs \<sigma> q P \<longrightarrow> models CFGs \<sigma> q' P"

lemma by_threadI [intro]: "(\<And>CFGs \<sigma> q q'. (case thread_subst \<sigma> t of Some
     t' \<Rightarrow> q t' = q' t' | _ \<Rightarrow> True) \<and>
 tCFG CFGs instr_edges Seq \<and> models CFGs \<sigma> q P \<Longrightarrow> models CFGs \<
     sigma> q' P) \<Longrightarrow> by_thread P t"
```

```
apply (clarsimp simp add: by_thread_def)
by smt

lemma by_threadI2 [intro]: "(\<And>CFGs \<sigma> q q' l l'. (case thread_subst \<sigma> t
    of Some t' \<Rightarrow> q t' = q' t' | _ \<Rightarrow> True) \<and>
 l\<in>tCFG.RPaths CFGs q \<and> l'\<in>tCFG.RPaths CFGs q' \<and> tCFG CFGs instr_edges
     Seq \<and> models CFGs \<sigma> q P \<Longrightarrow>
 models CFGs \<sigma> q' P) \<Longrightarrow> by_thread P t"
by (force simp add: by_thread_def)

lemma by_threadD [dest]: "\<lbrakk>by_thread P t; thread_subst \<sigma> t = Some t'; q t' =
     q' t';
  l\<in>tCFG.RPaths CFGs q; l'\<in>tCFG.RPaths CFGs q'; tCFG CFGs instr_edges Seq; models
      CFGs \<sigma> q P\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' P"
apply (simp add: by_thread_def)
apply (erule_tac x=CFGs in allE, erule_tac x=\<sigma> in allE, erule_tac x=q in allE, force
    )
done

lemma dom_end [simp]: "dom (\<lambda>t. if t \<in> dom q then Some (Start (the (CFGs t)))
    else None) = dom q"
by (rule set_eqI, simp add: dom_def)

lemma start_points_dom [simp]: "(\<lambda>t. if t \<in> dom CFGs then Some (Start (the (
    CFGs t))) else None) =
 start_points CFGs"
by (auto intro!: ext simp add: start_points_def dom_def)

lemma by_thread_notD [dest]: "\<lbrakk>by_thread P t; thread_subst \<sigma> t = None; l\<in
    >tCFG.RPaths CFGs q;
 l'\<in>tCFG.RPaths CFGs q'; tCFG CFGs instr_edges Seq; models CFGs \<sigma> q P\<rbrakk>
     \<Longrightarrow>
 models CFGs \<sigma> q' P"
by (simp add: by_thread_def, erule_tac x=CFGs in allE,
 erule_tac x=\<sigma> in allE, erule_tac x=q in allE, erule_tac x=q' in allE, auto)

lemma by_thread_conj [intro!]: "by_thread P t \<and> by_thread Q t \<Longrightarrow>
    by_thread (P \<and>sc Q) t"
apply (clarsimp intro!: by_threadI2)
```

```
apply (case_tac "thread_subst \<sigma> t", clarsimp)
apply (drule_tac \<sigma>=\<sigma> and q=q and q'=q' in by_thread_notD, simp+)
apply (erule by_thread_notD, simp+)
apply (drule_tac \<sigma>=\<sigma> and q=q in by_threadD, simp+)
apply (drule_tac \<sigma>=\<sigma> and q=q in by_threadD, simp+)
done

lemma by_thread_not [intro!]: "by_thread P t \<Longrightarrow> by_thread (\<not>sc P) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "thread_subst \<sigma> t", clarsimp)
apply (drule_tac \<sigma>=\<sigma> and q=q' and q'=q in by_thread_notD, simp+)
apply (drule_tac \<sigma>=\<sigma> and q=q' and q'=q in by_threadD, simp+)
done

lemma by_thread_ex [intro!]: "\<lbrakk>by_thread P t; x \<noteq> t\<rbrakk> \<
    Longrightarrow> by_thread (SCEx x P) t"
apply (rule by_threadI2, clarsimp)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'="\<sigma>(x := obj)" and t=t in
    thread_same_subst, simp)
apply (case_tac "thread_subst (\<sigma>(x := obj)) t", simp_all)
apply (drule_tac \<sigma>="\<sigma>(x := obj)" and q=q and q'=q' in by_thread_notD, simp+,
    metis)
apply (drule_tac \<sigma>="\<sigma>(x := obj)" and q=q and q'=q' in by_threadD, simp+,
    metis)
done

lemma by_thread_exs [intro!]: "\<lbrakk>by_thread P t; t \<notin> set xs\<rbrakk> \<
    Longrightarrow> by_thread (SCExs xs P) t"
by (induct xs, auto)

lemma by_thread_EU [intro!]: "\<lbrakk>by_thread P t; by_thread Q t\<rbrakk> \<
    Longrightarrow> by_thread (E P \<U> Q) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "thread_subst \<sigma> t", clarsimp)
apply (frule_tac q=q' in tCFG.exists_path, clarsimp, rule bexI, simp_all)
apply (rule_tac x=0 in exI, clarsimp)
apply (frule_tac i=i in tCFG.path_rpath, simp+, clarsimp)
apply (erule_tac by_thread_notD, simp_all, simp add: tCFG.Path_first)
apply (cut_tac q=q and q'=q' and t=a in tCFG.path_by_thread, simp+, clarsimp)
apply (rule_tac x=l'a in bexI, simp_all)
```

```
apply (rule_tac x=i in exI, rule conjI)
apply (cut_tac i=i in tCFG.path_rpath, simp+, clarsimp)
apply (cut_tac l=l'a and i=i in tCFG.path_rpath, simp+, clarsimp)
apply (drule_tac P=Q and q="la i" and t'=a in by_threadD, simp_all)
apply clarsimp
apply (cut_tac i=j in tCFG.path_rpath, simp+, clarsimp)
apply (frule_tac l=l'a and i=j in tCFG.path_rpath, simp+, clarsimp)
apply (drule_tac P=P and q="la j" and t'=a in by_threadD, simp_all)
done

lemma by_thread_EB [intro!]: "\<lbrakk>by_thread P t; by_thread Q t\<rbrakk> \<
    Longrightarrow> by_thread (E P \<B> Q) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "thread_subst \<sigma> t", clarsimp)
apply (rule bexI, simp_all, rule_tac x=0 in exI, simp)
apply (erule_tac by_thread_notD, simp_all)
apply (erule tCFG.rpath_suffix, simp+)+
apply (frule_tac l=la and q=q and q'=q' and t=a in tCFG.rpath_by_thread, simp+, clarsimp)
apply (rule_tac x=l'a in bexI, simp_all)
apply (rule_tac x=i in exI, rule conjI, drule_tac P=Q and t'=a in by_threadD, simp_all)
apply (erule tCFG.rpath_suffix, simp+)+
apply (clarsimp, erule_tac x=j in allE, simp)
apply (drule_tac P=P and t'=a in by_threadD, simp_all)
apply (erule tCFG.rpath_suffix, simp+)+
done

lemma by_thread_AU [intro!]: "\<lbrakk>by_thread P t; by_thread Q t\<rbrakk> \<
    Longrightarrow> by_thread (A P \<U> Q) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "thread_subst \<sigma> t", clarsimp)
apply (frule_tac q=q in tCFG.exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (rule_tac x=0 in exI, clarsimp)
apply (frule_tac l=lb and i=i in tCFG.path_rpath, simp+, clarsimp)
apply (erule_tac by_thread_notD, simp_all, simp add: tCFG.Path_first)
apply (frule_tac q=q' and q'=q and t=a in tCFG.path_by_thread, simp+, clarsimp)
apply (erule_tac x=l'a in ballE, simp_all, clarsimp)
apply (rule_tac x=i in exI, rule conjI)
apply (frule_tac i=i in tCFG.path_rpath, simp+, clarsimp)
apply (frule_tac l=l'a and i=i in tCFG.path_rpath, simp+, clarsimp)
```

```
apply (erule_tac P=Q and t'=a in by_threadD, simp_all)
apply clarsimp
apply (frule_tac i=j in tCFG.path_rpath, simp+, clarsimp)
apply (frule_tac l=l'a and i=j in tCFG.path_rpath, simp+, clarsimp)
apply (erule_tac P=P and t'=a and q="l'a j" in by_threadD, simp_all)
done

lemma by_thread_AB [intro!]: "\<lbrakk>by_thread P t; by_thread Q t\<rbrakk> \<
    Longrightarrow> by_thread (A P \<B> Q) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "thread_subst \<sigma> t", clarsimp)
apply (erule_tac x=l in ballE, simp_all, rule_tac x=0 in exI, clarsimp)
apply (erule_tac by_thread_notD, simp_all)
apply (erule tCFG.rpath_suffix, simp+)+
apply (frule_tac l=la and q=q' and q'=q and t=a in tCFG.rpath_by_thread, simp+, clarsimp)
apply (erule_tac x=l'a in ballE, simp_all, clarsimp)
apply (rule_tac x=i in exI, rule conjI, drule_tac P=Q and t'=a in by_threadD, simp_all)
apply (erule tCFG.rpath_suffix, simp+)+
apply (clarsimp, erule_tac x=j in allE, simp)
apply (drule_tac P=P and t'=a in by_threadD, simp_all)
apply (erule tCFG.rpath_suffix, simp+)+
done

lemma by_thread_node [intro!]: "by_thread (node t n) t"
by (rule by_threadI, clarsimp split: option.splits)

lemma by_thread_stmt [intro!]: "by_thread (stmt t i) t"
by (rule by_threadI, clarsimp split: option.splits, force)

lemma by_thread_out [intro!]: "by_thread (SCPred (Out t ty n)) t"
by (rule by_threadI, auto split: option.splits)

lemma by_thread_in [intro!]: "by_thread (SCPred (In t ty n)) t"
by (rule by_threadI, auto split: option.splits)

lemma by_thread_path: "\<lbrakk>by_thread P t; thread_subst \<sigma> t = Some t'; l i t' =
    Some n;
  l\<in>tCFG.Paths CFGs (start_points CFGs); tCFG CFGs instr_edges Seq\<rbrakk> \<
      Longrightarrow>
  models CFGs \<sigma> (l i) P = models CFGs \<sigma> [t' \<mapsto> n] P"
```

```
apply (frule_tac i=i in tCFG.reverse_path, simp, clarsimp)
apply (frule_tac tCFG.rpath_one_thread, simp+)
apply (simp add: by_thread_def, rule iffI)
apply (erule_tac x=CFGs in allE, erule_tac x=\<sigma> in allE, erule_tac x="l i" in allE,
  erule_tac x="[t' \<mapsto> n]" in allE, force)
apply (erule_tac x=CFGs in allE, erule_tac x=\<sigma> in allE, erule_tac x="[t' \<mapsto> n
    ]" in allE,
  erule_tac x="l i" in allE, force)
done

corollary by_thread_pathD1: "\<lbrakk>by_thread P t; thread_subst \<sigma> t = Some t'; l i
     t' = Some n;
  l\<in>tCFG.Paths CFGs (start_points CFGs); tCFG CFGs instr_edges Seq\<rbrakk> \<
      Longrightarrow>
  models CFGs \<sigma> (l i) P \<Longrightarrow> models CFGs \<sigma> [t' \<mapsto> n] P"
by (simp add: by_thread_path)

corollary by_thread_pathD2: "\<lbrakk>by_thread P t; thread_subst \<sigma> t = Some t'; l i
     t' = Some n;
  l\<in>tCFG.Paths CFGs (start_points CFGs); tCFG CFGs instr_edges Seq\<rbrakk> \<
      Longrightarrow>
  models CFGs \<sigma> [t' \<mapsto> n] P \<Longrightarrow> models CFGs \<sigma> (l i) P"
by (simp add: by_thread_path)

end

end

(* trans_sim.thy *)
(* William Mansky *)
(* The basics of simulation relations on tCFGs. *)

theory trans_sim
imports trans_semantics "~~/src/AFP/JinjaThreads/Framework/Bisimulation"
begin

definition trsys_of_tCFG where
"trsys_of_tCFG CFGs step_rel obs get_mem C l C' \<equiv> step_rel CFGs C C' \<and> l =
    get_mem C' |` obs"
(* This needs to reflect the actual changes made! *)
abbreviation "trsys_of_tCFG_full CFGs step_rel \<equiv> trsys_of_tCFG CFGs step_rel UNIV"
```

```
lemma dom_UNIV [simp]: "dom l = UNIV \<Longrightarrow> m ++ l = l"
by (rule ext, case_tac "l x", auto)

(* One-directional simulation, modified from the bisimulation theory of JinjaThreads *)
locale simulation = bisimulation_base +
  constrains trsys1 :: "('s1, 'tl1) trsys"
  and trsys2 :: "('s2, 'tl2) trsys"
  and bisim :: "('s1, 's2) bisim"
  and tlsim :: "('tl1, 'tl2) bisim"
assumes simulation: "\<lbrakk>sa \<approx> sb; sa -1-tl1\<rightarrow> sa'\<rbrakk> \<
    Longrightarrow> \<exists>sb' tl2. sb -2-tl2\<rightarrow> sb' \<and> sa' \<approx> sb' \<
    and> tl1 \<sim> tl2"
begin

lemma simulation_rtrancl:
  "[|s1 -1-tls1\<rightarrow>* s1'; s1 \<approx> s2|]
  ==> \<exists>s2' tls2. s2 -2-tls2\<rightarrow>* s2' \<and> s1' \<approx> s2' \<and> tls1
      [\<sim>] tls2"
proof(induct rule: rtrancl3p.induct)
  case rtrancl3p_refl thus ?case by(auto intro: rtrancl3p.rtrancl3p_refl)
next
  case (rtrancl3p_step s1 tls1 s1' tl1 s1'')
  from `s1 \<approx> s2 ==> \<exists>s2' tls2. s2 -2-tls2\<rightarrow>* s2' \<and> s1' \<
      approx> s2' \<and> tls1 [\<sim>] tls2` `s1 \<approx> s2`
  obtain s2' tls2 where "s2 -2-tls2\<rightarrow>* s2'" "s1' \<approx> s2'" "tls1 [\<sim>]
      tls2" by blast
  moreover from `s1' -1-tl1\<rightarrow> s1''` `s1' \<approx> s2'`
  obtain s2'' tl2 where "s2' -2-tl2\<rightarrow> s2''" "s1'' \<approx> s2''" "tl1 \<sim>
      tl2" by(auto dest: simulation)
  ultimately have "s2 -2-tls2 @ [tl2]\<rightarrow>* s2''" "tls1 @ [tl1] [\<sim>] tls2 @ [
      tl2]"
    by(auto intro: rtrancl3p.rtrancl3p_step list_all2_appendI)
  with `s1'' \<approx> s2''` show ?case by(blast)
qed

lemma simulation_inf_step:
  assumes red1: "s1 -1-tls1\<rightarrow>* \<infinity>" and bisim: "s1 \<approx> s2"
  shows "\<exists>tls2. s2 -2-tls2\<rightarrow>* \<infinity> \<and> tls1 [[\<sim>]] tls2"
proof -
  from r1.inf_step_imp_inf_step_table[OF red1]
```

```
obtain stls1 where red1': "s1 -1-stls1\<rightarrow>*t \<infinity>"
  and tls1: "tls1 = lmap (fst \<circ> snd) stls1" by blast
def tl1_to_tl2_def: tl1_to_tl2 \<equiv> "\<lambda>(s2 :: 's2) (stls1 :: ('s1 \<times> '
    tl1 \<times> 's1) llist). llist_unfold
   (\<lambda>(s2, stls1). lnull stls1)
   (\<lambda>(s2, stls1). let (s1, tl1, s1') = lhd stls1;
                       (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' \<and> s1' \<approx
                           > s2' \<and> tl1 \<sim> tl2
                   in (s2, tl2, s2'))
   (\<lambda>(s2, stls1). let (s1, tl1, s1') = lhd stls1;
                       (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' \<and> s1' \<approx
                           > s2' \<and> tl1 \<sim> tl2
                   in (s2', ltl stls1))
   (s2, stls1)"

have tl1_to_tl2_simps [simp]:
  "\<And>s2 stls1. lnull (tl1_to_tl2 s2 stls1) \<longleftrightarrow> lnull stls1"
  "\<And>s2 stls1. \<not> lnull stls1 \<Longrightarrow> lhd (tl1_to_tl2 s2 stls1) =
  (let (s1, tl1, s1') = lhd stls1;
       (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' \<and> s1' \<approx> s2' \<and>
           tl1 \<sim> tl2
   in (s2, tl2, s2'))"
  "\<And>s2 stls1. \<not> lnull stls1 \<Longrightarrow> ltl (tl1_to_tl2 s2 stls1) =
  (let (s1, tl1, s1') = lhd stls1;
       (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' \<and> s1' \<approx> s2' \<and>
           tl1 \<sim> tl2
   in tl1_to_tl2 s2' (ltl stls1))"
  "\<And>s2. tl1_to_tl2 s2 LNil = LNil"
  "\<And>s2 s1 tl1 s1' stls1'. tl1_to_tl2 s2 (LCons (s1, tl1, s1') stls1') =
      LCons (s2, SOME (tl2, s2'). trsys2 s2 tl2 s2' \<and> s1' \<approx> s2' \<and> tl1
          \<sim> tl2)
            (tl1_to_tl2 (snd (SOME (tl2, s2'). trsys2 s2 tl2 s2' \<and> s1' \<approx> s2'
                \<and> tl1 \<sim> tl2)) stls1')"
  by(simp_all add: tl1_to_tl2_def split_beta)

have [simp]: "llength (tl1_to_tl2 s2 stls1) = llength stls1"
  apply (coinduction arbitrary: s2 stls1 rule: enat_coinduct) apply (auto simp add:
      epred_llength split_beta llength_def)
  apply (metis enat_unfold_eq_0 tl1_to_tl2_simps(1) zero_enat_def)
  apply (metis enat_unfold_eq_0 zero_enat_def)
```

```
done

from red1' bisim have "s2 -2-tl1_to_tl2 s2 stls1\<rightarrow>*t \<infinity>"
proof(coinduction arbitrary: s2 s1 stls1)
  case (inf_step_table s2 s1 stls1)
  note red1' = `s1 -1-stls1\<rightarrow>*t \<infinity>` and bisim = `s1 \<approx> s2`
  from red1' show ?case
  proof(cases)
    case (inf_step_tableI s1' stls1' tl1)
    hence stls1: "stls1 = LCons (s1, tl1, s1') stls1'"
      and r: "s1 -1-tl1\<rightarrow> s1'" and reds1: "s1' -1-stls1'\<rightarrow>*t \<
          infinity>" by simp_all
    let ?tl2s2' = "SOME (tl2, s2'). s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2'
        \<and> tl1 \<sim> tl2"
    let ?tl2 = "fst ?tl2s2'" let ?s2' = "snd ?tl2s2'"
    from simulation[OF bisim r] obtain s2' tl2
      where "s2 -2-tl2\<rightarrow> s2'" "s1' \<approx> s2'" "tl1 \<sim> tl2" by blast
    hence "(\<lambda>(tl2, s2'). s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2' \<
        and> tl1 \<sim> tl2) (tl2, s2')" by simp
    hence "(\<lambda>(tl2, s2'). s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2' \<
        and> tl1 \<sim> tl2) ?tl2s2'" by(rule someI)
    hence "s2 -2-?tl2\<rightarrow> ?s2'" "s1' \<approx> ?s2'" "tl1 \<sim> ?tl2" by(
        simp_all add: split_beta)
    then show ?thesis using reds1 stls1 by(fastforce intro: prod_eqI)
  qed
qed
hence "s2 -2-lmap (fst \<circ> snd) (tl1_to_tl2 s2 stls1)\<rightarrow>* \<infinity>"
  by(rule r2.inf_step_table_imp_inf_step)
moreover have "tls1 [[\<sim>]] lmap (fst \<circ> snd) (tl1_to_tl2 s2 stls1)"
proof(rule llist_all2_all_lnthI)
  show "llength tls1 = llength (lmap (fst \<circ> snd) (tl1_to_tl2 s2 stls1))"
    using tls1 by simp
next
  fix n
  assume "enat n < llength tls1"
  thus "lnth tls1 n \<sim> lnth (lmap (fst \<circ> snd) (tl1_to_tl2 s2 stls1)) n"
    using red1' bisim unfolding tls1
  proof(induct n arbitrary: s1 s2 stls1 rule: nat_less_induct)
    case (1 n)
```

```
hence IH: "\<And>m s1 s2 stls1. \<lbrakk> m < n; enat m < llength (lmap (fst \<circ>
    snd) stls1);
                             s1 -1-stls1\<rightarrow>*t \<infinity>; s1 \<approx> s2
                                 \<rbrakk>
           \<Longrightarrow> lnth (lmap (fst \<circ> snd) stls1) m \<sim> lnth (lmap
               (fst \<circ> snd) (tl1_to_tl2 s2 stls1)) m"
  by blast
from `s1 -1-stls1\<rightarrow>*t \<infinity>` show ?case
proof cases
  case (inf_step_tableI s1' stls1' tl1)
  hence  stls1: "stls1 = LCons (s1, tl1, s1') stls1'"
    and r: "s1 -1-tl1\<rightarrow> s1'" and reds: "s1' -1-stls1'\<rightarrow>*t \<
        infinity>" by simp_all
  let ?tl2s2' = "SOME (tl2, s2').  s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2
      ' \<and> tl1 \<sim> tl2"
  let ?tl2 = "fst ?tl2s2'" let ?s2' = "snd ?tl2s2'"
  from simulation[OF `s1 \<approx> s2` r] obtain s2' tl2
    where "s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2' \<and> tl1 \<sim> tl2"
         by blast
  hence "(\<lambda>(tl2, s2'). s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2' \<
      and> tl1 \<sim> tl2) (tl2, s2')" by simp
  hence "(\<lambda>(tl2, s2'). s2 -2-tl2\<rightarrow> s2' \<and> s1' \<approx> s2' \<
      and> tl1 \<sim> tl2) ?tl2s2'" by(rule someI)
  hence bisim': "s1' \<approx> ?s2'" and tlsim: "tl1 \<sim> ?tl2" by(simp_all add:
      split_beta)
  show ?thesis
  proof(cases n)
    case 0
    with stls1 tlsim show ?thesis by simp
  next
    case (Suc m)
    hence "m < n" by simp
    moreover have "enat m < llength (lmap (fst \<circ> snd) stls1')"
      using stls1 `enat n < llength (lmap (fst \<circ> snd) stls1)` Suc by(simp add:
          Suc_ile_eq)
    ultimately have "lnth (lmap (fst \<circ> snd) stls1') m \<sim> lnth (lmap (fst \<
        circ> snd) (tl1_to_tl2 ?s2' stls1')) m"
      using reds bisim' by(rule IH)
    with Suc stls1 show ?thesis by(simp del: o_apply)
  qed
```

```
      qed
    qed
  qed
  ultimately show ?thesis by blast
qed

end

sublocale bisimulation \<subseteq> simulation
by (unfold_locales, simp add: simulation1)

locale tCFG_bisim = bisimulation where trsys1 = "trsys_of_tCFG CFGs step_rel obs get_mem"
 and trsys2 = "trsys_of_tCFG CFGs' step_rel obs get_mem" for CFGs CFGs' step_rel obs
     get_mem

locale tCFG_sim = simulation where trsys1 = "trsys_of_tCFG CFGs step_rel obs get_mem"
 and trsys2 = "trsys_of_tCFG CFGs' step_rel obs get_mem" for CFGs CFGs' step_rel obs
     get_mem

definition (in TRANS_basics) "opt_sim T sim tlsim step_rel obs get_mem \<equiv>
  \<forall>\<tau> CFGs. tCFG CFGs instr_edges Seq \<longrightarrow> (\<forall>CFGs' \<in>
      trans_sf T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
    tCFG_sim sim tlsim CFGs' CFGs step_rel obs get_mem)"

(* composition *)
lemma sim_comp: "\<lbrakk>tCFG_sim sim tlsim CFGs CFGs' step_rel obs get_mem;
  tCFG_sim sim' tlsim' CFGs' CFGs'' step_rel obs get_mem\<rbrakk> \<Longrightarrow>
tCFG_sim (sim \<circ>\<^sub>B sim') (tlsim \<circ>\<^sub>B tlsim') CFGs CFGs'' step_rel obs
     get_mem"
apply (clarsimp simp add: tCFG_sim_def simulation_def bisim_compose_def trsys_of_tCFG_def)
by (metis (hide_lams, mono_tags))

context TRANS_basics begin

lemma then_sim [intro]: "\<lbrakk>opt_sim T1 sim1 tlsim1 step_rel obs get_mem; opt_sim T2
    sim2 tlsim2 step_rel obs get_mem\<rbrakk> \<Longrightarrow>
  opt_sim (TThen T1 T2) (sim2 \<circ>\<^sub>B sim1) (tlsim2 \<circ>\<^sub>B tlsim1)
      step_rel obs get_mem"
unfolding opt_sim_def
apply clarify
```

```
apply (erule_tac x=\<tau> in allE, erule_tac x=CFGs in allE, erule impE, simp, clarsimp)
apply (erule_tac x=CFGs'a in ballE, simp_all)
apply (erule_tac x=\<tau> in allE, erule_tac x=CFGs'a in allE, erule impE, simp,
  erule_tac x=CFGs' in ballE, simp_all, clarsimp)
apply (rule sim_comp, simp_all)
done

lemma applysome_sim: "\<lbrakk>opt_sim T sim tlsim step_rel obs get_mem; apply_some (
    trans_sf T) \<tau> CFGs CFGs';
  tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
  tCFG CFGs' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_sim sim' tlsim' CFGs' CFGs
      step_rel obs get_mem)"
apply (clarsimp simp add: opt_sim_def, drule_tac P="\<lambda>t \<tau> G G'. t = trans_sf T
    \<and> tCFG G instr_edges Seq \<longrightarrow>
  (tCFG G' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_sim sim' tlsim' G' G step_rel
       obs get_mem))"
  in apply_some.induct, auto)
apply (rule_tac x="op =" in exI, rule_tac x="op =" in exI, unfold_locales, force)
apply (erule_tac x=\<tau> in allE, erule_tac x=G in allE, simp, erule_tac x=G' in ballE,
    simp_all, clarsimp)
apply (rule exI, rule exI, rule sim_comp, simp+)
done

(* is this sufficient? *)
definition "trans_sim T step_rel obs get_mem \<equiv>
  \<forall>\<tau> CFGs. tCFG CFGs instr_edges Seq \<longrightarrow> (\<forall>CFGs' \<in>
      trans_sf T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
    (\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs step_rel obs get_mem))"

lemma opt_sim_trans [intro]: "opt_sim T sim tlsim step_rel obs get_mem \<Longrightarrow>
    trans_sim T step_rel obs get_mem"
by (force simp add: opt_sim_def trans_sim_def)

definition "cond_sim P T step_rel obs get_mem \<equiv> \<forall>\<tau> CFGs. P \<tau> CFGs
    \<and> tCFG CFGs instr_edges Seq \<longrightarrow>
  (\<forall>CFGs' \<in> trans_sf T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
    (\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs step_rel obs get_mem))"

lemma cond_simI [intro]: "(\<And>\<tau> CFGs CFGs'. \<lbrakk>P \<tau> CFGs; tCFG CFGs
    instr_edges Seq; CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow>
```

```
  tCFG CFGs' instr_edges Seq \<and> (\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs
      step_rel obs get_mem))
  \<Longrightarrow> cond_sim P T step_rel obs get_mem"
by (force simp add: cond_sim_def)

lemma applyall_sim [intro]: "opt_sim T sim tlsim step_rel obs get_mem \<Longrightarrow>
    trans_sim (TApplyAll T) step_rel obs get_mem"
apply (clarsimp simp add: trans_sim_def)
by (metis applysome_sim)

lemma match_cond_sim [intro]: "cond_sim (\<lambda>\<tau> CFGs. \<exists>\<sigma> \<tau>'.
    side_cond_sf \<phi> \<sigma> CFGs \<and> part_matches \<sigma> \<tau>' \<and>
  \<tau> = part_extend \<tau>' (cond_fv pred_fv \<phi>) \<sigma>) T step_rel obs get_mem \<
      Longrightarrow>
  trans_sim (TMatch \<phi> T) step_rel obs get_mem"
apply (clarsimp simp add: trans_sim_def cond_sim_def)
by metis

lemma then_cond_sim [intro]: "\<lbrakk>cond_sim P T1 step_rel obs get_mem; cond_sim P T2
    step_rel obs get_mem;
  \<forall>\<tau> CFGs CFGs'. P \<tau> CFGs \<and> CFGs' \<in> trans_sf T1 \<tau> CFGs \<
      longrightarrow> P \<tau> CFGs'\<rbrakk> \<Longrightarrow>
  cond_sim P (TThen T1 T2) step_rel obs get_mem"
unfolding cond_sim_def proof clarify
  fix CFGs \<tau> CFGs' assume "CFGs' \<in> trans_sf (TThen T1 T2) \<tau> CFGs"
  hence "\<exists>CFGs''. CFGs'' \<in> trans_sf T1 \<tau> CFGs \<and> CFGs' \<in> trans_sf
      T2 \<tau> CFGs''" by simp
  then obtain CFGs'' where A: "CFGs'' \<in> trans_sf T1 \<tau> CFGs \<and> CFGs' \<in>
      trans_sf T2 \<tau> CFGs''" ..
  assume P: "P \<tau> CFGs" "tCFG CFGs instr_edges Seq" "\<forall>\<tau> CFGs. P \<tau>
      CFGs \<and> tCFG CFGs instr_edges Seq \<longrightarrow>
  (\<forall>CFGs'\<in>trans_sf T1 \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and> (\<exists>
      sim tlsim. tCFG_sim sim tlsim CFGs' CFGs step_rel obs get_mem))"
  from this A have tCFG'': "tCFG CFGs'' instr_edges Seq" "\<exists>sim tlsim. tCFG_sim sim
      tlsim CFGs'' CFGs step_rel obs get_mem" by auto
  then obtain sim tlsim where sim: "tCFG_sim sim tlsim CFGs'' CFGs step_rel obs get_mem" by
       auto
  assume "\<forall>\<tau> CFGs CFGs'. P \<tau> CFGs \<and> CFGs' \<in> trans_sf T1 \<tau>
      CFGs \<longrightarrow> P \<tau> CFGs'"
  from this P A have P'': "P \<tau> CFGs''" by metis
```

```
  assume "\<forall>\<tau> CFGs. P \<tau> CFGs \<and> tCFG CFGs instr_edges Seq \<
      longrightarrow> (\<forall>CFGs'\<in>trans_sf T2 \<tau> CFGs. tCFG CFGs'
  instr_edges Seq \<and> (\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs step_rel obs
      get_mem))"
  from this P'' tCFG'' A have tCFG': "tCFG CFGs' instr_edges Seq"
    "\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs'' step_rel obs get_mem" by auto
  then obtain sim' tlsim' where sim': "tCFG_sim sim' tlsim' CFGs' CFGs'' step_rel obs
      get_mem" by auto
  show "tCFG CFGs' instr_edges Seq \<and> (\<exists>sim tlsim. tCFG_sim sim tlsim CFGs'
      CFGs step_rel obs get_mem)"
  proof
    from tCFG' show "tCFG CFGs' instr_edges Seq" by simp
    next show "\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs step_rel obs get_mem"
    proof ((rule exI)+, rule sim_comp)
      from sim show "tCFG_sim sim tlsim CFGs'' CFGs step_rel obs get_mem" .
      from sim' show "tCFG_sim sim' tlsim' CFGs' CFGs'' step_rel obs get_mem" .
    qed
  qed
qed

lemma applysome_cond_sim: "\<lbrakk>apply_some (trans_sf T) \<tau> CFGs CFGs'; cond_sim P T
     step_rel obs get_mem;
  \<forall>\<tau> CFGs CFGs'. (P \<tau> CFGs \<and> CFGs' \<in> trans_sf T \<tau> CFGs) \<
      longrightarrow> P \<tau> CFGs'; P \<tau> CFGs; tCFG CFGs instr_edges Seq\<rbrakk> \<
      Longrightarrow>
  tCFG CFGs' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_sim sim' tlsim' CFGs' CFGs
      step_rel obs get_mem)"
unfolding cond_sim_def proof (induct rule: apply_some.induct)
  fix G::"'thread \<Rightarrow> ('node, 'edge_type, 'instr) flowgraph option" assume "tCFG
      G instr_edges Seq"
  thus "tCFG G instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_sim sim' tlsim' G G
      step_rel obs get_mem)"
  proof
    have "tCFG_sim op = op = G G step_rel obs get_mem" by (unfold_locales, simp)
    thus "\<exists>sim' tlsim'. tCFG_sim sim' tlsim' G G step_rel obs get_mem" by force
  qed
next
  case (apply_more G' T \<tau> G G'') hence G'': "tCFG G'' instr_edges Seq \<and>
  (\<exists>sim' tlsim'. tCFG_sim sim' tlsim' G'' G' step_rel obs get_mem)" by metis
```

```
  then obtain sim' tlsim' where sim': "tCFG_sim sim' tlsim' G'' G' step_rel obs get_mem" by
      force
  moreover assume "G' \<in> T \<tau> G" "P \<tau> G" "tCFG G instr_edges Seq"
  "\<forall>\<tau> CFGs. P \<tau> CFGs \<and> tCFG CFGs instr_edges Seq \<longrightarrow>
      (\<forall>CFGs'\<in>T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
  (\<exists>sim tlsim. tCFG_sim sim tlsim CFGs' CFGs step_rel obs get_mem))"
  then obtain sim tlsim where sim: "tCFG_sim sim tlsim G' G step_rel obs get_mem" by force
  have "\<exists>sim' tlsim'. tCFG_sim sim' tlsim' G'' G step_rel obs get_mem"
    proof ((rule exI)+, rule sim_comp)
      show "tCFG_sim sim' tlsim' G'' G' step_rel obs get_mem" by (rule sim')
      next show "tCFG_sim sim tlsim G' G step_rel obs get_mem" by (rule sim)
    qed
  from this G'' show ?case by simp
qed

lemma applyall_cond_sim [intro]: "\<lbrakk>cond_sim P T step_rel obs get_mem;
  \<forall>\<tau> CFGs CFGs'. P \<tau> CFGs \<and> CFGs' \<in> trans_sf T \<tau> CFGs \<
      longrightarrow> P \<tau> CFGs'\<rbrakk> \<Longrightarrow>
  cond_sim P (TApplyAll T) step_rel obs get_mem"
by (clarsimp intro!: cond_simI simp add: applysome_cond_sim)

end

(* stuttering simulation *)
locale delay_bisimulation_base =
  bisimulation_base +
  trsys1: \<tau>trsys trsys1 \<tau>move1 +
  trsys2: \<tau>trsys trsys2 \<tau>move2
  for \<tau>move1 \<tau>move2 +
  constrains trsys1 :: "('s1, 'tl1) trsys"
  and trsys2 :: "('s2, 'tl2) trsys"
  and bisim :: "('s1, 's2) bisim"
  and tlsim :: "('tl1, 'tl2) bisim"
  and \<tau>move1 :: "('s1, 'tl1) trsys"
  and \<tau>move2 :: "('s2, 'tl2) trsys"
begin

notation
  trsys1.silent_move ("_/ -\<tau>1\<rightarrow> _" [50, 50] 60) and
  trsys2.silent_move ("_/ -\<tau>2\<rightarrow> _" [50, 50] 60)
```

```
notation
  trsys1.silent_moves ("_/ -\<tau>1\<rightarrow>* _" [50, 50] 60) and
  trsys2.silent_moves ("_/ -\<tau>2\<rightarrow>* _" [50, 50] 60)

notation
  trsys1.silent_movet ("_/ -\<tau>1\<rightarrow>+ _" [50, 50] 60) and
  trsys2.silent_movet ("_/ -\<tau>2\<rightarrow>+ _" [50, 50] 60)

notation
  trsys1.\<tau>rtrancl3p ("_ -\<tau>1-_\<rightarrow>* _" [50, 0, 50] 60) and
  trsys2.\<tau>rtrancl3p ("_ -\<tau>2-_\<rightarrow>* _" [50, 0, 50] 60)

notation
  trsys1.\<tau>inf_step ("_ -\<tau>1-_\<rightarrow>* \<infinity>" [50, 0] 80) and
  trsys2.\<tau>inf_step ("_ -\<tau>2-_\<rightarrow>* \<infinity>" [50, 0] 80)

notation
  trsys1.\<tau>diverge ("_ -\<tau>1\<rightarrow> \<infinity>" [50] 80) and
  trsys2.\<tau>diverge ("_ -\<tau>2\<rightarrow> \<infinity>" [50] 80)

notation
  trsys1.\<tau>inf_step_table ("_ -\<tau>1-_\<rightarrow>*t \<infinity>" [50, 0] 80) and
  trsys2.\<tau>inf_step_table ("_ -\<tau>2-_\<rightarrow>*t \<infinity>" [50, 0] 80)

notation
  trsys1.\<tau>Runs ("_ \<Down>1 _" [50, 50] 51) and
  trsys2.\<tau>Runs ("_ \<Down>2 _" [50, 50] 51)

lemma simulation_silent1I':
  assumes "\<exists>s2'. (if \<mu>1 s1' s1 then trsys2.silent_moves else trsys2.
      silent_movet) s2 s2' \<and> s1' \<approx> s2'"
  shows "s1' \<approx> s2 \<and> \<mu>1^++ s1' s1 \<or> (\<exists>s2'. s2 -\<tau>2\<
      rightarrow>+ s2' \<and> s1' \<approx> s2')"
proof -
  from assms obtain s2' where red: "(if \<mu>1 s1' s1 then trsys2.silent_moves else trsys2.
      silent_movet) s2 s2'"
    and bisim: "s1' \<approx> s2'" by blast
  show ?thesis
  proof(cases "\<mu>1 s1' s1")
```

```
    case True
    with red have "s2 -\<tau>2\<rightarrow>* s2'" by simp
    thus ?thesis using bisim True by cases(blast intro: rtranclp_into_tranclp1)+
  next
    case False
    with red bisim show ?thesis by auto
  qed
qed

lemma simulation_silent2I':
  assumes "\<exists>s1'. (if \<mu>2 s2' s2 then trsys1.silent_moves else trsys1.
      silent_movet) s1 s1' \<and> s1' \<approx> s2'"
  shows "s1 \<approx> s2' \<and> \<mu>2^++ s2' s2 \<or> (\<exists>s1'. s1 -\<tau>1\<
      rightarrow>+ s1' \<and> s1' \<approx> s2')"
using assms
by(rule delay_bisimulation_base.simulation_silent1I')

end

locale delay_simulation_obs = delay_bisimulation_base _ _ _ _ \<tau>move1 \<tau>move2
  for \<tau>move1 :: "'s1 => 'tl1 => 's1 => bool"
  and \<tau>move2 :: "'s2 => 'tl2 => 's2 => bool" +
  assumes simulation:
  "\<lbrakk> s1 \<approx> s2; s1 -1-tl1\<rightarrow> s1'; \<not> \<tau>move1 s1 tl1 s1' \<
      rbrakk>
  \<Longrightarrow> \<exists>s2' s2'' tl2. s2 -\<tau>2\<rightarrow>* s2' \<and> s2' -2-tl2
      \<rightarrow> s2'' \<and> \<not> \<tau>move2 s2' tl2 s2'' \<and> s1' \<approx> s2'' \<
      and> tl1 \<sim> tl2"

locale delay_simulation_diverge = delay_simulation_obs _ _ _ _ \<tau>move1 \<tau>move2
  for \<tau>move1 :: "'s1 => 'tl1 => 's1 => bool"
  and \<tau>move2 :: "'s2 => 'tl2 => 's2 => bool" +
  assumes simulation_silent:
  "[| s1 \<approx> s2; s1 -\<tau>1\<rightarrow> s1' |] ==> \<exists>s2'. s2 -\<tau>2\<
      rightarrow>* s2' \<and> s1' \<approx> s2'"
  and \<tau>diverge_sim_inv: "s1 \<approx> s2 ==> s1 -\<tau>1\<rightarrow> \<infinity> ==>
      s2 -\<tau>2\<rightarrow> \<infinity>"
begin

lemma simulation_silents:
```

```
  assumes bisim: "s1 \<approx> s2" and moves: "s1 -\<tau>1\<rightarrow>* s1'"
  shows "\<exists>s2'. s2 -\<tau>2\<rightarrow>* s2' \<and> s1' \<approx> s2'"
using moves bisim
proof induct
  case base thus ?case by(blast)
next
  case (step s1' s1'')
  from `s1 \<approx> s2 ==> \<exists>s2'. s2 -\<tau>2\<rightarrow>* s2' \<and> s1' \<approx
      > s2'` `s1 \<approx> s2`
  obtain s2' where "s2 -\<tau>2\<rightarrow>* s2'" "s1' \<approx> s2'" by blast
  from simulation_silent[OF `s1' \<approx> s2'` `s1' -\<tau>1\<rightarrow> s1''`]
  obtain s2'' where "s2' -\<tau>2\<rightarrow>* s2''" "s1'' \<approx> s2''" by blast
  from `s2 -\<tau>2\<rightarrow>* s2'` `s2' -\<tau>2\<rightarrow>* s2''` have "s2 -\<tau
      >2\<rightarrow>* s2''" by(rule rtranclp_trans)
  with `s1'' \<approx> s2''` show ?case by blast
qed

lemma simulation_\<tau>rtrancl3p:
  "[| s1 -\<tau>1-tls1\<rightarrow>* s1'; s1 \<approx> s2 |]
  ==> \<exists>tls2 s2'. s2 -\<tau>2-tls2\<rightarrow>* s2' \<and> s1' \<approx> s2' \<and>
       tls1 [\<sim>] tls2"
proof(induct arbitrary: s2 rule: trsys1.\<tau>rtrancl3p.induct)
  case (\<tau>rtrancl3p_refl s)
  thus ?case by(auto intro: \<tau>trsys.\<tau>rtrancl3p.intros)
next
  case (\<tau>rtrancl3p_step s1 s1' tls1 s1'' tl1)
  from simulation[OF `s1 \<approx> s2` `s1 -1-tl1\<rightarrow> s1'` `\<not> \<tau>move1 s1
      tl1 s1'`]
  obtain s2' s2'' tl2 where \<tau>red: "s2 -\<tau>2\<rightarrow>* s2'"
    and red: "s2' -2-tl2\<rightarrow> s2''" and n\<tau>: "\<not> \<tau>move2 s2' tl2 s2''"
    and bisim': "s1' \<approx> s2''" and tlsim: "tl1 \<sim> tl2" by blast
  from bisim' `s1' \<approx> s2'' ==> \<exists>tls2 s2'. s2'' -\<tau>2-tls2\<rightarrow>*
      s2' \<and> s1'' \<approx> s2' \<and> tls1 [\<sim>] tls2`
  obtain tls2 s2''' where IH: "s2'' -\<tau>2-tls2\<rightarrow>* s2'''" "s1'' \<approx> s2
      '''" "tls1 [\<sim>] tls2" by blast
  from \<tau>red have "s2 -\<tau>2-[]\<rightarrow>* s2'" by(rule trsys2.silent_moves_into_
      \<tau>rtrancl3p)
  also from red n\<tau> IH(1) have "s2' -\<tau>2-tl2 # tls2\<rightarrow>* s2'''" by(rule \<
      tau>rtrancl3p.\<tau>rtrancl3p_step)
  finally show ?case using IH tlsim by fastforce
```

```
next
  case (\<tau>rtrancl3p_\<tau>step s1 s1' tls1 s1'' tl1)
  from `s1 -1-tl1\<rightarrow> s1'` `\<tau>move1 s1 tl1 s1'` have "s1 -\<tau>1\<rightarrow>
       s1'" ..
  from simulation_silent[OF `s1 \<approx> s2` this]
  obtain s2' where \<tau>red: "s2 -\<tau>2\<rightarrow>* s2'" and bisim': "s1' \<approx> s2
      '" by blast
  from \<tau>red have "s2 -\<tau>2-[]\<rightarrow>* s2'" by(rule trsys2.silent_moves_into_
      \<tau>rtrancl3p)
  also from bisim' `s1' \<approx> s2' ==> \<exists>tls2 s2''. s2' -\<tau>2-tls2\<rightarrow
      >* s2'' \<and> s1'' \<approx> s2'' \<and> tls1 [\<sim>] tls2`
  obtain tls2 s2'' where IH: "s2' -\<tau>2-tls2\<rightarrow>* s2''" "s1'' \<approx> s2''" "
      tls1 [\<sim>] tls2" by blast
  note `s2' -\<tau>2-tls2\<rightarrow>* s2''`
  finally show ?case using IH by auto
qed

lemma simulation_\<tau>inf_step:
  assumes \<tau>inf1: "s1 -\<tau>1-tls1\<rightarrow>* \<infinity>" and bisim: "s1 \<approx>
       s2"
  shows "\<exists>tls2. s2 -\<tau>2-tls2\<rightarrow>* \<infinity> \<and> tls1 [[\<sim>]]
      tls2"
proof -
  from trsys1.\<tau>inf_step_imp_\<tau>inf_step_table[OF \<tau>inf1]
  obtain sstls1 where \<tau>inf1': "s1 -\<tau>1-sstls1\<rightarrow>*t \<infinity>"
    and tls1: "tls1 = lmap (fst o snd o snd) sstls1" by blast
  def tl1_to_tl2 \<equiv> "\<lambda>(s2 :: 's2) (sstls1 :: ('s1 \<times> 's1 \<times> 'tl1
      \<times> 's1) llist). llist_unfold
     (\<lambda>(s2, sstls1). lnull sstls1)
     (\<lambda>(s2, sstls1).
        let (s1, s1', tl1, s1'') = lhd sstls1;
            (s2', tl2, s2'') = SOME (s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and>
                trsys2 s2' tl2 s2'' \<and>
                                     \<not> \<tau>move2 s2' tl2 s2'' \<and>  s1'' \<approx>
                                          s2'' \<and> tl1 \<sim> tl2
        in (s2, s2', tl2, s2''))
     (\<lambda>(s2, sstls1).
        let (s1, s1', tl1, s1'') = lhd sstls1;
            (s2', tl2, s2'') = SOME (s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and>
                trsys2 s2' tl2 s2'' \<and>
```

```
                              \<not> \<tau>move2 s2' tl2 s2'' \<and>  s1'' \<approx>
                                 s2'' \<and> tl1 \<sim> tl2
    in (s2'', ltl sstls1))
   (s2, sstls1)"
have [simp]:
  "!!s2 sstls1. lnull (tl1_to_tl2 s2 sstls1) <-> lnull sstls1"
  "!!s2 sstls1. \<not> lnull sstls1 ==> lhd (tl1_to_tl2 s2 sstls1) =
    (let (s1, s1', tl1, s1'') = lhd sstls1;
       (s2', tl2, s2'') = SOME (s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and>
          trsys2 s2' tl2 s2'' \<and>
                              \<not> \<tau>move2 s2' tl2 s2'' \<and>  s1'' \<approx>
                                 s2'' \<and> tl1 \<sim> tl2
    in (s2, s2', tl2, s2''))"
  "!!s2 sstls1. \<not> lnull sstls1 ==> ltl (tl1_to_tl2 s2 sstls1) =
    (let (s1, s1', tl1, s1'') = lhd sstls1;
       (s2', tl2, s2'') = SOME (s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and>
          trsys2 s2' tl2 s2'' \<and>
                              \<not> \<tau>move2 s2' tl2 s2'' \<and>  s1'' \<approx>
                                 s2'' \<and> tl1 \<sim> tl2
    in tl1_to_tl2 s2'' (ltl sstls1))"
  "!!s2. tl1_to_tl2 s2 LNil = LNil"
  "!!s2 s1 s1' tl1 s1'' stls1'. tl1_to_tl2 s2 (LCons (s1, s1', tl1, s1'') stls1') =
    LCons (s2, SOME (s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and> trsys2 s2'
       tl2 s2'' \<and>
                                 \<not> \<tau>move2 s2' tl2 s2'' \<and> s1'' \<
                                    approx> s2'' \<and> tl1 \<sim> tl2)
         (tl1_to_tl2 (snd (snd (SOME (s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2'
            \<and> trsys2 s2' tl2 s2'' \<and>
                                                   \<not> \<tau>move2 s2' tl2 s2''
                                                       \<and> s1'' \<approx> s2''
                                                      \<and> tl1 \<sim> tl2)))
                        stls1')"
  by(simp_all add: tl1_to_tl2_def split_beta)

have [simp]: "llength (tl1_to_tl2 s2 sstls1) = llength sstls1"
  apply (coinduction arbitrary: s2 sstls1 rule: enat_coinduct) apply (auto simp add:
    epred_llength split_beta llength_def)
  apply (metis `\<And>sstls1 s2. lnull (tl1_to_tl2 s2 sstls1) = lnull sstls1`
    enat_unfold_eq_0 zero_enat_def)
  apply (metis enat_unfold_eq_0 zero_enat_def)
```

```
done

def sstls2: sstls2 \<equiv> "tl1_to_tl2 s2 sstls1"
with \<tau>inf1' bisim have "\<exists>s1 sstls1. s1 -\<tau>1-sstls1\<rightarrow>*t \<
    infinity> \<and> sstls2 = tl1_to_tl2 s2 sstls1 \<and> s1 \<approx> s2" by blast

from \<tau>inf1' bisim have "s2 -\<tau>2-tl1_to_tl2 s2 sstls1\<rightarrow>*t \<infinity>"
proof(coinduction arbitrary: s2 s1 sstls1)
  case (\<tau>inf_step_table s2 s1 sstls1)
  note \<tau>inf' = `s1 -\<tau>1-sstls1\<rightarrow>*t \<infinity>` and bisim = `s1 \<
      approx> s2`
  from \<tau>inf' show ?case
  proof(cases)
    case (\<tau>inf_step_table_Cons s1' s1'' sstls1' tl1)
    hence sstls1: "sstls1 = LCons (s1, s1', tl1, s1'') sstls1'"
      and \<tau>s: "s1 -\<tau>1\<rightarrow>* s1'" and r: "s1' -1-tl1\<rightarrow> s1''"
          and n\<tau>: "\<not> \<tau>move1 s1' tl1 s1''"
      and reds1: "s1'' -\<tau>1-sstls1'\<rightarrow>*t \<infinity>" by simp_all
    let ?P = "\<lambda>(s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and> trsys2 s2'
        tl2 s2'' \<and> \<not> \<tau>move2 s2' tl2 s2'' \<and>  s1'' \<approx> s2'' \<and>
         tl1 \<sim> tl2"
    let ?s2tl2s2' = "Eps ?P"
    let ?s2'' = "snd (snd ?s2tl2s2')"
    from simulation_silents[OF `s1 \<approx> s2` \<tau>s]
    obtain s2' where "s2 -\<tau>2\<rightarrow>* s2'" "s1' \<approx> s2'" by blast
    from simulation[OF `s1' \<approx> s2'` r n\<tau>] obtain s2'' s2''' tl2
      where "s2' -\<tau>2\<rightarrow>* s2''"
      and rest: "s2'' -2-tl2\<rightarrow> s2'''" "\<not> \<tau>move2 s2'' tl2 s2'''" "s1
          '' \<approx> s2'''" "tl1 \<sim> tl2" by blast
    from `s2 -\<tau>2\<rightarrow>* s2'` `s2' -\<tau>2\<rightarrow>* s2''` have "s2 -\<
        tau>2\<rightarrow>* s2''" by(rule rtranclp_trans)
    with rest have "?P (s2'', tl2, s2''')" by simp
    hence "?P ?s2tl2s2'" by(rule someI)
    then show ?thesis using reds1 sstls1 by fastforce
  next
    case \<tau>inf_step_table_Nil
    hence [simp]: "sstls1 = LNil" and "s1 -\<tau>1\<rightarrow> \<infinity>" by simp_all
    from `s1 -\<tau>1\<rightarrow> \<infinity>` `s1 \<approx> s2` have "s2 -\<tau>2\<
        rightarrow> \<infinity>" by(simp add: \<tau>diverge_sim_inv)
    thus ?thesis using sstls2 by simp
```

```
  qed
qed
hence "s2 -\<tau>2-lmap (fst o snd o snd) (tl1_to_tl2 s2 sstls1)\<rightarrow>* \<infinity
    >"
  by(rule trsys2.\<tau>inf_step_table_into_\<tau>inf_step)
moreover have "tls1 [[\<sim>]] lmap (fst o snd o snd) (tl1_to_tl2 s2 sstls1)"
proof(rule llist_all2_all_lnthI)
  show "llength tls1 = llength (lmap (fst o snd o snd) (tl1_to_tl2 s2 sstls1))"
    using tls1 by simp
next
  fix n
  assume "enat n < llength tls1"
  thus "lnth tls1 n \<sim> lnth (lmap (fst o snd o snd) (tl1_to_tl2 s2 sstls1)) n"
    using \<tau>inf1' bisim unfolding tls1
  proof(induct n arbitrary: s1 s2 sstls1 rule: less_induct)
    case (less n)
    note IH = `!!m s1 s2 sstls1. [| m < n; enat m < llength (lmap (fst o snd o snd)
        sstls1);
                                 s1 -\<tau>1-sstls1\<rightarrow>*t \<infinity>; s1 \<
                                     approx> s2 |]
               ==> lnth (lmap (fst o snd o snd) sstls1) m \<sim> lnth (lmap (fst o snd o
                   snd) (tl1_to_tl2 s2 sstls1)) m`
    from `s1 -\<tau>1-sstls1\<rightarrow>*t \<infinity>` show ?case
    proof cases
      case (\<tau>inf_step_table_Cons s1' s1'' sstls1' tl1)
      hence sstls1: "sstls1 = LCons (s1, s1', tl1, s1'') sstls1'"
        and \<tau>s: "s1 -\<tau>1\<rightarrow>* s1'" and r: "s1' -1-tl1\<rightarrow> s1
            ''"
        and n\<tau>: "\<not> \<tau>move1 s1' tl1 s1''" and reds: "s1'' -\<tau>1-sstls1'\<
            rightarrow>*t \<infinity>" by simp_all
      let ?P = "\<lambda>(s2', tl2, s2''). s2 -\<tau>2\<rightarrow>* s2' \<and> trsys2 s2
          ' tl2 s2'' \<and> \<not> \<tau>move2 s2' tl2 s2'' \<and>  s1'' \<approx> s2'' \<
          and> tl1 \<sim> tl2"
      let ?s2tl2s2' = "Eps ?P" let ?tl2 = "fst (snd ?s2tl2s2')" let ?s2'' = "snd (snd ?
          s2tl2s2')"
      from simulation_silents[OF `s1 \<approx> s2` \<tau>s] obtain s2'
        where "s2 -\<tau>2\<rightarrow>* s2'" "s1' \<approx> s2'" by blast
      from simulation[OF `s1' \<approx> s2'` r n\<tau>] obtain s2'' s2''' tl2
        where "s2' -\<tau>2\<rightarrow>* s2''"
```

```
        and rest: "s2'' -2-tl2\<rightarrow> s2'''" "\<not> \<tau>move2 s2'' tl2 s2'''" "
            s1'' \<approx> s2'''" "tl1 \<sim> tl2" by blast
      from `s2 -\<tau>2\<rightarrow>* s2'` `s2' -\<tau>2\<rightarrow>* s2''` have "s2 -\<
          tau>2\<rightarrow>* s2''" by(rule rtranclp_trans)
      with rest have "?P (s2'', tl2, s2''')" by auto
      hence "?P ?s2tl2s2'" by(rule someI)
      hence "s1'' \<approx> ?s2''" "tl1 \<sim> ?tl2" by(simp_all add: split_beta)
      show ?thesis
      proof(cases n)
        case 0
        with sstls1 `tl1 \<sim> ?tl2` show ?thesis by simp
      next
        case (Suc m)
        hence "m < n" by simp
        moreover have "enat m < llength (lmap (fst o snd o snd) sstls1')"
          using sstls1 `enat n < llength (lmap (fst o snd o snd) sstls1)` Suc by(simp add
              : Suc_ile_eq)
        ultimately have "lnth (lmap (fst o snd o snd) sstls1') m \<sim> lnth (lmap (fst o
             snd o snd) (tl1_to_tl2 ?s2'' sstls1')) m"
          using reds `s1'' \<approx> ?s2''` by(rule IH)
        with Suc sstls1 show ?thesis by(simp del: o_apply)
      qed
    next
      case \<tau>inf_step_table_Nil
      with `enat n < llength (lmap (fst o snd o snd) sstls1)` have False by simp
      thus ?thesis ..
    qed
  qed
qed
ultimately show ?thesis by blast
qed

end

sublocale delay_bisimulation_diverge \<subseteq> delay_simulation_diverge
by (unfold_locales, rule simulation1, auto simp add: simulation1 simulation_silent1 \<tau>
    diverge_bisim_inv)

definition \<tau>moves_of_tCFG where
```

```
"\<tau>moves_of_tCFG CFGs step_rel obs get_mem C l C' \<equiv> step_rel CFGs C C' \<and>
    get_mem C' |` obs = l
  \<and> get_mem C |` obs = l"

locale tCFG_delay_bisim = delay_bisimulation_diverge where trsys1 = "trsys_of_tCFG CFGs
    step_rel obs get_mem"
 and trsys2 = "trsys_of_tCFG CFGs' step_rel obs get_mem"
 and \<tau>move1 = "\<tau>moves_of_tCFG CFGs step_rel obs get_mem"
 and \<tau>move2 = "\<tau>moves_of_tCFG CFGs' step_rel obs get_mem" for CFGs CFGs' step_rel
      obs get_mem

locale tCFG_delay_sim = delay_simulation_diverge where trsys1 = "trsys_of_tCFG CFGs
    step_rel obs get_mem"
 and trsys2 = "trsys_of_tCFG CFGs' step_rel obs get_mem"
 and \<tau>move1 = "\<tau>moves_of_tCFG CFGs step_rel obs get_mem"
 and \<tau>move2 = "\<tau>moves_of_tCFG CFGs' step_rel obs get_mem" for CFGs CFGs' step_rel
      obs get_mem

lemma delay_bisim_comp: "\<lbrakk>tCFG_delay_bisim sim tlsim CFGs CFGs' step_rel obs
    get_mem;
  tCFG_delay_bisim sim' tlsim' CFGs' CFGs'' step_rel obs get_mem\<rbrakk> \<Longrightarrow>
tCFG_delay_bisim (sim \<circ>\<^sub>B sim') (tlsim \<circ>\<^sub>B tlsim') CFGs CFGs''
    step_rel obs get_mem"
by (simp add: tCFG_delay_bisim_def delay_bisimulation_diverge_compose)

context TRANS_basics begin

definition (in TRANS_basics) "opt_delay_bisim T sim tlsim step_rel obs get_mem \<equiv>
  \<forall>\<tau> CFGs. tCFG CFGs instr_edges Seq \<longrightarrow> (\<forall>CFGs'\<in>
      trans_sf T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
  tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem)"

lemma opt_delay_bisimI [intro]: "(\<And>\<tau> CFGs CFGs'. \<lbrakk>tCFG CFGs instr_edges
    Seq; CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow>
  tCFG CFGs' instr_edges Seq \<and> tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs
      get_mem) \<Longrightarrow>
  opt_delay_bisim T sim tlsim step_rel obs get_mem"
by (simp add: opt_delay_bisim_def)

lemma opt_delay_bisimD [dest]: "\<lbrakk>opt_delay_bisim T sim tlsim step_rel obs get_mem;
```

```
  tCFG CFGs instr_edges Seq; CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow>
      tCFG CFGs' instr_edges Seq \<and>
  tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem"
by (simp add: opt_delay_bisim_def)

lemma then_delay_bisim [intro]: "\<lbrakk>opt_delay_bisim T1 sim1 tlsim1 step_rel obs
    get_mem;
  opt_delay_bisim T2 sim2 tlsim2 step_rel obs get_mem\<rbrakk> \<Longrightarrow>
  opt_delay_bisim (TThen T1 T2) (sim2 \<circ>\<^sub>B sim1) (tlsim2 \<circ>\<^sub>B tlsim1)
       step_rel obs get_mem"
apply (clarsimp intro!: opt_delay_bisimI)
apply (drule opt_delay_bisimD, simp+, clarsimp)+
apply (rule delay_bisim_comp, simp_all)
done
(* These work much better than just unfolding the definition. *)

lemma applysome_delay_bisim: "\<lbrakk>opt_delay_bisim T sim tlsim step_rel obs get_mem;
  tCFG CFGs instr_edges Seq; apply_some (trans_sf T) \<tau> CFGs CFGs'\<rbrakk> \<
      Longrightarrow>
  tCFG CFGs' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_delay_bisim sim' tlsim'
      CFGs' CFGs step_rel obs get_mem)"
apply (drule_tac P="\<lambda>t \<tau> G G'. t = trans_sf T \<and> tCFG G instr_edges Seq \<
    longrightarrow>
  (tCFG G' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_delay_bisim sim' tlsim' G' G
      step_rel obs get_mem))"
  in apply_some.induct, auto)
apply (rule_tac x="op =" in exI, rule_tac x="op =" in exI, unfold_locales, force+)
apply (drule opt_delay_bisimD, simp+, clarsimp)
apply (rule exI, rule exI, rule delay_bisim_comp, simp+)
done

definition "trans_delay_bisim T step_rel obs get_mem \<equiv>
  \<forall>\<tau> CFGs. tCFG CFGs instr_edges Seq \<longrightarrow> (\<forall>CFGs' \<in>
      trans_sf T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
  (\<exists>sim tlsim. tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem))"

lemma trans_delay_bisimI [intro]: "(\<And>\<tau> CFGs CFGs'. \<lbrakk>tCFG CFGs instr_edges
     Seq; CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow>
  tCFG CFGs' instr_edges Seq \<and> (\<exists>sim tlsim. tCFG_delay_bisim sim tlsim CFGs'
      CFGs step_rel obs get_mem)) \<Longrightarrow>
```

```
  trans_delay_bisim T step_rel obs get_mem"
by (simp add: trans_delay_bisim_def)

lemma trans_delay_bisimD [dest]: "\<lbrakk>trans_delay_bisim T step_rel obs get_mem;
  tCFG CFGs instr_edges Seq; CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow>
      tCFG CFGs' instr_edges Seq \<and>
  (\<exists>sim tlsim. tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem)"
by (simp add: trans_delay_bisim_def)

lemma opt_delay_bisim_trans [intro]: "opt_delay_bisim T sim tlsim step_rel obs get_mem \<
    Longrightarrow>
  trans_delay_bisim T step_rel obs get_mem"
by (force simp add: opt_delay_bisim_def trans_delay_bisim_def)

definition "cond_delay_bisim P T step_rel obs get_mem \<equiv> \<forall>\<tau> CFGs. P \<
    tau> CFGs \<and> tCFG CFGs instr_edges Seq \<longrightarrow>
  (\<forall>CFGs' \<in> trans_sf T \<tau> CFGs. tCFG CFGs' instr_edges Seq \<and>
  (\<exists>sim tlsim. tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem))"

lemma cond_delay_bisimI [intro]: "(\<And>\<tau> CFGs CFGs'. \<lbrakk>P \<tau> CFGs; tCFG
    CFGs instr_edges Seq;
  CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow> tCFG CFGs' instr_edges Seq
      \<and>
  (\<exists>sim tlsim. tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem)) \<
      Longrightarrow>
  cond_delay_bisim P T step_rel obs get_mem"
by (force simp add: cond_delay_bisim_def)

lemma cond_delay_bisimD [dest]: "\<lbrakk>cond_delay_bisim P T step_rel obs get_mem; P \<
    tau> CFGs;
  tCFG CFGs instr_edges Seq; CFGs' \<in> trans_sf T \<tau> CFGs\<rbrakk> \<Longrightarrow>
      tCFG CFGs' instr_edges Seq \<and>
  (\<exists>sim tlsim. tCFG_delay_bisim sim tlsim CFGs' CFGs step_rel obs get_mem)"
by (force simp add: cond_delay_bisim_def)

lemma applyall_delay_bisim [intro]: "opt_delay_bisim T sim tlsim step_rel obs get_mem \<
    Longrightarrow>
  trans_delay_bisim (TApplyAll T) step_rel obs get_mem"
apply (clarsimp simp add: trans_delay_bisim_def)
by (metis applysome_delay_bisim)
```

```
lemma match_cond_delay_bisim [intro]: "cond_delay_bisim (\<lambda>\<tau> CFGs. \<exists>\<
    sigma> \<tau>'. side_cond_sf \<phi> \<sigma> CFGs \<and>
  part_matches \<sigma> \<tau>' \<and> \<tau> = part_extend \<tau>' (cond_fv pred_fv \<phi
      >) \<sigma>) T step_rel obs get_mem \<Longrightarrow>
  trans_delay_bisim (TMatch \<phi> T) step_rel obs get_mem"
apply (clarsimp simp add: trans_delay_bisim_def cond_delay_bisim_def)
by metis

lemma then_cond_delay_bisim [intro]: "\<lbrakk>cond_delay_bisim P T1 step_rel obs get_mem;
  cond_delay_bisim P T2 step_rel obs get_mem;
  \<forall>\<tau> CFGs CFGs'. P \<tau> CFGs \<and> CFGs' \<in> trans_sf T1 \<tau> CFGs \<
      longrightarrow> P \<tau> CFGs'\<rbrakk> \<Longrightarrow>
  cond_delay_bisim P (TThen T1 T2) step_rel obs get_mem"
apply (clarsimp intro!: cond_delay_bisimI)
apply (drule cond_delay_bisimD, simp+, clarsimp)
apply (erule_tac x=\<tau> in allE, erule_tac x=CFGs in allE, erule_tac x=CFGs'a in allE,
    simp)
apply (drule cond_delay_bisimD, simp+, clarsimp)
apply (rule exI, rule exI, rule delay_bisim_comp, simp+)
done

lemma applysome_cond_delay_bisim: "\<lbrakk>apply_some (trans_sf T) \<tau> CFGs CFGs';
  tCFG CFGs instr_edges Seq; cond_delay_bisim P T step_rel obs get_mem;
  \<forall>\<tau> CFGs CFGs'. P \<tau> CFGs \<and> CFGs' \<in> trans_sf T \<tau> CFGs \<
      longrightarrow> P \<tau> CFGs'; P \<tau> CFGs\<rbrakk> \<Longrightarrow>
  tCFG CFGs' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_delay_bisim sim' tlsim'
      CFGs' CFGs step_rel obs get_mem)"
apply (drule_tac P="\<lambda>t \<tau> G G'. t = trans_sf T \<and> P \<tau> G \<and> tCFG G
    instr_edges Seq \<longrightarrow>
  (tCFG G' instr_edges Seq \<and> (\<exists>sim' tlsim'. tCFG_delay_bisim sim' tlsim' G' G
      step_rel obs get_mem))"
  in apply_some.induct, simp_all)
apply clarsimp
apply (rule_tac x="op =" in exI, rule_tac x="op =" in exI, unfold_locales, force, force,
    force, force,
  force)
apply clarsimp
apply (erule_tac x=\<tau>' in allE, erule_tac x=G in allE, erule_tac x=G' in allE, simp)
apply (drule_tac CFGs=G in cond_delay_bisimD, simp+, clarsimp)
```

```
apply (rule exI, rule exI, rule delay_bisim_comp, simp+)
done

lemma applyall_cond_delay_bisim [intro]: "\<lbrakk>cond_delay_bisim P T step_rel obs
    get_mem;
  \<forall>\<tau> CFGs CFGs'. P \<tau> CFGs \<and> CFGs' \<in> trans_sf T \<tau> CFGs \<
      longrightarrow> P \<tau> CFGs'\<rbrakk> \<Longrightarrow>
  cond_delay_bisim P (TApplyAll T) step_rel obs get_mem"
apply (clarsimp intro!: cond_delay_bisimI)
apply (drule applysome_cond_delay_bisim, force simp add: cond_delay_bisim_def)
apply simp+
done

end

end

(* memory_model.thy *)
(* William Mansky *)
(* Memory model locales for PTRANS. *)

theory memory_model
imports "~~/src/AFP/List-Infinite/ListInfinite"
begin

lemma map_add_dom_upd [simp]: "dom m' = {k} \<Longrightarrow> (m ++ m')(k \<mapsto> v) = m(
    k \<mapsto> v)"
by (auto intro!: ext simp add: map_add_def split: option.splits)

lemma dud_set [simp]: "{(l, v). False} = {}"
by simp

(* Extra utility function: enumerate the elements of a set in arbitrary order.
   Useful for memory models. Could conceivably be replaced by linorder_class.
       sorted_list_of_set,
   though this is a bit more general. *)
function list_of_set where
"list_of_set S = (if infinite S \<or> S = {} then [] else let a = SOME a. a \<in> S in a #
    list_of_set (S - {a}))"
by pat_completeness auto
termination
```

```
apply (relation "measure (\<lambda>S. card S)", auto)
apply (frule card_Diff_singleton, rule someI, simp)
apply (case_tac "card S", simp_all)
done
lemma list_of_set_empty [simp]: "list_of_set {} = []"
by simp
lemma list_of_set_inf [simp]: "infinite S \<Longrightarrow> list_of_set S = []"
by simp
lemma list_of_set_card [simp]: "(list_of_set S \<noteq> []) = (card S \<noteq> 0)"
by (auto simp add: Let_def)
declare list_of_set.simps [simp del]

lemma set_some [simp]: "S \<noteq> {} \<Longrightarrow> insert (SOME a. a \<in> S) S = S"
by (metis insert_absorb not_ex_in_conv someI)

lemma set_some2 [simp]: "S \<noteq> {} \<Longrightarrow> (SOME a. a \<in> S) \<in> S"
by (metis not_ex_in_conv someI)

lemma list_of_set_set [simp]: "finite S \<Longrightarrow> set (list_of_set S) = S"
apply (induct "card S" arbitrary: S, simp_all)
apply (rule trans, simp add: list_of_set.simps, simp add: Let_def)
done

corollary list_of_set_nth: "\<lbrakk>list_of_set S ! i = x; i < length (list_of_set S)\<
    rbrakk> \<Longrightarrow> x \<in> S"
apply (subgoal_tac "finite S", subgoal_tac "x \<in> set (list_of_set S)", simp,
 simp add: set_conv_nth, force)
apply (auto simp add: list_of_set.simps split: if_splits)
done

lemma list_of_set_distinct [simp]: "distinct (list_of_set S)"
apply (induct "card S" arbitrary: S, clarsimp simp add: list_of_set.simps)
apply (rule_tac P=distinct in list_of_set.simps [THEN sym [THEN subst]], clarsimp simp add:
     Let_def)
done

datatype ('thread, 'loc, 'val) access = Read 'thread 'loc 'val | Write 'thread 'loc 'val
  | ARW 'thread 'loc 'val 'val | Alloc 'thread 'loc | Free 'thread 'loc
primrec get_thread where
"get_thread (Read t _ _) = t" |
```

```
"get_thread (Write t _ _) = t" |
"get_thread (ARW t _ _ _) = t" |
"get_thread (Alloc t _) = t" |
"get_thread (Free t _) = t"
primrec get_loc where
"get_loc (Read _ l _) = l" |
"get_loc (Write _ l _) = l" |
"get_loc (ARW _ l _ _) = l" |
"get_loc (Alloc _ l) = l" |
"get_loc (Free _ l) = l"

abbreviation "get_ptrs ops \<equiv> get_loc ` ops"

locale memory_model = fixes free_set::"'memory \<Rightarrow> 'loc set"
and can_read::"'memory \<Rightarrow> 'thread \<Rightarrow> 'loc \<Rightarrow> 'val set"
and update_mem::"'memory \<Rightarrow> ('thread, 'loc, 'val) access set \<Rightarrow> '
    memory \<Rightarrow> bool"
and start_mem::'memory
assumes alloc_not_free: "\<lbrakk>update_mem mem ops mem'; Alloc t l \<in> ops; \<forall>t.
     Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
    and stays_not_free: "\<lbrakk>update_mem mem ops mem'; l \<notin> free_set mem; \<
        forall>t. Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"

(* TSO memory model *)
locale TSO = fixes undef::'val begin
(* This isn't really a TSO-specific locale, but I might have other assumptions later.
   More to the point, it's convenient to have a separate locale for each memory model, even
        if
   they don't actually rely on separate assumptions. *)

abbreviation "free_set mem \<equiv> UNIV - dom (fst mem)"
definition "can_read mem t l \<equiv> case List.find (\<lambda>(l', v). l' = l) ((snd mem)
    t) of
 Some (l, v) \<Rightarrow> {v} | None \<Rightarrow> {v. fst mem l = Some v}"
definition "can_read2 mem t l \<equiv> case mem of (mem_map, bufs) \<Rightarrow> (case List
    .find (\<lambda>(l', v). l' = l) (bufs t) of
 Some (l, v) \<Rightarrow> {v} | None \<Rightarrow> {v. mem_map l = Some v})"
```

```
inductive update_mem where
no_atomic [intro]: "\<lbrakk>\<And>t l v v'. ARW t l v v' \<notin> ops; \<And>t. \<exists>
    up. bufs' t = up @ bufs t \<and>
 set up = {(l, v) | l v. Write t l v \<in> ops} \<and> distinct up\<rbrakk> \<
     Longrightarrow>
 update_mem (mem, bufs) ops (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> ops then Some undef else None), bufs')" |
update [intro]: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs'); bufs' t = buf @ [(l, v)
    ]\<rbrakk> \<Longrightarrow>
 update_mem (mem, bufs) ops (mem'(l \<mapsto> v), bufs'(t := buf))" |
atomic [intro!]: "bufs t = [] \<Longrightarrow> update_mem (mem, bufs) {ARW t l v v'} (mem(
    l \<mapsto> v'), bufs)"
abbreviation "start_mem \<equiv> (empty, \<lambda>t. [])"

lemma alloc_not_free: "\<lbrakk>update_mem mem ops mem'; Alloc t l \<in> ops; \<forall>t.
    Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
by (induct rule: update_mem.induct, auto split: if_splits)

lemma stays_not_free: "\<lbrakk>update_mem mem ops mem'; l \<notin> free_set mem; \<forall>
    t. Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
by (induct rule: update_mem.induct, auto split: if_splits)

end

sublocale TSO \<subseteq> memory_model free_set can_read update_mem start_mem
by (unfold_locales, metis alloc_not_free, metis stays_not_free)

context TSO begin

lemma update_none [intro!, simp]: "update_mem C {} C"
by (cases C, cut_tac ops="{}" and mem=a and bufs=b in no_atomic, auto simp add:
    restrict_map_def)

lemma can_read_thread: "\<lbrakk>v \<in> can_read (mem, b) t l; b t = b' t\<rbrakk> \<
    Longrightarrow>
 v \<in> can_read (mem, b') t l"
by (auto simp add: can_read_def split: option.splits)
```

```
lemma first_entry: "\<lbrakk>update_mem (mem, bufs) {Write t l v} (mem', bufs');
 bufs' t = a # rest\<rbrakk> \<Longrightarrow> a = (l, v)"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). \<forall>a rest. ops = {Write t
     l v} \<and>
 bufs' t = a # rest \<longrightarrow> a = (l, v)" in update_mem.induct, simp_all, clarsimp)
apply (subgoal_tac "\<exists>up. bufs'a t = up @ bufs t \<and> set up = {ab. t = t \<and> (
    case ab of (la, va) \<Rightarrow>
 la = l \<and> va = v)} \<and> distinct up", clarify, simp+)
apply (case_tac up, simp, simp)
apply (thin_tac "bufs'a t = ((aa, b) # resta)", force)
apply auto
apply (cases a, auto)
done

lemma update_map: "update_mem (mem, bufs) {} (mem', bufs') \<Longrightarrow>
 \<exists>map. \<forall>mem2. update_mem (mem2, bufs) {} (mem2 ++ map, bufs')"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). ops = {} \<longrightarrow>
 (\<exists>map. \<forall>mem2. update_mem (mem2, bufs) {} (mem2 ++ map, bufs'))" in
     update_mem.induct, auto)
apply (subgoal_tac "bufs' = bufs", rule_tac x=empty in exI, simp, rule ext)
apply (subgoal_tac "\<exists>up. bufs' x = (up @ bufs x) \<and> set up = {(l, v). False} \<
    and> distinct up", clarify, auto)
apply (rule_tac x="map(l \<mapsto> v)" in exI, auto)
done

lemma update_trans_rev: "\<lbrakk>update_mem (mem', bufs') {} (mem'', bufs'');
 update_mem (mem, bufs) ops (mem', bufs')\<rbrakk> \<Longrightarrow> update_mem (mem, bufs)
      ops (mem'', bufs'')"
apply (drule_tac P="\<lambda>(mem', bufs') ops' (mem'', bufs''). ops' = {} \<and>
    update_mem (mem, bufs) ops (mem', bufs') \<longrightarrow>
 update_mem (mem, bufs) ops (mem'', bufs'')" in update_mem.induct, auto simp add:
     restrict_map_def)
apply (subgoal_tac "bufs'a = bufsa", simp, rule ext)
apply (subgoal_tac "\<exists>up. bufs'a x = up @ bufsa x \<and> set up = {(l, v). False} \<
    and> distinct up",
 clarify, auto)
done

lemma update_trans [trans]: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs');
```

```
 update_mem (mem', bufs') {} (mem'', bufs'')\<rbrakk> \<Longrightarrow> update_mem (mem,
     bufs) ops (mem'', bufs'')"
by (erule update_trans_rev, simp)

lemma update_canonical: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs');
 \<And>t l v v'. ARW t l v v' \<notin> ops\<rbrakk> \<Longrightarrow>
 \<exists>writes bufs''. (\<forall>t. bufs'' t = writes t @ bufs t \<and> set (writes t) =
     {(l, v) | l v. Write t l v \<in> ops} \<and> distinct (writes t)) \<and>
 update_mem (mem, bufs) ops (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> ops then Some undef else None), bufs'') \<and>
 update_mem (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> ops then Some undef else None), bufs'') {} (
     mem', bufs')"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). (\<forall>t l v v'. ARW t l v v
    ' \<notin> ops) \<longrightarrow>
 (\<exists>writes bufs''. (\<forall>t. bufs'' t = writes t @ bufs t \<and>
 set (writes t) = {(l, v) | l v. Write t l v \<in> ops} \<and> distinct (writes t)) \<and>
     update_mem (mem, bufs) ops
 (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++ (\<lambda>l. if \<exists>t. Alloc
     t l \<in> ops then Some undef else None), bufs'') \<and>
 update_mem (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> ops then Some undef else None), bufs'') {} (
     mem', bufs'))" in update_mem.induct, auto)
apply (rule_tac x="\<lambda>t. SOME up. bufs' t = up @ bufs t \<and> set up = {(l, v).
    Write t l v \<in> opsa} \<and>
 distinct up" in exI)
apply (rule_tac x=bufs' in exI)
apply (rule conjI, clarsimp)
apply (rule someI_ex, auto)
done

corollary update_write: "\<lbrakk>update_mem (mem, bufs) {Write t l v} (mem', bufs')\<
    rbrakk> \<Longrightarrow>
 update_mem (mem, bufs(t := (l, v) # bufs t)) {} (mem', bufs')"
apply (drule update_canonical, auto)
apply (subgoal_tac "bufs'' = bufs(t := (l, v) # bufs t)", simp add: restrict_map_def, rule
    ext,
 clarsimp)
apply (erule_tac x=x in allE, rule conjI, clarsimp)
apply (case_tac "writes t", simp+, case_tac list, simp+)
```

```
apply clarsimp
done

lemma update_later: "\<lbrakk>update_mem (mem, bufs) {} (mem', bufs')\<rbrakk> \<
    Longrightarrow>
 update_mem (mem, bufs) {Write t l v} (mem', bufs'(t := (l, v) # bufs' t))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). ops = {} \<longrightarrow>
 update_mem (mem, bufs) {Write t l v} (mem', bufs'(t := (l, v) # bufs' t))" in update_mem.
     induct, auto)
apply (cut_tac ops="{Write t l v}" in no_atomic, auto)
apply (drule_tac ops="{Write t l v}" in update, auto)
apply (drule_tac ops="{Write t l v}" and t=ta in update, auto)
by (metis (hide_lams, no_types) fun_upd_twist)

lemma update_later2: "\<lbrakk>update_mem (mem, bufs) {} (mem', bufs'(t := buf)); bufs' t =
     (l, v) # buf\<rbrakk> \<Longrightarrow>
 update_mem (mem, bufs) {Write t l v} (mem', bufs')"
by (smt fun_upd_idem_iff fun_upd_upd list.inject update_later)

lemma update_past: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs'); t \<notin>
    get_thread ` ops\<rbrakk> \<Longrightarrow>
 update_mem (mem, bufs(t := b @ bufs t)) ops (mem', bufs'(t := b @ bufs' t))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). t \<notin> get_thread ` ops \<
    longrightarrow>
 update_mem (mem, bufs(t := b @ bufs t)) ops (mem', bufs'(t := b @ bufs' t))"
 in update_mem.induct, auto)
apply (rule no_atomic, auto)
apply (rule_tac x="[]" in exI, simp)
apply (subgoal_tac "\<forall>a b. Write t a b \<notin> opsa", subgoal_tac "\<exists>up.
    bufs' t = up @ bufs t
 \<and> set up = {(l, v). Write t l v \<in> opsa} \<and> distinct up", clarify, simp, metis
     , auto)
apply (metis get_thread.simps(2) imageI)
apply (drule_tac bufs="bufs(t := b @ bufs t)" in update, auto)
apply (drule_tac bufs="bufs(t := b @ bufs t)" and t=ta in update, auto)
apply (simp add: fun_upd_twist)
done

lemma update_past2: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs');
```

```
 \<And>l v. Write t l v \<notin> ops; \<And>l v v'. ARW t l v v' \<notin> ops\<rbrakk> \<
     Longrightarrow>
 update_mem (mem, bufs(t := b @ bufs t)) ops (mem', bufs'(t := b @ bufs' t))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). (\<forall>l v. Write t l v \<
    notin> ops) \<and>
 (\<forall>l v v'. ARW t l v v' \<notin> ops) \<longrightarrow> update_mem (mem, bufs(t :=
     b @ bufs t)) ops (mem', bufs'(t := b @ bufs' t))"
 in update_mem.induct, auto)
apply (rule no_atomic, auto)
apply (subgoal_tac "\<exists>up. bufs' t = up @ bufs t \<and> set up = {(l, v). Write t l v
     \<in> opsa} \<and>
 distinct up", simp)
apply (metis (full_types))
apply (drule_tac bufs="bufs(t := b @ bufs t)" in update, auto)
apply (drule_tac bufs="bufs(t := b @ bufs t)" and t=ta in update, auto simp add:
    fun_upd_twist)
done

lemma process_buffer: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs'); bufs' t = a @ b\<
    rbrakk> \<Longrightarrow>
 update_mem (mem, bufs) ops (mem' ++ map_of b, bufs'(t := a))"
apply (induct b arbitrary: mem' bufs' rule: rev_induct, auto simp add: map_upd_triv)
apply (drule_tac t=t in update, auto)
apply force
done

end

(* Sequential consistency memory model *)
locale SC = fixes undef::'val begin

abbreviation "free_set mem \<equiv> UNIV - dom mem"
abbreviation "can_read mem t l \<equiv> {v. mem l = Some v}"
inductive update_mem where
no_atomic [intro]: "\<lbrakk>\<And>t l v v'. ARW t l v v' \<notin> ops; \<And>l. (\<forall>
    t v. Write t l v \<notin> ops) \<Longrightarrow> writes l = None;
 \<And>t l v. Write t l v \<in> ops \<Longrightarrow> \<exists>t v. Write t l v \<in> ops
     \<and> writes l = Some v; finite ops\<rbrakk> \<Longrightarrow>
 update_mem mem ops (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> ops then Some undef else None) ++ writes)" |
```

```
atomic [intro!]: "update_mem mem {ARW t l v v'} (mem(l \<mapsto> v'))"
abbreviation "start_mem \<equiv> empty"

lemma stays_not_free: "\<lbrakk>update_mem mem ops mem'; l \<notin> free_set mem; \<forall>
    t. Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
by (induct rule: update_mem.induct, auto split: if_splits)

lemma alloc_not_free: "\<lbrakk>update_mem mem ops mem'; Alloc t l \<in> ops; Free t l \<
    notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
by (induct rule: update_mem.induct, auto split: if_splits)

end

sublocale SC \<subseteq> memory_model free_set can_read update_mem start_mem
by (unfold_locales, metis alloc_not_free, metis stays_not_free)

context SC begin

lemma update_none [intro!, simp]: "update_mem C {} C"
by (cut_tac no_atomic, auto simp add: restrict_map_def)

lemma update_none_only [simp, dest!]: "update_mem mem {} mem' \<Longrightarrow> mem' = mem"
by (erule update_mem.cases, auto simp add: restrict_map_def map_add_def)

lemma update_one_write [intro!, simp]: "mem' = mem(l \<mapsto> v) \<Longrightarrow>
 update_mem mem {Write t l v} mem'"
by (cut_tac ops="{Write t l v}" and writes="[l \<mapsto> v]" in no_atomic, auto simp add:
    restrict_map_def)

lemma update_one_writeD [dest!, simp]: "update_mem mem {Write t l v} mem' \<Longrightarrow>
     mem' = mem(l \<mapsto> v)"
apply (erule update_mem.cases, auto intro!: ext simp add: map_add_def split: option.splits)
apply metis+
done

lemma update_one_read [intro, simp]: "update_mem mem {Read t l v} mem"
by (cut_tac ops="{Read t l v}" in no_atomic, auto simp add: restrict_map_def)
```

```
lemma update_one_readD [dest!, simp]: "update_mem mem {Read t l v} mem' \<Longrightarrow>
    mem' = mem"
by (erule update_mem.cases, auto intro!: ext simp add: restrict_map_def map_add_def)

lemma update_one_alloc [intro!, simp]: "mem' = mem(l \<mapsto> undef) \<Longrightarrow>
 update_mem mem {Alloc t l} mem'"
apply (cut_tac ops="{Alloc t l}" and mem=mem in no_atomic, auto simp add: restrict_map_def)
apply (subgoal_tac "mem ++ (\<lambda>la. if la = l then Some undef else None) = mem(l \<
    mapsto> undef)", simp,
 auto intro!: ext simp add: map_add_def)
done

lemma update_one_allocD [dest!]: "update_mem mem {Alloc t l} mem' \<Longrightarrow> mem' =
    mem(l \<mapsto> undef)"
by (erule update_mem.cases, auto simp add: restrict_map_def map_add_def)

lemma update_frees [intro!, simp]: "\<lbrakk>mem' = mem |` (UNIV - S); finite S\<rbrakk> \<
    Longrightarrow>
 update_mem mem (Free t ` S) mem'"
apply (cut_tac ops="Free t ` S" and mem=mem in no_atomic, auto)
apply (rule subst, rule_tac f="update_mem mem (Free t ` S)" in arg_cong, simp_all)
apply (auto intro!: ext simp add: map_add_def restrict_map_def, force)
done

lemma update_freesD [dest!, simp]: "update_mem mem (Free t ` S) mem' \<Longrightarrow>
 mem' = mem |` (UNIV - S)"
apply (erule update_mem.cases, auto intro!: ext simp add: map_add_def restrict_map_def
 split: option.splits)
apply (metis image_eqI)
apply (metis access.distinct(13) image_iff option.distinct(1))
apply (metis image_eqI)
by (metis access.distinct(13) image_iff option.distinct(1))

lemma update_ARW [intro!]: "mem' = mem(l \<mapsto> v') \<Longrightarrow> update_mem mem {
    ARW t l v v'} mem'"
by clarsimp

lemma update_ARWD [dest!]: "update_mem mem {ARW t l v v'} mem' \<Longrightarrow> mem' = mem
    (l \<mapsto> v')"
by (erule update_mem.cases, auto)
```

```
lemma update_past: "\<lbrakk>update_mem mem ops mem'; \<And>t v. Write t l v \<notin> ops;
 \<And>t v v'. ARW t l v v' \<notin> ops; \<And>t. Free t l \<notin> ops; \<And>t. Alloc t
     l \<notin> ops\<rbrakk> \<Longrightarrow>
 update_mem (mem(l := v)) ops (mem'(l := v))"
apply (induct rule: update_mem.induct, auto)
apply (cut_tac ops=ops and mem="mem(l := v)" and writes=writes in no_atomic, auto)
apply (rule_tac f1="update_mem (mem(l := v))" in arg_cong2 [THEN subst],
 auto intro!: ext simp add: map_add_def restrict_map_def split: option.splits)
by (metis atomic fun_upd_twist)

end

locale undef = fixes undef::'val
sublocale undef \<subseteq> TSO: TSO .
sublocale undef \<subseteq> SC: SC .

locale PSO = fixes undef::'val begin

(* PSO memory model *)
abbreviation "free_set mem \<equiv> UNIV - dom (fst mem)"
definition "can_read mem t l \<equiv> case snd mem t l of v # buf \<Rightarrow> {v}
 | [] \<Rightarrow> {v. fst mem l = Some v}"
inductive update_mem where
no_atomic [intro]: "\<lbrakk>\<And>t l v v'. ARW t l v v' \<notin> ops; \<And>t l. \<exists
    >up. bufs' t l = up @ bufs t l \<and>
 set up = {v. Write t l v \<in> ops} \<and> distinct up\<rbrakk> \<Longrightarrow>
 update_mem (mem, bufs) ops (mem |` (UNIV - {l. \<exists>t. Free t l \<in> ops}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> ops then Some undef else None), bufs')" |
update [intro]: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs'); bufs' t l = buf @ [v]\<
    rbrakk> \<Longrightarrow>
 update_mem (mem, bufs) ops (mem'(l \<mapsto> v), bufs'(t := (bufs' t)(l := buf)))" |
atomic [intro!]: "bufs t l = [] \<Longrightarrow>
 update_mem (mem, bufs) {ARW t l v v'} (mem(l \<mapsto> v'), bufs)"
abbreviation "start_mem \<equiv> (empty, \<lambda>t l. [])"

lemma alloc_not_free: "\<lbrakk>update_mem mem ops mem'; Alloc t l \<in> ops; \<forall>t.
    Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
by (induct rule: update_mem.induct, auto split: if_splits)
```

```
lemma stays_not_free: "\<lbrakk>update_mem mem ops mem'; l \<notin> free_set mem; \<forall>
    t. Free t l \<notin> ops\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
by (induct rule: update_mem.induct, auto split: if_splits)

end

sublocale PSO \<subseteq> memory_model free_set can_read update_mem start_mem
by (unfold_locales, metis alloc_not_free, metis stays_not_free)

context PSO begin

lemma update_none [intro!, simp]: "update_mem C {} C"
by (cases C, cut_tac ops="{}" and mem=a and bufs=b in no_atomic, auto simp add:
    restrict_map_def)

lemma process_buffer: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs'); bufs' t l = a @ v
     # b\<rbrakk> \<Longrightarrow>
 update_mem (mem, bufs) ops (mem'(l \<mapsto> v), bufs'(t := (bufs' t)(l := a)))"
apply (induct b arbitrary: mem' bufs' v rule: rev_induct, auto simp add: map_upd_triv)
apply (drule_tac t=t and l=l in update, simp, force)
apply force
done

lemma update_later: "\<lbrakk>update_mem (mem, bufs) {} (mem', bufs')\<rbrakk> \<
    Longrightarrow>
 update_mem (mem, bufs) {Write t l v} (mem', bufs'(t := (bufs' t)(l := v # bufs' t l)))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). ops = {} \<longrightarrow>
 update_mem (mem, bufs) {Write t l v} (mem', bufs'(t := (bufs' t)(l := v # bufs' t l)))"
 in update_mem.induct, auto)
apply (cut_tac ops="{Write t l v}" in no_atomic, auto)
apply (drule_tac ops="{Write t l v}" in update, auto)
apply (drule_tac ops="{Write t l v}" and t=ta in update, auto simp add: fun_upd_twist)
apply (drule_tac ops="{Write t l v}" and l=la in update, auto simp add: fun_upd_twist)
apply (drule_tac ops="{Write t l v}" and t=ta in update, auto simp add: fun_upd_twist)
done

lemma update_later2: "\<lbrakk>update_mem (mem, bufs) {} (mem', bufs'(t := (bufs' t)(l :=
    buf)));
```

```
 bufs' t l = v # buf\<rbrakk> \<Longrightarrow> update_mem (mem, bufs) {Write t l v} (mem',
      bufs')"
by (smt fun_upd_idem_iff fun_upd_upd list.inject update_later)

lemma update_past2: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs');
 \<And>v. Write t l v \<notin> ops; \<And>l v v'. ARW t l v v' \<notin> ops\<rbrakk> \<
     Longrightarrow>
 update_mem (mem, bufs(t := (bufs t)(l := b @ bufs t l))) ops
               (mem', bufs'(t := (bufs' t)(l := b @ bufs' t l)))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). (\<forall>v. Write t l v \<
    notin> ops) \<and>
 (\<forall>l v v'. ARW t l v v' \<notin> ops) \<longrightarrow> update_mem (mem, bufs(t :=
     (bufs t)(l := b @ bufs t l))) ops
               (mem', bufs'(t := (bufs' t)(l := b @ bufs' t l)))" in update_mem.induct,
                   auto)
apply (rule no_atomic, auto)
apply (subgoal_tac "\<exists>up. bufs' t l = up @ bufs t l \<and> set up = {v. Write t l v
    \<in> opsa} \<and>
 distinct up", simp)
apply (metis (full_types))
apply (drule_tac bufs="bufs(t := (bufs t)(l := b @ bufs t l))" in update, auto)
apply (drule_tac bufs="bufs(t := (bufs t)(l := b @ bufs t l))" and t=ta in update,
 auto simp add: fun_upd_twist)
apply (drule_tac bufs="bufs(t := (bufs t)(l := b @ bufs t l))" in update,
 auto simp add: fun_upd_twist)
apply (drule_tac bufs="bufs(t := (bufs t)(l := b @ bufs t l))" and t=ta in update,
 auto simp add: fun_upd_twist)
done

lemma update_write: "\<lbrakk>update_mem (mem, bufs) {Write t l v} (mem', bufs')\<rbrakk>
    \<Longrightarrow>
 update_mem (mem, bufs(t := (bufs t)(l := v # bufs t l))) {} (mem', bufs')"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). ops = {Write t l v} \<
    longrightarrow>
 update_mem (mem, bufs(t := (bufs t)(l := v # bufs t l))) {} (mem', bufs')"
 in update_mem.induct, auto)
apply (cut_tac ops="{}" and bufs="bufs(t := (bufs t)(l := v # bufs t l))" and
 bufs'=bufs' in no_atomic, simp_all)
apply (subgoal_tac "\<exists>up. bufs' ta la = up @ bufs ta la \<and> set up = {va. va = v
    \<and> ta = t \<and> la = l} \<and>
```

```
 distinct up", clarify, auto)
apply (case_tac up, auto, case_tac list, auto)
done

lemma update_trans_rev: "\<lbrakk>update_mem (mem', bufs') {} (mem'', bufs'');
 update_mem (mem, bufs) ops (mem', bufs')\<rbrakk> \<Longrightarrow> update_mem (mem, bufs)
      ops (mem'', bufs'')"
apply (drule_tac P="\<lambda>(mem', bufs') ops' (mem'', bufs''). ops' = {} \<and>
    update_mem (mem, bufs) ops (mem', bufs') \<longrightarrow>
 update_mem (mem, bufs) ops (mem'', bufs'')" in update_mem.induct, auto simp add:
     restrict_map_def)
apply (subgoal_tac "bufs'a = bufsa", auto)
done

end

end

(* step_rel.thy *)
(* William Mansky *)
(* The general theory of tCFG step relations and simulations on them. *)

theory step_rel
imports memory_model trans_sim
begin

print_locale memory_model
locale step_rel = memory_model where update_mem=
  "update_mem::'memory \<Rightarrow> ('thread, 'loc, 'val) access set \<Rightarrow> 'memory
       \<Rightarrow> bool" for update_mem +
  fixes step_rel::"'thread \<Rightarrow> ('node, 'edge_type, 'instr) flowgraph \<Rightarrow
      > 'memory \<Rightarrow> 'config \<Rightarrow>
  ('thread, 'loc, 'val) access set \<Rightarrow> 'config \<Rightarrow> bool"
  and start_state::"('thread \<rightharpoonup> ('node, 'edge_type, 'instr) flowgraph) \<
      Rightarrow> ('thread \<rightharpoonup> 'config) \<Rightarrow> 'memory \<Rightarrow>
      bool"
  and get_point::"'config \<Rightarrow> 'node"
  and instr_edges::"'instr \<Rightarrow> ('edge_type \<Rightarrow> nat) set"
  and Seq::'edge_type
  assumes start_points [simp]: "start_state CFGs S m \<Longrightarrow> ((\<lambda>C. Some (
      get_point C)) o_m S) = start_points CFGs"
```

```
      and step_along_edge: "\<lbrakk>step_rel t G m C ops C'; is_flowgraph G Seq
          instr_edges; get_point C \<in> Nodes G\<rbrakk> \<Longrightarrow>
    \<exists>ty. (get_point C, get_point C', ty) \<in> Edges (G::('node, 'edge_type, 'instr
        ) flowgraph)"
begin

lemma step_safe: "\<lbrakk>step_rel t G m C ops C'; is_flowgraph G Seq instr_edges;
    get_point C \<in> Nodes G\<rbrakk> \<Longrightarrow>
    get_point C' \<in> Nodes G"
by (drule step_along_edge, simp+, simp add: is_flowgraph_def flowgraph_def
    pointed_graph_def)

abbreviation "well_formed G \<equiv> is_flowgraph G Seq instr_edges"

inductive one_step where
step_single [intro!]: "\<lbrakk>step_rel t G mem C ops C'; update_mem mem ops mem'\<rbrakk>
     \<Longrightarrow>
  one_step t G (C, mem) (C', mem')"

lemma one_step_safe: "\<lbrakk>one_step t G (C, mem) (C', mem'); well_formed G; get_point C
     \<in> Nodes G\<rbrakk> \<Longrightarrow>
  get_point C' \<in> Nodes G"
by (erule one_step.cases, rule step_safe, simp+)

lemma one_step_along_edge: "\<lbrakk>one_step t G (C, m) (C', m'); is_flowgraph G Seq
    instr_edges;
  get_point C \<in> Nodes G\<rbrakk> \<Longrightarrow> \<exists>ty. (get_point C, get_point
       C', ty) \<in> Edges G"
by (erule one_step.cases, rule step_along_edge, simp+)

inductive conc_step where
step_thread [intro]: "\<lbrakk>CFGs t = Some G; states t = Some C; step_rel t G mem C ops C
    ';
  update_mem mem ops mem'\<rbrakk> \<Longrightarrow> conc_step CFGs (states, mem) (states(t
       \<mapsto> C'), mem')"

abbreviation "get_points (S::'thread \<rightharpoonup> 'config) \<equiv> (\<lambda>C. Some
    (get_point C)) o_m S"
```

```
lemma one_step_conc_step: "\<lbrakk>one_step t G (C, m) (C', m'); CFGs t = Some G; states t
     = Some C\<rbrakk> \<Longrightarrow>
  conc_step CFGs (states, m) (states(t \<mapsto> C'), m')"
by (erule one_step.cases, rule step_thread, auto)

lemma conc_step_safe: "\<lbrakk>conc_step CFGs (states, mem) (states', mem'); tCFG CFGs
    instr_edges Seq;
  safe_points CFGs (get_points states)\<rbrakk> \<Longrightarrow>
  safe_points CFGs (get_points states')"
apply (erule conc_step.cases, clarsimp simp add: safe_points_def)
apply (case_tac "t = ta", clarsimp)
apply (rule step_safe, simp+)
apply (erule tCFG.CFGs, simp)
apply force
apply (force simp add: map_comp_def)
done

lemma conc_steps_safe: "\<lbrakk>(conc_step CFGs)^** (states, mem) (states', mem'); tCFG
    CFGs instr_edges Seq;
  safe_points CFGs (get_points states)\<rbrakk> \<Longrightarrow>
  safe_points CFGs (get_points states')"
by (drule_tac P="\<lambda>(states, mem) (states', mem'). safe_points CFGs (get_points
    states) \<longrightarrow>
  safe_points CFGs (get_points states')" in rtranclp.induct, auto intro: conc_step_safe)

definition "run_prog CFGs C \<equiv> \<exists>C0 mem0. start_state CFGs C0 mem0 \<and> (
    conc_step CFGs)^** (C0, mem0) C"

lemma run_progI [intro]: "\<lbrakk>start_state CFGs C0 mem0; (conc_step CFGs)^** (C0, mem0)
     C\<rbrakk> \<Longrightarrow>
  run_prog CFGs C"
by (force simp add: run_prog_def)

lemma run_prog_conc_step [intro]: "\<lbrakk>run_prog CFGs C; conc_step CFGs C C'\<rbrakk>
    \<Longrightarrow> run_prog CFGs C'"
by (force simp add: run_prog_def)

lemma run_prog_conc_steps [intro]: "\<lbrakk>run_prog CFGs C; (conc_step CFGs)^** C C'\<
    rbrakk> \<Longrightarrow> run_prog CFGs C'"
by (force simp add: run_prog_def)
```

```
lemma run_prog_induct [elim]: "\<lbrakk>run_prog CFGs C; \<And>C0 mem0. start_state CFGs C0
     mem0 \<Longrightarrow> P (C0, mem0);
 \<And>C C'. run_prog CFGs C \<Longrightarrow> P C \<Longrightarrow> conc_step CFGs C C' \<
     Longrightarrow> P C'\<rbrakk> \<Longrightarrow> P C"
apply (clarsimp simp add: run_prog_def)
apply (drule_tac P="\<lambda>C C'. start_state CFGs (fst C) (snd C) \<longrightarrow> P C'"
     in rtranclp.induct, auto)
by (metis (mono_tags))

lemma run_prog_one_step: "\<lbrakk>run_prog CFGs (S, mem); S t = Some s; CFGs t = Some G;
 one_step t G (s, mem) (s', mem')\<rbrakk> \<Longrightarrow> run_prog CFGs (S(t \<mapsto> s
     '), mem')"
by (erule run_prog_conc_step, erule one_step.cases, auto)

lemma run_prog_one_steps: "\<lbrakk>run_prog CFGs (S, mem); S t = Some s; CFGs t = Some G;
 (one_step t G)^** (s, mem) (s', mem')\<rbrakk> \<Longrightarrow> run_prog CFGs (S(t \<
     mapsto> s'), mem')"
apply (drule_tac P="\<lambda>(s, mem) (s', mem'). run_prog CFGs (S, mem) \<and> S t = Some
    s \<and> CFGs t = Some G \<longrightarrow>
  run_prog CFGs (S(t \<mapsto> s'), mem')" in rtranclp.induct, auto simp add: map_upd_triv)
apply (drule_tac mem=ba in run_prog_one_step, simp+, force, simp+)
done

lemma run_prog_step: "\<lbrakk>run_prog CFGs (S, mem); S t = Some s; CFGs t = Some G;
 step_rel t G mem s ops s'; update_mem mem ops mem'\<rbrakk> \<Longrightarrow> run_prog
     CFGs (S(t \<mapsto> s'), mem')"
by (erule run_prog_one_step, auto)

lemma run_prog_safe: "\<lbrakk>run_prog CFGs (S, mem); tCFG CFGs instr_edges Seq\<rbrakk>
    \<Longrightarrow>
  safe_points CFGs (get_points S)"
apply (clarsimp simp add: run_prog_def, rule conc_steps_safe, simp+)
apply (erule tCFG.safe_start)
done

(* paths *)
lemma step_increment_path: "\<lbrakk>step_rel t G m C a C'; tCFG CFGs instr_edges Seq; CFGs
     t = Some G;
```

```
  l \<in> tCFG.Paths CFGs ps; p = get_point C; p' = get_point C'; ps t = Some p'; p \<in>
      Nodes G\<rbrakk> \<Longrightarrow>
  [ps(t \<mapsto> p)] \<frown> l \<in> tCFG.Paths CFGs (ps(t \<mapsto> p))"
apply (rule tCFG.path_incremental_gen, simp+, clarsimp)
apply (frule step_along_edge)
apply (erule tCFG.CFGs, simp+)
done

lemma step_increment_rpath: "\<lbrakk>step_rel t G m C a C'; tCFG CFGs instr_edges Seq;
    CFGs t = Some G;
  l \<in> tCFG.RPaths CFGs ps; p0 = get_point C; p = get_point C'; ps t = Some p0; p0 \<in>
       Nodes G\<rbrakk> \<Longrightarrow>
 [ps(t \<mapsto> p)] \<frown> l \<in> tCFG.RPaths CFGs (ps(t \<mapsto> p))"
apply (rule tCFG.rpath_incremental_gen, simp+, clarsimp)
apply (frule step_along_edge)
apply (erule tCFG.CFGs, simp+)
done

lemma one_step_increment_path: "\<lbrakk>one_step t G (C, m) (C', m'); tCFG CFGs
    instr_edges Seq; CFGs t = Some G;
  l \<in> tCFG.Paths CFGs ps; p = get_point C; p' = get_point C'; ps t = Some p'; p \<in>
      Nodes G\<rbrakk> \<Longrightarrow>
  [ps(t \<mapsto> p)] \<frown> l \<in> tCFG.Paths CFGs (ps(t \<mapsto> p))"
by (erule one_step.cases, rule step_increment_path, simp+)

lemma one_step_increment_rpath: "\<lbrakk>one_step t G (C, m) (C', m'); tCFG CFGs
    instr_edges Seq; CFGs t = Some G;
  l \<in> tCFG.RPaths CFGs ps; p0 = get_point C; p = get_point C'; ps t = Some p0; p0 \<in>
       Nodes G\<rbrakk> \<Longrightarrow>
 [ps(t \<mapsto> p)] \<frown> l \<in> tCFG.RPaths CFGs (ps(t \<mapsto> p))"
by (erule one_step.cases, rule step_increment_rpath, simp+)

lemma conc_step_increment_path: "\<lbrakk>conc_step CFGs C C'; tCFG CFGs instr_edges Seq;
  l \<in> tCFG.Paths CFGs (get_points (fst C')); safe_points CFGs (get_points (fst C))\<
      rbrakk> \<Longrightarrow>
 [get_points (fst C)] \<frown> l \<in> tCFG.Paths CFGs (get_points (fst C))"
apply (auto elim!: conc_step.cases)
apply (frule_tac ps="get_points (states(t \<mapsto> C'))" in step_increment_path, auto simp
     add: safe_points_def)
apply force
```

```
apply (subgoal_tac "get_points (states(t \<mapsto> C'))(t \<mapsto> get_point C) =
    get_points states", simp)
apply (rule ext, clarsimp simp add: map_comp_def)
done

lemma conc_steps_path: "\<lbrakk>(conc_step CFGs)^** (states, mem) (states', mem'); tCFG
    CFGs instr_edges Seq;
  l \<in> tCFG.Paths CFGs (get_points states'); safe_points CFGs (get_points states)\<
      rbrakk> \<Longrightarrow>
 \<exists>l'. hd (l' @ [l 0]) = get_points states \<and> l' \<frown> l \<in> tCFG.Paths
     CFGs (get_points states)"
apply (drule_tac P="\<lambda>(states, mem) (states', mem'). \<forall>l. l \<in> tCFG.Paths
    CFGs (get_points states') \<and>
  safe_points CFGs (get_points states) \<longrightarrow> (\<exists>l'. hd (l' @ [l 0]) =
      get_points states \<and>
  l' \<frown> l \<in> tCFG.Paths CFGs (get_points states))" in rtranclp.induct, auto)
apply (rule_tac x="[]" in exI, simp, rule tCFG.Path_first, simp+)
apply (drule conc_step_increment_path, simp+)
apply (drule conc_steps_safe, simp+)
by (metis append_is_Nil_conv hd_append2 i_append_assoc i_append_nth_Cons_0 not_Cons_self2)

lemma run_prog_path: "\<lbrakk>run_prog CFGs C; tCFG CFGs instr_edges Seq\<rbrakk> \<
    Longrightarrow>
  \<exists>C0 mem0. start_state CFGs C0 mem0 \<and> (\<exists>l\<in>tCFG.Paths CFGs (
      start_points CFGs). \<exists>i. l i = get_points (fst C))"
apply (clarsimp simp add: run_prog_def)
apply (cut_tac q="get_points (fst C)" in tCFG.exists_path, simp, clarsimp)
apply (case_tac C, clarsimp)
apply (drule conc_steps_path, simp+)
apply (erule tCFG.safe_start)
apply clarsimp
apply (rule conjI, force)
apply (rule_tac x="l' \<frown> l" in bexI, simp_all)
apply (rule_tac x="length l'" in exI, simp)
apply (rule tCFG.Path_first, simp+)
done

(* The step-star relation as inducing a list of intermediate states. *)
lemma conc_step_star_steps: "(conc_step CFGs)^** C C' \<Longrightarrow>
```

```
 \<exists>l. hd (l @ [C']) = C \<and> (\<forall>i<length l. conc_step CFGs (l ! i) ((l @ [C
     ']) ! Suc i))"
apply (induct rule: rtranclp_induct, auto)
apply (rule_tac x="[]" in exI, simp)
apply (rule_tac x="l @ [(a, b)]" in exI, clarsimp)
apply (rule conjI, case_tac l, simp+)
apply clarsimp
apply (case_tac "i = length l", clarsimp simp add: nth_append)
apply (erule_tac x=i in allE, auto simp add: nth_append)
done

lemma step_star_path: "\<lbrakk>hd (l' @ [C']) = C; \<forall>i<length l'. conc_step CFGs (l
    ' ! i) ((l' @ [C']) ! Suc i);
 l \<in> tCFG.Paths CFGs (get_points (fst C')); safe_points CFGs (get_points (fst C)); tCFG
      CFGs instr_edges Seq\<rbrakk> \<Longrightarrow>
 map (get_points o fst) l' \<frown> l \<in> tCFG.Paths CFGs (get_points (fst C))"
apply (induct l' arbitrary: C l, auto)
apply (case_tac l', auto)
apply (drule conc_step_increment_path, simp+)
apply (subgoal_tac "(get_points a # map (get_points \<circ> fst) list) \<frown> l \<in>
    tCFG.Paths CFGs (get_points a)")
apply (erule_tac x=0 in allE, clarsimp)
apply (drule conc_step_increment_path, simp+)
apply (subgoal_tac "\<forall>i<Suc (length list). conc_step CFGs (((a, b) # list) ! i) ((
    list @ [C']) ! i)")
apply (subgoal_tac "safe_points CFGs (get_points a)", force)
apply (erule_tac x=0 in allE, clarsimp, rule conc_step_safe, auto)
done

lemma run_prog_steps: "\<lbrakk>run_prog CFGs C; tCFG CFGs instr_edges Seq\<rbrakk> \<
    Longrightarrow>
 \<exists>l C0 mem0. start_state CFGs C0 mem0 \<and>
 (\<forall>i<length l. conc_step CFGs (l ! i) ((l @ [C]) ! Suc i)) \<and>
 (\<exists>l'\<in>tCFG.Paths CFGs (get_points (fst C)). map (get_points o fst) l \<frown> l
     ' \<in> tCFG.Paths CFGs (start_points CFGs))"
apply (clarsimp simp add: run_prog_def)
apply (drule conc_step_star_steps, clarsimp)
apply (cut_tac q="get_points (fst C)" in tCFG.exists_path, simp+, clarsimp)
apply (frule step_star_path, simp+)
apply (erule tCFG.safe_start, simp+)
```

```
apply (rule conjI, force+)
done

lemma conc_step_star_path2: "\<lbrakk>(conc_step CFGs)^** C C'; l \<in> tCFG.Paths CFGs (
    get_points (fst C'));
 safe_points CFGs (get_points (fst C)); tCFG CFGs instr_edges Seq\<rbrakk> \<Longrightarrow
    >
 \<exists>l'. hd (l' @ [l 0]) = get_points (fst C) \<and> l' \<frown> l \<in> tCFG.Paths
    CFGs (get_points (fst C)) \<and>
 (\<forall>i<length l'. \<exists>C''. l' ! i = get_points (fst C'') \<and> (conc_step CFGs)
    ^** C C'' \<and> (conc_step CFGs)^** C'' C')"
apply (induct arbitrary: l rule: rtranclp_induct, auto)
apply (rule_tac x="[]" in exI, simp)
apply (rule tCFG.Path_first, simp+)
apply (frule conc_step_increment_path, simp+)
apply (case_tac C, rule conc_steps_safe, simp+)
apply (subgoal_tac "\<exists>l'. hd (l' @ [get_points a]) = get_points (fst C) \<and> l' \<
    frown> [get_points a] \<frown> l \<in>
 tCFG.Paths CFGs (get_points (fst C)) \<and> (\<forall>i<length l'. \<exists>aa. l' ! i =
    get_points aa \<and> (\<exists>ba.
 (conc_step CFGs)^** C (aa, ba) \<and> (conc_step CFGs)^** (aa, ba) (a, b)))", clarsimp)
apply (rule_tac x="l' @ [get_points a]" in exI, clarsimp)
apply (rule conjI, case_tac l', simp+)
apply clarsimp
apply (case_tac "i = length l'", clarsimp)
apply (rule_tac x=a in exI, simp, rule_tac x=b in exI, simp)
apply (erule_tac x=i in allE, clarsimp)
apply (rule_tac x=aaa in exI, simp add: nth_append)
apply (rule_tac x=baa in exI, simp)
by (metis i_append_nth_Cons_0)

lemma run_prog_path2: "\<lbrakk>run_prog CFGs C; tCFG CFGs instr_edges Seq\<rbrakk> \<
    Longrightarrow>
  \<exists>C0 mem0. start_state CFGs C0 mem0 \<and> (\<exists>l\<in>tCFG.Paths CFGs (
      start_points CFGs).
    \<exists>i. l i = get_points (fst C) \<and> (\<forall>j<i. \<exists>C'. run_prog CFGs C
        ' \<and> l j = get_points (fst C')))"
apply (clarsimp simp add: run_prog_def)
apply (cut_tac q="get_points (fst C)" in tCFG.exists_path, simp+, clarsimp)
apply (drule conc_step_star_path2, simp+)
```

```
apply (erule tCFG.safe_start, simp+)
apply clarsimp
apply (rule conjI, force)
apply (rule_tac x="l' \<frown> l" in bexI, simp_all)
apply (rule_tac x="length l'" in exI, clarsimp)
apply (rule conjI, rule tCFG.Path_first, simp+)
by smt

lemma run_prog_rpath: "\<lbrakk>run_prog CFGs C; tCFG CFGs instr_edges Seq\<rbrakk> \<
    Longrightarrow>
  \<exists>l. l \<in> tCFG.RPaths CFGs (get_points (fst C))"
by (drule run_prog_path, simp, clarsimp, drule_tac i=i in tCFG.reverse_path, simp, force)


(* simulation relations and lifting *)
definition "lift_reach_sim_rel sim_rel CFGs CFGs' t C C' \<equiv>
 run_prog CFGs C \<and> run_prog CFGs' C' \<and> (case (C, C') of ((states, mem), (states',
      mem')) \<Rightarrow>
 (\<exists>s s' G G'. states t = Some s \<and> states' t = Some s' \<and> CFGs t = Some G
     \<and> CFGs' t = Some G' \<and>
 sim_rel G G' (s, mem) (s', mem')) \<and> (\<forall>t'. t' \<noteq> t \<longrightarrow>
     states t' = states' t'))"

definition "add_reach CFGs CFGs' t sim_rel G G' c c' \<equiv> \<exists>S S'. S t = Some (
    fst c) \<and>
 run_prog CFGs (S, snd c) \<and> S' t = Some (fst c') \<and> run_prog CFGs' (S', snd c') \<
     and> sim_rel G G' c c'"

lemma add_reach_sim_rel: "\<lbrakk>sim_rel G G' c c'; \<exists>S S'. S t = Some (fst c) \<
    and> run_prog CFGs (S, snd c) \<and>
  S' t = Some (fst c') \<and> run_prog CFGs' (S', snd c')\<rbrakk> \<Longrightarrow>
 add_reach CFGs CFGs' t sim_rel G G' c c'"
by (simp add: add_reach_def)

end

locale sim_base = step_rel where start_state="start_state::('thread \<rightharpoonup> ('
    node, 'edge_type, 'instr) flowgraph) \<Rightarrow>
  ('thread \<rightharpoonup> 'config) \<Rightarrow> 'memory \<Rightarrow> bool" +
```

```
  tCFG?: tCFG where CFGs="CFGs::('thread \<rightharpoonup> ('node, 'edge_type, 'instr)
      flowgraph)" +
  tCFG': tCFG where CFGs="CFGs'::('thread \<rightharpoonup> ('node, 'edge_type, 'instr)
      flowgraph)"
  for start_state CFGs CFGs' +
  assumes step_read_ops: "\<lbrakk>step_rel t G mem C ops C'; CFGs t = Some G;
    \<forall>l\<in>get_ptrs ops. can_read mem t l = can_read mem' t l; free_set mem =
        free_set mem'\<rbrakk> \<Longrightarrow>
    step_rel t G mem' C ops C'"
      and ops_thread: "\<lbrakk>step_rel t G mem state ops state'; CFGs t = Some G; a \<in>
           ops\<rbrakk> \<Longrightarrow> get_thread a = t"
begin

lemma sim_by_reachable_thread_mem_obs [rule_format]:
"\<lbrakk>tCFG_sim (add_reach CFGs' CFGs t sim_rel G' G) (op =) G' G (one_step t) obs (
    get_mem o snd);
 CFGs t = Some G; CFGs' t = Some G'; \<forall>t'. t' \<noteq> t \<longrightarrow> CFGs t' =
     CFGs' t'; \<forall>mem s mem' s'.
 sim_rel G' G (s, mem) (s', mem') \<longrightarrow> (\<forall>S. run_prog CFGs' (S, mem) \<
     longrightarrow> (free_set mem = free_set mem' \<and>
 (\<forall>t' ops s1 s2. run_prog CFGs' (S, mem) \<and> run_prog CFGs (S(t \<mapsto> s'),
     mem') \<and> S t = Some s \<and>
 S t' = Some s1 \<and> t' \<noteq> t \<and> t' \<in> dom CFGs \<longrightarrow> (step_rel t
     ' (the (CFGs t')) mem s1 ops s2  \<longrightarrow>
 (\<forall>l\<in>get_ptrs ops. can_read mem t' l = can_read mem' t' l)) \<and>
 (step_rel t' (the (CFGs t')) mem' s1 ops s2  \<longrightarrow> (\<forall>mem2. update_mem
     mem ops mem2 \<and> t \<notin> get_thread ` ops \<longrightarrow>
 (\<exists>mem2'. update_mem mem' ops mem2' \<and> (\<forall>l\<in>obs. get_mem mem2' l =
     get_mem mem2 l) \<and> sim_rel G' G (s, mem2) (s', mem2'))))))\<rbrakk> \<
     Longrightarrow>
 tCFG_sim (lift_reach_sim_rel sim_rel CFGs' CFGs t) (op =) CFGs' CFGs conc_step obs (
     get_mem o snd)"
apply (simp add: tCFG_sim_def, unfold_locales, clarsimp simp add: trsys_of_tCFG_def)
apply (rule conc_step.cases, simp+, clarsimp simp add: lift_reach_sim_rel_def)
apply (rule conjI, clarsimp)
apply (drule_tac sa="(C, mem)" and sb="(s', ba)" in simulation.simulation)
apply (clarsimp simp add: add_reach_def)
apply (rule_tac x=states in exI, simp, rule_tac x=aa in exI, simp)
apply (clarsimp simp add: trsys_of_tCFG_def)
apply (rule conjI, erule step_single, simp+)
```

```
apply (clarsimp simp add: trsys_of_tCFG_def add_reach_def)
apply (erule one_step.cases, clarsimp)
apply ((rule exI)+, rule context_conjI, rule_tac t=t in step_thread, simp add: dom_def,
    simp+)
apply (rule conjI, erule run_prog_step, simp+)
apply (erule run_prog_step, simp+)
apply clarsimp
apply (subgoal_tac "aa = states(t \<mapsto> s')", thin_tac "\<forall>t'. t' \<noteq> t \<
    longrightarrow> states t' = aa t'",
 clarsimp)
apply (erule allE, erule allE, erule allE, erule allE, erule impE, simp)
apply (erule_tac x=states in allE, clarsimp)
apply (erule_tac x=ta in allE, erule_tac x=ops in allE, erule_tac x=C in allE, simp, erule
    impE,
  simp add: dom_def)
apply (erule_tac x=C' in allE, clarsimp)
apply (drule_tac mem=mem and mem'=ba in step_read_ops, simp+)
apply (erule_tac x=mem' in allE, clarsimp)
apply (erule impE, clarsimp)
apply (cut_tac t=t in ops_thread, simp_all, simp+)
apply clarsimp
apply (rule exI, rule_tac x=mem2' in exI, rule context_conjI, rule_tac t=ta in step_thread,
 simp add: dom_def, simp+)
apply (rule conjI, erule run_prog_conc_step, simp+)
apply (rule conjI, erule run_prog_conc_step, simp+)
apply (clarsimp intro!: ext simp add: restrict_map_def)
apply (rule ext, simp)
done

(* Slightly reorganized other-threads hypothesis. *)
lemma sim_by_reachable_thread_obs [rule_format]:
"\<lbrakk>tCFG_sim (add_reach CFGs' CFGs t sim_rel G' G) (op =) G' G (one_step t) obs (
    get_mem o snd);
 CFGs t = Some G; CFGs' t = Some G'; \<forall>t'. t' \<noteq> t \<longrightarrow> CFGs t' =
      CFGs' t';
 \<forall>mem s mem' s'. sim_rel G' G (s, mem) (s', mem') \<longrightarrow> (free_set mem =
      free_set mem' \<and>
 (\<forall>t' ops s1 s2 S. run_prog CFGs' (S, mem) \<and> run_prog CFGs (S(t \<mapsto> s'),
      mem') \<and>
```

```
 S t = Some s \<and> S t' = Some s1 \<and> t' \<noteq> t \<and> t' \<in> dom CFGs \<
     longrightarrow>
 (step_rel t' (the (CFGs t')) mem s1 ops s2  \<longrightarrow> (\<forall>l\<in>get_ptrs ops
     . can_read mem t' l = can_read mem' t' l)) \<and>
 (step_rel t' (the (CFGs t')) mem' s1 ops s2  \<longrightarrow> (\<forall>mem2. update_mem
     mem ops mem2 \<and> t \<notin> get_thread ` ops \<longrightarrow>
 (\<exists>mem2'. update_mem mem' ops mem2' \<and> (\<forall>l\<in>obs. get_mem mem2' l =
     get_mem mem2 l) \<and> sim_rel G' G (s, mem2) (s', mem2'))))))\<rbrakk> \<
     Longrightarrow>
 tCFG_sim (lift_reach_sim_rel sim_rel CFGs' CFGs t) (op =) CFGs' CFGs conc_step obs (
     get_mem o snd)"
apply (rule sim_by_reachable_thread_mem_obs, simp+)
apply clarsimp
apply (erule_tac x=mem in allE, erule_tac x=s in allE, erule_tac x=mem' in allE,
  erule_tac x=s' in allE, erule impE, assumption, clarsimp)
apply (erule_tac x=t' in allE, erule_tac x=ops in allE, erule_tac x=s1 in allE, erule impE)
apply (rule_tac x=S in exI, force)
apply (erule_tac x=s2 in allE, force)
done

lemma sim_by_reachable_thread [rule_format]: "\<lbrakk>tCFG_sim (add_reach CFGs' CFGs t
    sim_rel G' G) (op =)
 G' G (one_step t) UNIV (get_mem o snd); CFGs t = Some G; CFGs' t = Some G';
 \<forall>t'. t' \<noteq> t \<longrightarrow> CFGs t' = CFGs' t'; \<forall>mem s mem' s'.
     sim_rel G' G (s, mem) (s', mem') \<longrightarrow>
 (free_set mem = free_set mem' \<and> (\<forall>t'. t' \<noteq> t \<longrightarrow>
     can_read mem t' = can_read mem' t') \<and>
 (\<forall>ops mem2. update_mem mem ops mem2 \<and> t \<notin> get_thread ` ops \<
     longrightarrow>
 (\<exists>mem2'. update_mem mem' ops mem2' \<and> get_mem mem2' = get_mem mem2 \<and>
     sim_rel G' G (s, mem2) (s', mem2'))))\<rbrakk> \<Longrightarrow>
 tCFG_sim (lift_reach_sim_rel sim_rel CFGs' CFGs t) (op =) CFGs' CFGs conc_step UNIV (
     get_mem o snd)"
apply (erule sim_by_reachable_thread_obs, auto simp add: fun_upd_def)
by (smt UNIV_I UNIV_eq_I domD domI)

lemma sim_no_mem [rule_format]: "\<lbrakk>tCFG_sim (add_reach CFGs' CFGs t sim_rel G' G) (
    op =)
 G' G (one_step t) UNIV (get_mem o snd); CFGs t = Some G; CFGs' t = Some G';
```

```
 \<forall>t'. t' \<noteq> t \<longrightarrow> CFGs t' = CFGs' t'; \<forall>mem s mem' s'.
     sim_rel G' G (s, mem) (s', mem') \<longrightarrow> mem = mem';
 \<forall>s s' mem ops mem'. sim_rel G' G (s, mem) (s', mem) \<and> t \<notin> get_thread `
      ops \<and> update_mem mem ops mem' \<longrightarrow>
 sim_rel G' G (s, mem') (s', mem')\<rbrakk> \<Longrightarrow>
 tCFG_sim (lift_reach_sim_rel sim_rel CFGs' CFGs t) (op =) CFGs' CFGs conc_step UNIV (
     get_mem o snd)"
by (rule sim_by_reachable_thread, simp+, metis)

end

end

(* LLVM.thy *)
(* William Mansky *)
(* MiniLLVM for VeriF-OPT. *)

theory LLVM
imports trans_flowgraph step_rel
begin

(* Syntax *)
datatype LLVM_type = Int_ty | Pointer_ty LLVM_type ("_*")  (* Ignoring size for now. *)
datatype 'var LLVM_const = CInt 'var | CNull | CPointer int | CUndef
datatype ('var, 'cvar) LLVM_expr = Local 'var ("%_") | Const "'cvar LLVM_const" | Global
    string
datatype LLVM_op = add | sub | mul
datatype LLVM_cmp = eq | ne | sgt | sge | slt | sle

datatype ('node, 'var, 'type, 'expr, 'opr, 'cmp) LLVM_instr = Assign 'var 'opr 'type 'expr
    'expr ("_ = _ _ _, _" 160) |
 Ret 'type 'expr | Br_i1 'expr (* conditional branch *) | Br_label (* unconditional *) |
(* Note that, since control flow is implicit in the function body, label targets are
    unnecessary. *)
 Alloca 'var 'type (* the memory is freed when the allocing function returns. *) |
 Load 'var 'type 'expr | Store 'type 'expr 'type 'expr |
 Cmpxchg 'var 'type 'expr 'type 'expr 'type 'expr (* ordering constraint *) |
 ICmp 'var 'cmp 'type 'expr 'expr ("_ = icmp _ _ _, _" 160) |
 Phi 'var 'type "('node \<times> 'expr) list" ("_ = phi _ _" 160) |
 Call 'var 'type "'expr list" ("_ = call _ _" 160) | IsPointer 'expr
```

```
datatype LLVM_decl = Global_Decl string "int LLVM_const"

datatype LLVM_edge_type = seq | true | false | proc_call | ret
lemma finite_edge_types [simp]: "finite (UNIV::LLVM_edge_type set)"
apply (simp add: finite_conv_nat_seg_image)
apply (rule_tac x=5 in exI, rule_tac x="\<lambda>n. if n = 0 then seq else if n = 1 then
    true else
 if n = 2 then false else if n = 3 then proc_call else ret" in exI, auto)
apply (case_tac x, auto)
apply force
done
declare finite_edge_types [simp del]
datatype ('node, 'var, 'type, 'expr, 'opr, 'cmp) LLVM_function = Define LLVM_type string
    "('type \<times> string) list"
 "('node, LLVM_edge_type, ('node, 'var, 'type, 'expr, 'opr, 'cmp) LLVM_instr) flowgraph" ("
     define _ __ { _ }")

datatype ('node, 'var, 'type, 'expr, 'opr, 'cmp) LLVM_module =
 Module "LLVM_decl list" "('node, 'var, 'type, 'expr, 'opr, 'cmp) LLVM_function list"

(* Free variables *)
primrec expr_fv where
"expr_fv (Local x) = {x}" |
"expr_fv (Const _) = {}" |
"expr_fv (Global _) = {}"
lemma finite_expr_fv [simp]: "finite (expr_fv e)"
by (induct e, auto)
corollary finite_expr_pattern_fv [simp]: "finite (expr_pattern_fv expr_fv e)"
by (case_tac e, auto)
primrec instr_fv where
"instr_fv (Assign x _ _ e1 e2) = {x} \<union> expr_fv e1 \<union> expr_fv e2" |
"instr_fv (Ret _ e) = expr_fv e" |
"instr_fv (Br_i1 e) = expr_fv e" |
"instr_fv (Br_label) = {}" |
"instr_fv (Alloca x _) = {x}" |
"instr_fv (Load x _ e) = {x} \<union> expr_fv e" |
"instr_fv (Store _ e1 _ e2) = expr_fv e1 \<union> expr_fv e2" |
"instr_fv (Cmpxchg x _ e1 _ e2 _ e3) = {x} \<union> expr_fv e1 \<union> expr_fv e2 \<union>
     expr_fv e3" |
"instr_fv (ICmp x _ _ e1 e2) = {x} \<union> expr_fv e1 \<union> expr_fv e2" |
```

```
"instr_fv (Phi x _ es) = {x} \<union> (\<Union>(_, e)\<in>set es. expr_fv e)" |
"instr_fv (Call x _ es) = {x} \<union> (\<Union>e\<in>set es. expr_fv e)" |
"instr_fv (IsPointer e) = expr_fv e"


(* tICFG for LLVM module *)
fun LLVM_instr_edges where
"LLVM_instr_edges (Ret _ _) = {no_edges(ret := n) | n. True}" |
"LLVM_instr_edges (Br_i1 _) = {no_edges(true := 1, false := 1)}" |
"LLVM_instr_edges (Call _ _ _) = {no_edges(proc_call := 1, seq := 1)}" |
"LLVM_instr_edges _ = {no_edges(seq := 1)}"

corollary one_seq [simp]: "finite (Edges G) \<Longrightarrow> (edge_types {(u, t). (n, u, t
    ) \<in> Edges G} = no_edges(seq := Suc 0)) =
 (\<exists>m. (n, m, seq) \<in> Edges G \<and> (\<forall>u t. (n, u, t) \<in> Edges G \<
     longrightarrow> u = m \<and> t = seq))"
by (drule_tac n=n and ty=seq in out_one, auto simp add: out_edges_def)

lemma out_br [simp]: "finite (Edges G) \<Longrightarrow> (edge_types (out_edges (Edges G) n
    ) =
 no_edges(true := Suc 0, false := Suc 0)) =
 (\<exists>m1 m2. out_edges (Edges G) n = {(m1, true), (m2, false)})"
apply (rule iffI, frule_tac x=true in cong, simp, simp (no_asm_use) add: edge_types_def)
apply (frule_tac x=false in cong, simp, simp (no_asm_use) add: edge_types_def)
apply (auto simp add: card_Suc_eq)
done

(* Hypothesis: the additional bookkeeping involved in an ICFG is redundant for our purposes
    .
   Calls and returns can be stored in edge labels. *)

(* Semantics *)
(* basic expressions *)
abbreviation "start_env \<equiv> \<lambda>x. CUndef"


primrec eval_expr (*:: "('a \<Rightarrow> 'b LLVM_const) \<Rightarrow> (char list \<
    Rightarrow> 'b LLVM_const) \<Rightarrow> 'a LLVM_expr \<Rightarrow> 'b LLVM_const" *)
where
"eval_expr env _ (Local i) = env i" | (* i::type of LLVM_expr *)
```

```
"eval_expr env _ (Const c) = c" |     (* c::LLVM_const *)
"eval_expr _ gt (Global i) = gt i"    (* i::string *)

term eval_expr

primrec cmp_helper where
"cmp_helper eq v1 v2 = (v1 = v2)" |
"cmp_helper ne v1 v2 = (v1 \<noteq> v2)" |
"cmp_helper sgt v1 v2 = (v1 > v2)" |
"cmp_helper sge v1 v2 = (v1 \<ge> v2)" |
"cmp_helper slt v1 v2 = (v1 < v2)" |
"cmp_helper sle v1 v2 = (v1 \<le> v2)"

fun eval_cmp where
"eval_cmp env gt cmp e1 e2 = (case (eval_expr env gt e1, eval_expr env gt e2) of
   (CInt v1, CInt v2) \<Rightarrow> if cmp_helper cmp v1 v2 then CInt 1 else CInt 0
 | (CPointer v1, CPointer v2) \<Rightarrow> if cmp_helper cmp v1 v2 then CInt 1 else CInt 0
 | _ \<Rightarrow> CUndef)"

fun eval where
"eval env gt opr e1 e2 = (case (eval_expr env gt e1, eval_expr env gt e2(* eval_expr gt env
     e2 *)) of
 (CInt v1, CInt v2) \<Rightarrow> (case opr of add \<Rightarrow> CInt (v1 + v2) | sub \<
     Rightarrow> CInt (v1 - v2) | mul \<Rightarrow> CInt (v1 * v2))
 | _ \<Rightarrow> CUndef)"

(* Copied and adapted the good bits from JinjaThreads' ToString.thy. *)
function digit_toString :: "int \<Rightarrow> string"
where
  "digit_toString 0 = ''0''"
| "digit_toString 1 = ''1''"
| "digit_toString 2 = ''2''"
| "digit_toString 3 = ''3''"
| "digit_toString 4 = ''4''"
| "digit_toString 5 = ''5''"
| "digit_toString 6 = ''6''"
| "digit_toString 7 = ''7''"
| "digit_toString 8 = ''8''"
| "digit_toString 9 = ''9''"
| "n \<notin> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} ==> digit_toString n = undefined"
```

```
apply(case_tac x)
apply simp_all
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply(rename_tac n', case_tac n', simp)
apply simp
done
termination by lexicographic_order

function int_toString :: "int \<Rightarrow> string"
where
  "int_toString n =
  (if n < 0 then ''-'' @ int_toString (- n)
   else if n < 10 then digit_toString n
   else int_toString (n div 10) @ digit_toString (n mod 10))"
by pat_completeness simp
termination by size_change

lemma neg_int_toString [simp]: "n < 0 \<Longrightarrow> int_toString n = ''-'' @
    int_toString (- n)"
by simp

lemma digit_int_toString [simp]: "\<lbrakk>n \<ge> 0; n < 10\<rbrakk> \<Longrightarrow>
    int_toString n = digit_toString n"
by simp

lemma ten_int_toString [simp]: "n \<ge> 10 \<Longrightarrow> int_toString n = int_toString
    (n div 10) @
  digit_toString (n mod 10)"
by simp

lemmas int_toString.simps [simp del]
```

```
lemma int_digit: "\<lbrakk>(i::int) \<ge> 0; i < 10\<rbrakk> \<Longrightarrow> i \<in> {0,
    1, 2, 3, 4, 5, 6, 7, 8, 9}"
by (case_tac i, auto)

lemma one_digit: "i \<in> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} \<Longrightarrow> length (
    digit_toString i) = 1"
by auto

lemma inj_digit: "\<lbrakk>i \<ge> 0; i < 10; j \<ge> 0; j < 10; digit_toString i =
    digit_toString j\<rbrakk> \<Longrightarrow>
 i = j"
by (drule int_digit, simp, drule int_digit, auto)

lemma toString_nonempty [simp]: "int_toString i \<noteq> []"
apply (induct rule: int_toString.induct)
apply (case_tac "n < 0", simp)
apply (case_tac "n < 10", simp)
apply (cut_tac i=n in one_digit [OF int_digit], simp+)
apply (case_tac "digit_toString n", simp, simp)
apply (simp only: ten_int_toString)
by (metis Nil_is_append_conv)

abbreviation "nat_toString n \<equiv> int_toString (int n)"

lemma inj_toString: "\<lbrakk>int_toString x = int_toString y; x \<ge> 0; y \<ge> 0\<rbrakk
    > \<Longrightarrow> x = y"
oops

primrec init_env_aux where
"init_env_aux _ _ [] _ = start_env" |
"init_env_aux env gt (e # rest) n = (init_env_aux env gt rest (Suc n))((''arg'' @
    nat_toString n) := eval_expr env gt e)"
abbreviation "init_env env gt args \<equiv> init_env_aux env gt args 0"

locale LLVM_flowgraph = flowgraph where instr_edges=LLVM_instr_edges and Seq=seq

locale LLVM_tCFG = tCFG where instr_edges=LLVM_instr_edges and Seq=seq
begin

lemma LLVM_graph [simp, intro]: "CFGs t = Some G \<Longrightarrow>
```

```
 LLVM_flowgraph (Nodes G) (Start G) (Exit G) (Edges G) (Label G)"
apply (frule CFGs, simp add: is_flowgraph_def flowgraph_def pointed_graph_def
    flowgraph_axioms_def)
apply (unfold_locales, auto)
apply force
apply force
done

end

locale LLVM_MM = memory_model where update_mem="update_mem::'memory \<Rightarrow>
  ('thread, int, int LLVM_const) access set \<Rightarrow> 'memory \<Rightarrow> bool" for
      update_mem

locale LLVM = LLVM_flowgraph (* ASSUMED INT *)
 where L="L::'node \<Rightarrow> ('node, string, LLVM_type, (string, int) LLVM_expr,
     LLVM_op, LLVM_cmp) LLVM_instr" +
 LLVM_MM where update_mem="update_mem::'memory \<Rightarrow> ('thread, int, int LLVM_const)
      access set \<Rightarrow>
  'memory \<Rightarrow> bool" for L update_mem
begin

term eval
term L

(* Per-thread LLVM semantics *)
inductive LLVM_step where
assign [intro]: "\<lbrakk>L p = (x = opr ty e1, e2); p \<noteq> Exit\<rbrakk> \<
    Longrightarrow> LLVM_step t mem gt (p0, p, env, stack, allocad) {}
 (p, next_node Edges seq p, env(x := eval env gt opr e1 e2), stack, allocad)" |
ret_pop [intro]: "\<lbrakk>L p = Ret ty e; p \<noteq> Exit; (p, ret_addr, ret) \<in> Edges
    \<rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (_, p, env, (ret_addr, ret_var, ret_env, ret_allocad) # rest, allocad)
 (Free t ` allocad) (p, ret_addr, ret_env(ret_var := eval_expr env gt e), rest, ret_allocad
     )" |
ret_main [intro]: "\<lbrakk>L p = Ret ty e; p \<noteq> Exit; (p, Exit, ret) \<in> Edges\<
    rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (_, p, env, [], allocad) (Free t ` allocad)
 (p, Exit, env, [], {})" (* what to do when main function returns? *) |
```

```
branch_i [intro]: "\<lbrakk>L p = Br_i1 e; p \<noteq> Exit\<rbrakk> \<Longrightarrow>
    LLVM_step t mem gt (_, p, env, stack, allocad) {}
 (p, next_node Edges (if eval_expr env gt e = CInt 0 then false else true) p, env, stack,
     allocad)"
(* default to true unless cond is 0 *) |
branch_u [intro]: "\<lbrakk>L p = Br_label; p \<noteq> Exit\<rbrakk> \<Longrightarrow>
    LLVM_step t mem gt (_, p, env, stack, allocad) {}
 (p, next_node Edges seq p, env, stack, allocad)" |
(* memory ops *)
alloca [intro!]: "\<lbrakk>L p = Alloca x ty; new_loc \<in> free_set mem; p \<noteq> Exit\<
    rbrakk> \<Longrightarrow> LLVM_step t mem gt (_, p, env, stack, allocad)
 {Alloc t new_loc} (p, next_node Edges seq p, env(x := CPointer new_loc), stack, insert
     new_loc allocad)" |
load [intro!]: "\<lbrakk>L p = Load x ty e; eval_expr env gt e = CPointer l; v \<in>
    can_read mem t l; p \<noteq> Exit\<rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (_, p, env, stack, allocad) {Read t l v} (p, next_node Edges seq p, env
     (x := v), stack, allocad)" |
store [intro]: "\<lbrakk>L p = Store ty1 e1 ty2 e2; eval_expr env gt e2 = CPointer l; p \<
    noteq> Exit\<rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (_, p, env, stack, allocad) {Write t l (eval_expr env gt e1)} (p,
     next_node Edges seq p, env, stack, allocad)" |
cmpxchg_eq [intro]: "\<lbrakk>L p = Cmpxchg x ty1 e1 ty2 e2 ty3 e3; eval_expr env gt e1 =
    CPointer l; v \<in> can_read mem t l;
 eval_expr env gt e2 = v; p \<noteq> Exit\<rbrakk> \<Longrightarrow> LLVM_step t mem gt (_,
      p, env, stack, allocad) {ARW t l v (eval_expr env gt e3)}
 (p, next_node Edges seq p, env(x := v), stack, allocad)" |
cmpxchg_no [intro]: "\<lbrakk>L p = Cmpxchg x ty1 e1 ty2 e2 ty3 e3; eval_expr env gt e1 =
    CPointer l; v \<in> can_read mem t l;
 eval_expr env gt e2 \<noteq> v; p \<noteq> Exit\<rbrakk> \<Longrightarrow> LLVM_step t mem
      gt (_, p, env, stack, allocad) {ARW t l v v}
 (p, next_node Edges seq p, env(x := v), stack, allocad)" |
icmp [intro]: "\<lbrakk>L p = (x = icmp cmp ty e1, e2); p \<noteq> Exit\<rbrakk> \<
    Longrightarrow> LLVM_step t mem gt (_, p, env, stack, allocad) {}
 (p, next_node Edges seq p, env(x := eval_cmp env gt cmp e1 e2), stack, allocad)" |
phi [intro]: "\<lbrakk>L p = (x = phi ty vals); p \<noteq> Exit; (p0, e) \<in> set vals\<
    rbrakk> \<Longrightarrow> LLVM_step t mem gt
 (p0, p, env, stack, allocad) {} (p, next_node Edges seq p, env(x := eval_expr env gt e),
     stack, allocad)" |
call [intro]: "\<lbrakk>L p = (x = call ty args); p \<noteq> Exit\<rbrakk> \<Longrightarrow
    > LLVM_step t mem gt (_, p, env, stack, allocad) {}
```

```
 (p, next_node Edges proc_call p, init_env env gt args, (next_node Edges seq p, x, env,
     allocad) # stack, {})" |
ispointer [intro]: "\<lbrakk>L p = IsPointer e; eval_expr env gt e = CPointer l; p \<noteq>
     Exit\<rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (_, p, env, stack, allocad) {} (p, next_node Edges seq p, env, stack,
     allocad)"

lemma step_next [simp]: "LLVM_step t mem gt (p0, p, rest) a (p', rest') \<Longrightarrow> p
    ' = p"
by (erule LLVM_step.cases, simp_all)

lemma not_exit: "LLVM_step t mem gt (p0, p, rest) a C' \<Longrightarrow> p \<noteq> Exit"
by (erule LLVM_step.cases, simp_all)

lemma branch_true [intro]: "\<lbrakk>L p = Br_i1 e; p \<noteq> Exit; eval_expr env gt e \<
    noteq> CInt 0\<rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (p0, p, env, stack, allocad) {} (p, next_node Edges true p, env, stack,
      allocad)"
by (smt branch_i)

lemma branch_false [intro]: "\<lbrakk>L p = Br_i1 e; p \<noteq> Exit; eval_expr env gt e =
    CInt 0\<rbrakk> \<Longrightarrow>
 LLVM_step t mem gt (p0, p, env, stack, allocad) {} (p, next_node Edges false p, env, stack
     , allocad)"
by (smt branch_i)

end

definition "alloc_mem r \<equiv> case r of (stack, allocad) \<Rightarrow> allocad \<union>
    (\<Union>(a, b, c, d)\<in>set stack. d)"

context LLVM begin

lemma finite_ops: "\<lbrakk>LLVM_step t mem gt (p0, p, env, stack, allocad) ops (p, p', env
    ', stack', allocad');
 finite (alloc_mem (stack, allocad))\<rbrakk> \<Longrightarrow> finite ops \<and> finite (
     alloc_mem (stack', allocad'))"
by (erule LLVM_step.cases, auto simp add: alloc_mem_def)
```

```
lemma step_mem: "\<lbrakk>LLVM_step t mem gt (p0, p, env, stack, allocad) ops (p, p', env',
     stack', allocad');
 l \<notin> free_set mem; l \<notin> alloc_mem (stack, allocad)\<rbrakk> \<Longrightarrow>
     l \<notin> alloc_mem (stack', allocad')"
by (erule LLVM_step.cases, auto simp add: alloc_mem_def)

end

type_synonym ('thread, 'node, 'var) LLVM_tCFG = "('thread, ('node, LLVM_edge_type,
 ('node, 'var, LLVM_type, ('var,int) LLVM_expr, LLVM_op, LLVM_cmp) LLVM_instr) flowgraph)
     map"

context LLVM_MM begin

(* Process global variable declarations to get initial environments and memory. *)
inductive declare_global where
global [intro!]: "\<lbrakk>new_loc \<in> free_set mem; update_mem mem {Alloc t new_loc} mem
    ';
 update_mem mem' {ARW t' new_loc CUndef c} mem''\<rbrakk> \<Longrightarrow>
 declare_global (env, mem) (Global_Decl s c) (env(s := CPointer new_loc), mem'')"
inductive declare_globals where
no_globals [intro!, simp]: "declare_globals (env, mem) [] (env, mem)" |
a_global [intro!]: "\<lbrakk>declare_global (env, mem) d (env', mem');
 declare_globals (env', mem') rest (env'', mem'')\<rbrakk> \<Longrightarrow>
 declare_globals (env, mem) (d # rest) (env'', mem'')"

lemma declare_past: "\<lbrakk>\<forall>c. Global_Decl s c \<notin> set ds; declare_globals
    (env, mem0) ds (gt, mem)\<rbrakk> \<Longrightarrow>
 gt s = env s"
apply (induct ds arbitrary: env mem0, auto)
apply (erule declare_globals.cases, clarsimp+)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
done

lemma declare_not_free: "\<lbrakk>l \<notin> free_set mem; declare_globals (env, mem) ds (
    gt, mem')\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem'"
apply (induct ds arbitrary: env mem, auto)
apply (erule declare_globals.cases, clarsimp+)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
```

```
apply (drule stays_not_free, simp+)+
apply force
done

lemma global_alloc: "\<lbrakk>Global_Decl s c \<in> set ds; declare_globals C ds (gt, mem)
    \<rbrakk> \<Longrightarrow>
 \<exists>l. gt s = CPointer l \<and> l \<notin> free_set mem"
apply (induct ds arbitrary: c C gt mem, auto)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
apply (case_tac "\<exists>c. Global_Decl s c \<in> set rest", clarsimp+)
apply (drule declare_past, simp+)
apply (drule alloc_not_free, force, simp+)
apply (drule stays_not_free, simp+)
apply (drule declare_not_free, simp+)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
done

lemma declare_new: "\<lbrakk>Global_Decl s c \<in> set ds; declare_globals (e, mem) ds (gt,
     mem');
 gt s = CPointer l\<rbrakk> \<Longrightarrow> l \<in> free_set mem"
apply (induct ds arbitrary: e mem c, auto)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
apply (case_tac "\<exists>c. Global_Decl s c \<in> set rest", clarsimp)
apply (rule ccontr, drule stays_not_free, simp+, drule stays_not_free, simp+)
apply (drule declare_past, simp+)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
apply (rule ccontr, drule stays_not_free, simp+, drule stays_not_free, simp+)
done

lemma global_diff: "\<lbrakk>Global_Decl s c \<in> set ds; Global_Decl s' c' \<in> set ds;
 declare_globals C ds (gt, mem); gt s = gt s'\<rbrakk> \<Longrightarrow> s = s'"
apply (frule global_alloc, simp+, clarsimp)
apply (induct ds arbitrary: c c' C, auto)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
apply (case_tac "\<exists>c. Global_Decl s c \<in> set rest", clarsimp+)
apply (drule declare_past, simp+)
apply (drule declare_new, simp+)
apply (drule alloc_not_free, force, simp+)
apply (drule stays_not_free, simp+)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
```

```
apply (case_tac "\<exists>c. Global_Decl s' c \<in> set rest", clarsimp+)
apply (drule declare_past, simp+)
apply (drule declare_new, simp+)
apply (drule alloc_not_free, force, simp+)
apply (drule stays_not_free, simp+)
apply (erule declare_globals.cases, auto elim!: declare_global.cases)
done

end

locale LLVM_decls = fixes decls::"LLVM_decl list"

locale LLVM_base = LLVM_MM + LLVM_decls + fixes gt::"string \<Rightarrow> int LLVM_const"
begin

abbreviation "step t G mem \<equiv> LLVM.LLVM_step (Exit G) (Edges G) free_set can_read (
    Label G) t mem gt"

lemma LLVM_graph': "is_flowgraph G seq LLVM_instr_edges \<Longrightarrow>
  LLVM (Nodes G) (Start G) (Exit G) (Edges G) free_set (Label G) update_mem"
by (simp add: LLVM_def is_flowgraph_def LLVM_flowgraph_def, unfold_locales)

lemmas step_cases' = LLVM.LLVM_step.cases [OF LLVM_graph']
lemmas step_next' [simp] = LLVM.step_next [OF LLVM_graph']

lemma step_along_edge': "\<lbrakk>is_flowgraph G seq LLVM_instr_edges; step t G m (pp, p, r
    ) a (p, p', r'); p \<in> Nodes G\<rbrakk> \<Longrightarrow>
 \<exists>e. (p, p', e) \<in> Edges G"
apply (subgoal_tac "LLVM (Nodes G) (Start G) (Exit G) (Edges G) free_set (Label G)
    update_mem")
apply (clarsimp simp add: is_flowgraph_def flowgraph_def flowgraph_axioms_def)
apply ((erule_tac x=p in allE)+)
apply (frule pointed_graph.finite_edges)
apply (erule LLVM.LLVM_step.cases, simp_all, simp_all, clarsimp)
apply (clarsimp simp add: out_edges_def, force)
apply (clarsimp simp add: out_edges_def, force)
apply force
apply clarsimp
apply (rule conjI, force simp add: out_edges_def, force simp add: out_edges_def)
apply (force simp add: out_edges_def)
```

```
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (clarsimp, simp add: out_edges_def)
apply (metis (lifting) insertI1 mem_Collect_eq splitD)
apply (clarsimp, simp add: out_edges_def, force)
apply (simp add: LLVM_def LLVM_flowgraph_def is_flowgraph_def, unfold_locales)
done

end

(* Concurrent semantics *)
sublocale LLVM_base \<subseteq> step_rel where free_set=free_set and update_mem=update_mem
  and start_mem=start_mem and can_read=can_read and step_rel=step and
  get_point="\<lambda>(_, p, env, stack, allocad). p::'node" and instr_edges="
      LLVM_instr_edges::('node, char list,
  LLVM_type, (char list, int) LLVM_expr, LLVM_op, LLVM_cmp) LLVM_instr \<Rightarrow> (
      LLVM_edge_type \<Rightarrow> nat) set"
  and Seq=seq and start_state="\<lambda>CFGs C0 mem0. declare_globals (start_env, start_mem
      ) decls (gt, mem0) \<and>
    C0 = (\<lambda>t. case CFGs t of Some G \<Rightarrow> Some (Start G, Start G, start_env
        , [], {}) | None \<Rightarrow> None)"
apply unfold_locales
apply (rule ext, simp add: start_points_def map_comp_def)
apply clarsimp
apply (frule step_next', simp, clarsimp)
apply (metis step_along_edge')
done

lemma (in LLVM_base) one_step_next: "\<lbrakk>one_step t G ((p0, p, rest), m) ((p', rest'),
     m');
  is_flowgraph G seq LLVM_instr_edges\<rbrakk> \<Longrightarrow> p' = p"
by (erule one_step.cases, auto)

locale LLVM_threads = LLVM_tCFG where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
```

```
 LLVM_base where update_mem="update_mem::'memory \<Rightarrow> ('thread, int, int
     LLVM_const) access set \<Rightarrow> 'memory \<Rightarrow> bool"
 for CFGs update_mem
begin

lemma LLVM_graph [simp, intro]: "CFGs t = Some G \<Longrightarrow>
 LLVM (Nodes G) (Start G) (Exit G) (Edges G) free_set (Label G) update_mem"
by (frule CFGs, erule LLVM_graph')

lemmas step_cases = LLVM.LLVM_step.cases [OF LLVM_graph]
lemmas step_next [simp] = LLVM.step_next [OF LLVM_graph]

lemma step_along_edge: "\<lbrakk>CFGs t = Some G; step t G m (pp, p, r) a (p, p', r'); p \<
    in> Nodes G\<rbrakk> \<Longrightarrow>
 \<exists>e. (p, p', e) \<in> Edges G"
by (rule step_along_edge', drule CFGs, auto)

abbreviation "conc_step_star \<equiv> (conc_step CFGs)^**"

lemma ops_thread: "\<lbrakk>step t G mem state ops state'; CFGs t = Some G; a \<in> ops\<
    rbrakk> \<Longrightarrow> get_thread a = t"
by (rule step_cases, auto)

definition "all_alloc_mem states \<equiv> \<Union>(a0, a, b, c, d)\<in>ran states.
    alloc_mem (c, d)"

lemma run_global: "\<lbrakk>run_prog CFGs C; Global_Decl s c \<in> set decls\<rbrakk> \<
    Longrightarrow>
 \<exists>l. gt s = CPointer l \<and> l \<notin> free_set (snd C) \<and> l \<notin>
     all_alloc_mem (fst C)"
apply (rule run_prog_induct, simp, clarsimp simp add: run_prog_def)
apply (drule global_alloc, simp+, clarsimp simp add: all_alloc_mem_def ran_def
    alloc_mem_def
 split: option.splits)
apply clarsimp
apply (erule conc_step.cases, clarsimp simp add: all_alloc_mem_def ran_def)
apply (rule conjI, clarsimp)
apply (drule stays_not_free, simp, clarsimp simp add: alloc_mem_def)
apply (drule step_cases, simp, simp_all, force, force)
apply clarsimp
```

```
apply (case_tac "al \<noteq> t", force, clarsimp)
apply (frule step_next, simp+)
apply (drule_tac can_read=can_read and t=t in LLVM.step_mem [OF LLVM_graph], simp+, force,
    simp)
done

lemma call_edges [simp]: "finite (Edges G) \<Longrightarrow>
  (edge_types (out_edges (Edges G) p) = no_edges(proc_call := Suc 0, seq := Suc 0))
 = (\<exists>m n. out_edges (Edges G) p = {(m, proc_call), (n, seq)})"
by simp

end

(* Memory models *)
locale LLVM_TSO = tCFG where instr_edges=LLVM_instr_edges and Seq=seq and
  CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" + TSO where undef="CUndef::int LLVM_const
      " for CFGs

sublocale LLVM_TSO \<subseteq> TSO: LLVM_threads where update_mem=update_mem and free_set=
    free_set
  and can_read=can_read and start_mem=start_mem
by (unfold_locales)

locale LLVM_SC = tCFG where instr_edges=LLVM_instr_edges and Seq=seq and
  CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" + SC where undef="CUndef::int LLVM_const"
       for CFGs

sublocale LLVM_SC \<subseteq> SC: LLVM_threads where update_mem=update_mem and free_set=
    free_set
  and can_read=can_read and start_mem=start_mem
by (unfold_locales)

locale LLVM_PSO = tCFG where instr_edges=LLVM_instr_edges and Seq=seq and
  CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" + PSO where undef="CUndef::int LLVM_const
      " for CFGs

sublocale LLVM_PSO \<subseteq> PSO: LLVM_threads where update_mem=update_mem and free_set=
    free_set
  and can_read=can_read and start_mem=start_mem
by (unfold_locales)
```

```
lemma (in LLVM_SC) safe_mem: "\<lbrakk>l \<notin> free_set mem; l \<notin> alloc_mem (stack
    , allocad); CFGs t = Some G;
 LLVM.LLVM_step (Exit G) (Edges G) free_set can_read (Label G) t mem gt (p0, p, env, stack,
     allocad)
 ops (p, p', env', stack', allocad'); update_mem mem ops mem'; finite (alloc_mem (stack,
     allocad))\<rbrakk> \<Longrightarrow>
 l \<notin> free_set mem' \<and> l \<notin> alloc_mem (stack', allocad') \<and> finite (
     alloc_mem (stack', allocad'))"
by (rule LLVM.LLVM_step.cases, erule LLVM_graph, simp, auto simp add: alloc_mem_def)

(* Memory models in LLVM. *)
context LLVM_threads begin

declare step_cases [elim]

abbreviation "step_SC \<equiv> LLVM_base.step SC.free_set SC.can_read gt"
abbreviation "step_TSO \<equiv> LLVM_base.step TSO.free_set TSO.can_read gt"

declare eval.simps [simp del] eval_cmp.simps [simp del]

end

(* Patterns and substitution *)
type_synonym 'mvar pattern = "(('mvar node_lit, 'mvar, (LLVM_type, 'mvar) literal,
 ('mvar, ('mvar,int) LLVM_expr) expr_pattern, (LLVM_op, 'mvar) literal, (LLVM_cmp, 'mvar)
     literal) LLVM_instr, 'mvar) literal"

primrec instr_pattern_fv where
"instr_pattern_fv (Assign x opr ty e1 e2) = {x} \<union> type_fv opr \<union> type_fv ty \<
    union> expr_pattern_fv expr_fv e1 \<union> expr_pattern_fv expr_fv e2" |
"instr_pattern_fv (Ret ty e) = type_fv ty \<union> expr_pattern_fv expr_fv e" |
"instr_pattern_fv (Br_i1 e) = expr_pattern_fv expr_fv e" |
"instr_pattern_fv (Br_label) = {}" |
"instr_pattern_fv (Alloca x ty) = {x} \<union> type_fv ty" |
"instr_pattern_fv (Load x ty e) = {x} \<union> type_fv ty \<union> expr_pattern_fv expr_fv
    e" |
"instr_pattern_fv (Store ty1 e1 ty2 e2) = type_fv ty1 \<union> expr_pattern_fv expr_fv e1
    \<union> type_fv ty2 \<union> expr_pattern_fv expr_fv e2" |
```

```
"instr_pattern_fv (Cmpxchg x ty1 e1 ty2 e2 ty3 e3) = {x} \<union> type_fv ty1 \<union>
    expr_pattern_fv expr_fv e1 \<union>
 type_fv ty2 \<union> expr_pattern_fv expr_fv e2 \<union> type_fv ty3 \<union>
     expr_pattern_fv expr_fv e3" |
"instr_pattern_fv (ICmp x cmp ty e1 e2) = {x} \<union> type_fv cmp \<union> type_fv ty \<
    union> expr_pattern_fv expr_fv e1 \<union> expr_pattern_fv expr_fv e2" |
"instr_pattern_fv (Phi x ty es) = {x} \<union> type_fv ty \<union> (\<Union>(n, e)\<in>set
    es. node_fv n \<union> expr_pattern_fv expr_fv e)" |
"instr_pattern_fv (Call x ty es) = {x} \<union> type_fv ty \<union> (\<Union>e\<in>set es.
    expr_pattern_fv expr_fv e)" |
"instr_pattern_fv (IsPointer e) = expr_pattern_fv expr_fv e"
lemma finite_instr_pattern_fv [simp]: "finite (instr_pattern_fv i)"
by (induct i, auto)

abbreviation pattern_fv::"'mvar pattern \<Rightarrow> 'mvar set" where
"pattern_fv p \<equiv> lit_fv instr_pattern_fv p"
lemma finite_pattern_fv [simp]: "finite (pattern_fv p)"
by (case_tac p, auto)

(* Applying a valuation of metavariables to an expression pattern yields a concrete
    expression. *)

(* Objects are the values metavariables can take. *)
datatype ('thread, 'node, 'var) object = ONode 'node
 | OInstr "('node, 'var, LLVM_type, ('var,int) LLVM_expr, LLVM_op, LLVM_cmp) LLVM_instr" |
     OExpr "('var,int) LLVM_expr"
 | OSynFunc "('var,int) LLVM_expr" "('var,int) LLVM_expr" | OEdgeType LLVM_edge_type |
     OThread 'thread
 | OType LLVM_type | OOp LLVM_op | OCmp LLVM_cmp

primrec type_subst where
"type_subst \<sigma> (MVar m) = (case \<sigma> m of OEdgeType t \<Rightarrow> Some t | _ \<
    Rightarrow> None)" |
"type_subst \<sigma> (Inj t) = Some t"

lemma edge_type_same_subst: "\<forall>x\<in>type_fv ty. \<sigma> x = \<sigma>' x \<
    Longrightarrow> type_subst \<sigma> ty = type_subst \<sigma>' ty"
by (cases ty, auto)

fun node_subst where
```

```
"node_subst CFGs \<sigma> (MVar m) = (case \<sigma> m of ONode t \<Rightarrow> Some t | _
    \<Rightarrow> None)" |
"node_subst CFGs \<sigma> (Inj (NExit t)) = (case \<sigma> t of OThread t' \<Rightarrow> (
    case CFGs t' of
  Some G \<Rightarrow> Some (Exit G) | _ \<Rightarrow> None) | _ \<Rightarrow> None)" |
"node_subst CFGs \<sigma> (Inj (NStart t)) = (case \<sigma> t of OThread t' \<Rightarrow> (
    case CFGs t' of
  Some G \<Rightarrow> Some (Start G) | _ \<Rightarrow> None) | _ \<Rightarrow> None)"

lemma node_same_subst: "\<forall>x\<in>node_fv n. \<sigma> x = \<sigma>' x \<Longrightarrow
    > node_subst CFGs \<sigma> n = node_subst CFGs \<sigma>' n"
apply (cases n, auto)
apply (case_tac data, auto split: object.splits)
done

primrec ty_subst where
"ty_subst \<sigma> (MVar m) = (case \<sigma> m of OType t \<Rightarrow> Some t | _ \<
    Rightarrow> None)" |
"ty_subst \<sigma> (Inj t) = Some t"

lemma type_same_subst: "\<forall>x\<in>type_fv ty. \<sigma> x = \<sigma>' x \<
    Longrightarrow> ty_subst \<sigma> ty = ty_subst \<sigma>' ty"
by (cases ty, auto)

primrec op_subst where
"op_subst \<sigma> (MVar m) = (case \<sigma> m of OOp t \<Rightarrow> Some t | _ \<
    Rightarrow> None)" |
"op_subst \<sigma> (Inj t) = Some t"

lemma op_same_subst: "\<forall>x\<in>lit_fv (\<lambda>x. {}) p. \<sigma> x = \<sigma>' x \<
    Longrightarrow> op_subst \<sigma> p = op_subst \<sigma>' p"
by (cases p, auto)

primrec cmp_subst where
"cmp_subst \<sigma> (MVar m) = (case \<sigma> m of OCmp t \<Rightarrow> Some t | _ \<
    Rightarrow> None)" |
"cmp_subst \<sigma> (Inj t) = Some t"

lemma cmp_same_subst: "\<forall>x\<in>lit_fv (\<lambda>x. {}) p. \<sigma> x = \<sigma>' x
    \<Longrightarrow> cmp_subst \<sigma> p = cmp_subst \<sigma>' p"
```

```
by (cases p, auto)

lemma case_expr [simp]: "((case \<sigma> v of OExpr x \<Rightarrow> Some x | _ \<Rightarrow
    > None) = Some a) = (\<sigma> v = OExpr a)"
by (case_tac "\<sigma> v", auto)
lemma case_node [simp]: "((case \<sigma> v of ONode x \<Rightarrow> Some x | _ \<Rightarrow
    > None) = Some a) = (\<sigma> v = ONode a)"
by (case_tac "\<sigma> v", auto)
lemma case_type [simp]: "((case \<sigma> v of OType x \<Rightarrow> Some x | _ \<Rightarrow
    > None) = Some a) = (\<sigma> v = OType a)"
by (case_tac "\<sigma> v", auto)
lemma case_thread [simp]: "((case \<sigma> v of OThread x \<Rightarrow> P x | _ \<
    Rightarrow> False)) = (\<exists>t. \<sigma> v = OThread t \<and> P t)"
by (case_tac "\<sigma> v", auto)

primrec expr_subst where
"expr_subst \<sigma> (Local i) = (case \<sigma> i of OExpr e \<Rightarrow> Some e | _ \<
    Rightarrow> None)" |
"expr_subst \<sigma> (Const c) = Some (Const c)" |
"expr_subst \<sigma> (Global s) = Some (Global s)"

lemma expr_same_subst: "\<forall>v\<in>expr_fv e. \<sigma> v = \<sigma>' v \<Longrightarrow
    > expr_subst \<sigma> e = expr_subst \<sigma>' e"
by (induct e, auto)

lemma self_subst: "expr_subst (\<lambda>y. OExpr (Local y)) e = Some e"
by (induct e, auto)

corollary subst_out: "\<forall>v\<in>expr_fv e. \<sigma> v = OExpr (Local v) \<
    Longrightarrow> expr_subst \<sigma> e = Some e"
by (rule trans, rule expr_same_subst, auto simp add: self_subst)

lemma sub_out: "\<lbrakk>v \<in> expr_fv e; v' \<notin> expr_fv e\<rbrakk> \<Longrightarrow
    > \<exists>e'.
 expr_subst ((\<lambda>y. OExpr (Local y))(v := OExpr (Local v'))) e = Some e' \<and>
 expr_subst ((\<lambda>y. OExpr (Local y))(v' := OExpr (Local v))) e' = Some e"
by (induct e, auto)

primrec expr_pattern_subst::"('mvar \<Rightarrow> ('thread, 'node, 'var) object) \<
    Rightarrow>
```

```
 ('mvar, ('mvar,int) LLVM_expr) expr_pattern \<Rightarrow> ('var,int) LLVM_expr option"
     where
"expr_pattern_subst \<sigma> (EPInj e) = expr_subst \<sigma> e" |
"expr_pattern_subst \<sigma> (x<e>) = (case (\<sigma> x, expr_subst \<sigma> e) of
 (OSynFunc (Local x) e, Some e') \<Rightarrow> expr_subst ((\<lambda>y. OExpr (Local y))
 (x := (OExpr e')::('thread, 'node, 'var) object)) e | _ \<Rightarrow> None)"
(* Isabelle note: this type annotation is necessary because OExpr e', in itself,
   is of type (?, ?, 'var) object.  We need to identify the missing types with existing
   type variables, or they'll be introduced as separate variables. *)

lemma expr_pattern_same_subst: "\<forall>v\<in>expr_pattern_fv expr_fv e. \<sigma> v = \<
    sigma>' v \<Longrightarrow>
 expr_pattern_subst \<sigma> e = expr_pattern_subst \<sigma>' e"
apply (induct e, auto simp add: expr_same_subst split: object.split LLVM_expr.split)
apply (drule expr_same_subst, simp)+
done

lemma eplist_same_subst: "\<forall>x\<in>\<Union>x\<in>set es. expr_pattern_fv expr_fv x.
    \<sigma> x = \<sigma>' x \<Longrightarrow>
 foldr (\<lambda>e r. case (expr_pattern_subst \<sigma> e, r) of (Some e', Some es) \<
     Rightarrow> Some (e' # es) | _ \<Rightarrow> None) es (Some []) =
 foldr (\<lambda>e r. case (expr_pattern_subst \<sigma>' e, r) of (Some e', Some es) \<
     Rightarrow> Some (e' # es) | _ \<Rightarrow> None) es (Some [])"
apply (induct es, auto)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=a in expr_pattern_same_subst
    , simp, simp split: option.splits)+
done

lemma philist_same_subst: "\<forall>x\<in>\<Union>(n, x)\<in>set es. node_fv n \<union>
    expr_pattern_fv expr_fv x. \<sigma> x = \<sigma>' x \<Longrightarrow>
 foldr (\<lambda>(n, e) r. case (node_subst G \<sigma> n, expr_pattern_subst \<sigma> e, r)
      of (Some n', Some e', Some es) \<Rightarrow> Some ((n', e') # es) | _ \<Rightarrow>
     None) es (Some []) =
 foldr (\<lambda>(n, e) r. case (node_subst G \<sigma>' n, expr_pattern_subst \<sigma>' e,
     r) of (Some n', Some e', Some es) \<Rightarrow> Some ((n', e') # es) | _ \<Rightarrow>
     None) es (Some [])"
apply (induct es, auto)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and n=a and CFGs=G in
    node_same_subst, simp, simp split: option.splits)
```

```
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=b in expr_pattern_same_subst
    , simp, simp split: option.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and n=a and CFGs=G in
    node_same_subst, simp, simp split: option.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=b in expr_pattern_same_subst
    , simp, simp split: option.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and n=a and CFGs=G in
    node_same_subst, simp, simp split: option.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=b in expr_pattern_same_subst
    , simp, simp split: option.splits)
done

(* Similarly, applying a valuation of metavariables to an instruction pattern yields a
    concrete instruction. *)
primrec instr_subst where
"instr_subst _ \<sigma> (Assign x opr ty e1 e2) = (case (\<sigma> x, op_subst \<sigma> opr,
     ty_subst \<sigma> ty, expr_pattern_subst \<sigma> e1, expr_pattern_subst \<sigma> e2)
    of
  (OExpr (Local i'), Some opr', Some ty', Some e1', Some e2') \<Rightarrow> Some (Assign i'
      opr' ty' e1' e2') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (Ret ty e) = (case (ty_subst \<sigma> ty, expr_pattern_subst \<
    sigma> e) of (Some ty', Some e') \<Rightarrow>
  Some (Ret ty' e') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (Br_i1 e) = (case expr_pattern_subst \<sigma> e of Some e' \<
    Rightarrow> Some (Br_i1 e') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> Br_label = Some Br_label" |
"instr_subst _ \<sigma> (Alloca x ty) = (case (\<sigma> x, ty_subst \<sigma> ty) of (OExpr
    (Local x'), Some ty') \<Rightarrow>
  Some (Alloca x' ty') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (Load x ty e) = (case (\<sigma> x, ty_subst \<sigma> ty,
    expr_pattern_subst \<sigma> e) of
  (OExpr (Local x'), Some ty', Some e') \<Rightarrow> Some (Load x' ty' e') | _ \<
      Rightarrow> None)" |
"instr_subst _ \<sigma> (Store ty1 e1 ty2 e2) = (case (ty_subst \<sigma> ty1,
    expr_pattern_subst \<sigma> e1,
 ty_subst \<sigma> ty2, expr_pattern_subst \<sigma> e2) of (Some ty1', Some e1', Some ty2',
      Some e2') \<Rightarrow>
  Some (Store ty1' e1' ty2' e2') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (Cmpxchg x ty1 e1 ty2 e2 ty3 e3) = (case (\<sigma> x, ty_subst \<
    sigma> ty1, expr_pattern_subst \<sigma> e1,
```

```
 ty_subst \<sigma> ty2, expr_pattern_subst \<sigma> e2, ty_subst \<sigma> ty3,
     expr_pattern_subst \<sigma> e3) of
 (OExpr (Local x'), Some ty1', Some e1', Some ty2', Some e2', Some ty3', Some e3') \<
     Rightarrow>
 Some (Cmpxchg x' ty1' e1' ty2' e2' ty3' e3') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (ICmp x opr ty e1 e2) = (case (\<sigma> x, cmp_subst \<sigma> opr,
    ty_subst \<sigma> ty, expr_pattern_subst \<sigma> e1, expr_pattern_subst \<sigma> e2) of
  (OExpr (Local i'), Some opr', Some ty', Some e1', Some e2') \<Rightarrow> Some (ICmp i'
      opr' ty' e1' e2') | _ \<Rightarrow> None)" |
"instr_subst G \<sigma> (Phi x ty es) = (case (\<sigma> x, ty_subst \<sigma> ty, foldr (\<
    lambda>(n, e) r. case (node_subst G \<sigma> n, expr_pattern_subst \<sigma> e, r) of
 (Some n', Some e', Some es) \<Rightarrow> Some ((n', e') # es) | _ \<Rightarrow> None) es
     (Some [])) of (OExpr (Local x'), Some ty', Some es') \<Rightarrow>
 Some (Phi x' ty' es') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (Call x ty es) = (case (\<sigma> x, ty_subst \<sigma> ty, foldr (\<
    lambda>e r. case (expr_pattern_subst \<sigma> e, r) of
 (Some e', Some es) \<Rightarrow> Some (e' # es) | _ \<Rightarrow> None) es (Some [])) of (
     OExpr (Local x'), Some ty', Some es') \<Rightarrow>
 Some (Call x' ty' es') | _ \<Rightarrow> None)" |
"instr_subst _ \<sigma> (IsPointer e) = (case expr_pattern_subst \<sigma> e of Some e' \<
    Rightarrow> Some (IsPointer e')
 | _ \<Rightarrow> None)"

lemma instr_same_subst: "\<forall>x\<in>instr_pattern_fv i. \<sigma> x = \<sigma>' x \<
    Longrightarrow> instr_subst G \<sigma> i = instr_subst G \<sigma>' i"
apply (cases i, simp_all)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and p=opr in op_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr2 in
    expr_pattern_same_subst, simp+)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr in
    expr_pattern_same_subst, simp+)
```

```
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type1 in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type2 in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr2 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type1 in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type2 in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr2 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type3 in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr3 in
    expr_pattern_same_subst, simp+)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and p=cmp in cmp_same_subst, simp
    +)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
```

```
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr2 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and G=G and es=list in
    philist_same_subst, simp+)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and ty=type in type_same_subst,
    simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and es=list in eplist_same_subst,
    simp+)
apply (case_tac "\<sigma>' var", simp_all)
apply (clarsimp split: LLVM_expr.splits)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr in
    expr_pattern_same_subst, simp+)
done

lemma foldr_Some: "\<forall>e\<in>set es. \<exists>e'. expr_pattern_subst \<sigma> e = Some
     e' \<Longrightarrow>
 foldr (\<lambda>e r. case (expr_pattern_subst \<sigma> e, r) of
 (Some e', Some es) \<Rightarrow> Some (e' # es) | _ \<Rightarrow> None) es (Some []) =
 Some (map (\<lambda>x. the (expr_pattern_subst \<sigma> x)) es)"
by (induct es, auto)

lemma foldr_None: "\<exists>e\<in>set es. expr_pattern_subst \<sigma> e = None \<
    Longrightarrow>
 foldr (\<lambda>e r. case (expr_pattern_subst \<sigma> e, r) of
 (Some e', Some es) \<Rightarrow> Some (e' # es) | _ \<Rightarrow> None) es (Some []) =
     None"
by (induct es, auto split: option.split, force)

primrec subst where
"subst G \<sigma> (Inj i) = instr_subst G \<sigma> i" |
"subst G \<sigma> (MVar x) = (case \<sigma> x of OInstr i \<Rightarrow> Some i | _ \<
    Rightarrow> None)"
```

```
lemma pattern_same_subst: "\<forall>x\<in>pattern_fv p. \<sigma> x = \<sigma>' x \<
    Longrightarrow> subst G \<sigma> p = subst G \<sigma>' p"
by (cases p, auto simp add: instr_same_subst)

end

(* LLVM_preds.thy *)
(* William Mansky *)
(* Atomic and derived predicates for MiniLLVM. *)

theory LLVM_preds
imports LLVM trans_preds
begin

datatype 'mvar LLVM_pred =
  Conlit "('mvar, ('mvar,int) LLVM_expr) expr_pattern"
| LVarlit "('mvar, ('mvar,int) LLVM_expr) expr_pattern"
| GVarlit "('mvar, ('mvar,int) LLVM_expr) expr_pattern"
| SameVal "('mvar, ('mvar,int) LLVM_expr) expr_pattern" "('mvar, ('mvar,int) LLVM_expr)
    expr_pattern"
| IsCall 'mvar | IsRet 'mvar | Def 'mvar 'mvar

primrec LLVM_pred_fv where
"LLVM_pred_fv (Conlit e) = expr_pattern_fv expr_fv e" |
"LLVM_pred_fv (LVarlit e) = expr_pattern_fv expr_fv e" |
"LLVM_pred_fv (GVarlit e) = expr_pattern_fv expr_fv e" |
"LLVM_pred_fv (SameVal e1 e2) = expr_pattern_fv expr_fv e1 \<union> expr_pattern_fv expr_fv
     e2" |
"LLVM_pred_fv (IsCall t) = {t}" |
"LLVM_pred_fv (IsRet t) = {t}" |
"LLVM_pred_fv (Def t x) = {t, x}"

lemma LLVM_pred_fv_finite [simp]: "finite (LLVM_pred_fv p)"
by (case_tac p, auto)

context LLVM_decls begin

primrec eval_LLVM_pred where
"eval_LLVM_pred CFGs q \<sigma> (Conlit e) = (case expr_pattern_subst \<sigma> e of Some e'
     \<Rightarrow>
```

```
 (case e' of Const _ \<Rightarrow> True | _ \<Rightarrow> False) | _ \<Rightarrow> False)"
    |
"eval_LLVM_pred CFGs q \<sigma> (LVarlit e) = (case expr_pattern_subst \<sigma> e of Some e
    ' \<Rightarrow>
 (case e' of Local _ \<Rightarrow> True | _ \<Rightarrow> False) | _ \<Rightarrow> False)"
    |
"eval_LLVM_pred CFGs q \<sigma> (GVarlit e) = (case expr_pattern_subst \<sigma> e of Some e
    ' \<Rightarrow>
 (case e' of Global s \<Rightarrow> \<exists>c. Global_Decl s c \<in> set decls | _ \<
    Rightarrow> False) | _ \<Rightarrow> False)" |
"eval_LLVM_pred CFGs q \<sigma> (SameVal e1 e2) = (case (expr_pattern_subst \<sigma> e1,
    expr_pattern_subst \<sigma> e2) of
 (Some e1', Some e2') \<Rightarrow> e1' = e2' | _ \<Rightarrow> False)" |
"eval_LLVM_pred CFGs q \<sigma> (IsCall t) = (case \<sigma> t of OThread t' \<Rightarrow>
 \<exists>n G x ty es. q t' = Some n \<and> CFGs t' = Some G \<and> n \<noteq> Exit G \<and
    > Label G n = Call x ty es | _ \<Rightarrow> False)" |
"eval_LLVM_pred CFGs q \<sigma> (IsRet t) = (case \<sigma> t of OThread t' \<Rightarrow>
 \<exists>n G e ty. q t' = Some n \<and> CFGs t' = Some G \<and> n \<noteq> Exit G \<and>
    Label G n = Ret e ty | _ \<Rightarrow> False)" |
"eval_LLVM_pred CFGs q \<sigma> (Def t x) = (case (\<sigma> t, \<sigma> x) of (OThread t',
    OExpr (Local x')) \<Rightarrow>
 \<exists>n G ty. q t' = Some n \<and> CFGs t' = Some G \<and> n \<noteq> Exit G \<and>
    ((\<exists>opr e1 e2. Label G n = Assign x' opr ty e1 e2) \<or>
 Label G n = Alloca x' ty \<or> (\<exists>e. Label G n = Load x' ty e) \<or> (\<exists>e
    ty2 e2 ty3 e3. Label G n = Cmpxchg x' ty e ty2 e2 ty3 e3) \<or>
 (\<exists>c e1 e2. Label G n = ICmp x' c ty e1 e2) \<or> (\<exists>es. Label G n = Phi x'
    ty es) \<or> (\<exists>es. Label G n = Call x' ty es)) | _ \<Rightarrow> False)"

lemma LLVM_pred_same_subst: "\<forall>x\<in>LLVM_pred_fv p. \<sigma> x = \<sigma>' x \<
    Longrightarrow>
  eval_LLVM_pred CFGs q \<sigma> p = eval_LLVM_pred CFGs q \<sigma>' p"
apply (case_tac p, simp_all)
apply (drule expr_pattern_same_subst, simp)+
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr_pattern1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr_pattern2 in
    expr_pattern_same_subst, simp+)
done

end
```

```
datatype ('mvar, 'other) ext_pred = Base "'mvar LLVM_pred" | Ext 'other

locale LLVM_preds = LLVM_decls + fixes eval_other::"(('thread, 'node, string) LLVM_tCFG) \<
    Rightarrow>
  ('thread \<rightharpoonup> 'node) \<Rightarrow> (string \<Rightarrow> ('thread, 'node,
      string) object) \<Rightarrow> 'other \<Rightarrow> bool"
  and other_fv::"'other \<Rightarrow> string set"
  assumes other_same_subst: "\<forall>x\<in>other_fv r. \<sigma> x = \<sigma>' x \<
      Longrightarrow>
    eval_other CFGs q \<sigma> r = eval_other CFGs q \<sigma>' r"
  and other_fv_finite [simp]: "finite (other_fv r)"
  and infinite_nodes [simp]: "\<not>finite (UNIV::'node set)"
begin

primrec ext_fv where
"ext_fv (Base p) = LLVM_pred_fv p" |
"ext_fv (Ext z) = other_fv z"

primrec eval_ext where
"eval_ext CFGs q \<sigma> (Base p) = eval_LLVM_pred CFGs q \<sigma> p" |
"eval_ext CFGs q \<sigma> (Ext z) = eval_other CFGs q \<sigma> z"

lemma ext_same_subst: "\<forall>x\<in>ext_fv P. \<sigma> x = \<sigma>' x \<Longrightarrow>
    eval_ext CFGs q \<sigma> P = eval_ext CFGs q \<sigma>' P"
apply (case_tac P, simp, rule LLVM_pred_same_subst, simp)
apply (simp, rule other_same_subst, simp)
done

end

sublocale LLVM_preds \<subseteq> TRANS_preds where subst=subst and node_subst=node_subst
  and type_subst=type_subst and thread_subst="\<lambda>\<sigma> x. case \<sigma> x of
      OThread t \<Rightarrow> Some t | _ \<Rightarrow> None"
  and var_subst="\<lambda>\<sigma> x. case \<sigma> x of OExpr (Local v) \<Rightarrow> Some
       v | _ \<Rightarrow> None"
  and eval_other=eval_ext and type_fv=type_fv and pat_fv=instr_pattern_fv and other_fv=
      ext_fv
  and instr_fv=instr_fv and int_of="\<lambda>\<sigma> x. case \<sigma> x of OExpr (Const (
      CInt i)) \<Rightarrow> Some i | _ \<Rightarrow> None"
```

```
  and instr_edges=LLVM_instr_edges and Seq=seq
apply (unfold_locales, simp_all)
apply (erule pattern_same_subst)
apply (erule node_same_subst)
apply (erule edge_type_same_subst)
apply (erule ext_same_subst)
apply (case_tac r, auto)
by (metis infinite_UNIV_listI)

context LLVM_preds begin

(* Helpful derived predicates. *)
abbreviation SCEX ("EX") where "EX t \<phi> \<equiv> let n = SOME y. y \<noteq> t \<and> y
    \<notin> cond_fv pred_fv \<phi> in
 SCEx n (SCPred (Node t (MVar n)) \<and>sc E SCPred (Node t (MVar n)) \<U> (\<not>sc (
     SCPred (Node t (MVar n))) \<and>sc \<phi>))"
abbreviation SCAX ("AX") where "AX t \<phi> \<equiv> let n = SOME y. y \<noteq> t \<and> y
    \<notin> cond_fv pred_fv \<phi> in
 SCEx n (SCPred (Node t (MVar n)) \<and>sc A SCPred (Node t (MVar n)) \<W> (\<not>sc (
     SCPred (Node t (MVar n))) \<and>sc \<phi>))"
abbreviation SCEP ("EP") where "EP t \<phi> \<equiv> let n = SOME y. y \<noteq> t \<and> y
    \<notin> cond_fv pred_fv \<phi> in
 SCEx n (SCPred (Node t (MVar n)) \<and>sc E SCPred (Node t (MVar n)) \<B> (\<not>sc (
     SCPred (Node t (MVar n))) \<and>sc \<phi>))"
abbreviation SCAP ("AP") where "AP t \<phi> \<equiv> let n = SOME y. y \<noteq> t \<and> y
    \<notin> cond_fv pred_fv \<phi> in
 SCEx n (SCPred (Node t (MVar n)) \<and>sc A SCPred (Node t (MVar n)) \<B> (\<not>sc (
     SCPred (Node t (MVar n))) \<and>sc \<phi>))"
definition SCEX_t ("EXt") where "EXt t ty \<phi> \<equiv> let n = SOME y. y \<noteq> t \<
    and> y \<notin> type_fv ty \<and> y \<notin> cond_fv pred_fv \<phi> in
 SCEx n (\<not>sc (SCPred (Node t (MVar n))) \<and>sc SCPred (Out t ty (MVar n)) \<and>sc
     EX t (SCPred (Node t (MVar n)) \<and>sc \<phi>))"

end

context LLVM_threads begin

lemma get_points_upd [simp]: "get_points (C(t \<mapsto> (a, n, r))) = (get_points C)(t \<
    mapsto> n)"
by (rule ext, simp add: map_comp_def)
```

```
lemma start_states_points [simp]: "get_points (\<lambda>t. case CFGs t of None \<Rightarrow
    > None | Some G \<Rightarrow>
 Some (Start G, Start G, start_env, [], {})) = start_points CFGs"
by (clarsimp intro!: ext simp add: map_comp_def start_points_def split: option.splits)

lemma start_points_dom1 [simp]: "dom (start_points CFGs) = dom CFGs"
by (auto intro!: set_eqI simp add: start_points_def split: option.splits)

lemma run_finite_dom [simp]: "run_prog CFGs C \<Longrightarrow> finite (dom (get_points (
    fst C)))"
apply (drule run_prog_path)
apply unfold_locales
apply (clarsimp, drule_tac i=i in Path_domain, simp+)
done

end

(* Alias analysis *)
locale alias_analysis = LLVM_base where can_read="can_read::'memory \<Rightarrow> 'thread
    \<Rightarrow> int \<Rightarrow>
  int LLVM_const set" for can_read +
 fixes cannot_alias::"('thread, 'node, string) LLVM_tCFG \<Rightarrow> 'node \<Rightarrow>
     'thread \<Rightarrow>
                      (string,int) LLVM_expr \<Rightarrow> (string,int) LLVM_expr \<
                          Rightarrow> bool"
 assumes AA_correct: "\<lbrakk>cannot_alias CFGs p t e1 e2; fst C t = Some (p0, p, env,
     rest);
 LLVM_threads free_set CFGs update_mem; run_prog CFGs C\<rbrakk> \<Longrightarrow>
  eval_expr env gt e1 \<noteq> eval_expr env gt e2"
    and AA_id: "\<not>cannot_alias CFGs p t e e"
    and AA_global [simp, intro!]: "\<lbrakk>Global_Decl x a \<in> set decls; Global_Decl y
        b \<in> set decls;
  x \<noteq> y\<rbrakk> \<Longrightarrow> cannot_alias CFGs p t (Global x) (Global y)"
begin

abbreviation "may_alias CFGs n t e1 e2 \<equiv> \<not>cannot_alias CFGs n t e1 e2"

end
```

```
sublocale LLVM_base \<subseteq> no_alias!: alias_analysis where cannot_alias="\<lambda>CFGs
     q t e1 e2.
 case (e1, e2) of (Global x, Global y) \<Rightarrow> (\<exists>a b. Global_Decl x a \<in>
     set decls \<and>
 Global_Decl y b \<in> set decls \<and> x \<noteq> y) | _ \<Rightarrow> False"
apply (unfold_locales, auto split: LLVM_expr.splits)
apply (clarsimp simp add: run_prog_def)
apply (drule global_diff, simp+)
done

locale LLVM_alias = LLVM_threads where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
 alias_analysis where cannot_alias="cannot_alias::('thread, 'node, string) LLVM_tCFG \<
     Rightarrow>
 'node \<Rightarrow> 'thread \<Rightarrow> (string,int) LLVM_expr \<Rightarrow> (string,int
     ) LLVM_expr \<Rightarrow> bool" for CFGs cannot_alias

(* Lock/mutex analysis *)
locale mutex_analysis = alias_analysis where cannot_alias="cannot_alias::('thread, 'node,
    string) LLVM_tCFG \<Rightarrow>
 'node \<Rightarrow> 'thread \<Rightarrow> (string,int) LLVM_expr \<Rightarrow> (string,int
     ) LLVM_expr \<Rightarrow> bool" for cannot_alias +
 fixes in_critical::"('thread, 'node, string) LLVM_tCFG \<Rightarrow> 'node \<Rightarrow> '
     thread \<Rightarrow>
 (string,int) LLVM_expr \<Rightarrow> (string,int) LLVM_expr \<Rightarrow> bool"
   and protected::"('thread, 'node, string) LLVM_tCFG \<Rightarrow> (string,int) LLVM_expr
       \<Rightarrow> (string,int) LLVM_expr \<Rightarrow> bool"
 assumes mutex: "\<lbrakk>protected CFGs x l; in_critical CFGs n t x l; in_critical CFGs n'
      t' x l;
 fst C t = Some (p, n, r); fst C t' = Some (p', n', r'); LLVM_threads free_set CFGs
     update_mem;
 run_prog CFGs C\<rbrakk> \<Longrightarrow> t = t'"
     and critical_protects: "\<lbrakk>protected CFGs x l; LLVM_threads free_set CFGs
         update_mem;
     run_prog CFGs C; fst C t = Some (p0, p, env, rest); step t G (snd C) (p0, p, env, rest
         ) ops s';
     CFGs t = Some G; eval_expr env gt l = CPointer l'; l' \<in> get_loc ` ops\<rbrakk> \<
         Longrightarrow> in_critical CFGs p t x l"
(* Translation validation approach: instead of transfer axioms, RSE THEN a transformation
    that fails
   unless protected still holds in each opt? *)
```

```
context alias_analysis begin

definition "in_critical CFGs p t l \<equiv> \<exists>C G env rest ops s' l' p0. run_prog
    CFGs C \<and> CFGs t = Some G \<and>
 fst C t = Some (p0, p, env, rest) \<and> step t G (snd C) (p0, p, env, rest) ops s' \<and>
 eval_expr env gt l = CPointer l' \<and> l' \<in> get_loc ` ops"

definition "protected CFGs l \<equiv> \<forall>p0 p p0' p' t t' r r' C. in_critical CFGs p
    t l \<and>
 in_critical CFGs p' t' l \<and> fst C t = Some (p0, p, r) \<and> fst C t' = Some (p0', p',
      r') \<and>
 run_prog CFGs C \<longrightarrow> t = t'"

end

sublocale alias_analysis \<subseteq> base!: mutex_analysis where in_critical="\<lambda>CFGs
     n t xs. in_critical CFGs n t"
 and protected="\<lambda>CFGs xs. protected CFGs"
apply unfold_locales
apply (smt protected_def)
apply (smt in_critical_def)
done

datatype ('mvar, 'expr) analysis_pred = CannotAlias 'mvar 'expr 'expr
 | InCritical 'mvar 'expr 'mvar | Protected 'expr 'mvar

primrec alias_fv where
"alias_fv (CannotAlias t e1 e2) = insert t (expr_pattern_fv expr_fv e1) \<union>
    expr_pattern_fv expr_fv e2" |
"alias_fv (InCritical t e x) = {t, x} \<union> expr_pattern_fv expr_fv e" |
"alias_fv (Protected e x) = insert x (expr_pattern_fv expr_fv e)"

lemma alias_fv_finite: "finite (alias_fv p)"
by (case_tac p, auto)

type_synonym 'mvar LLVM_ext_pred = "('mvar, (LLVM_edge_type, 'mvar) literal, 'mvar pattern,
  ('mvar, ('mvar, ('mvar, ('mvar,int) LLVM_expr) expr_pattern) analysis_pred) ext_pred)
      pred"
```

```
locale LLVM_apreds = alias_analysis where cannot_alias="cannot_alias::('thread, 'node,
    string) LLVM_tCFG \<Rightarrow>
  'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string,
      int) LLVM_expr \<Rightarrow> bool" for cannot_alias +
  fixes eval_other::"('thread, 'node, string) LLVM_tCFG \<Rightarrow> ('thread \<
      rightharpoonup> 'node) \<Rightarrow>
    (string \<Rightarrow> ('thread, 'node, string) object) \<Rightarrow>
    (string, (string, (string, int) LLVM_expr) expr_pattern) analysis_pred \<Rightarrow>
        bool"
  assumes infinite_nodes: "infinite (UNIV::'node set)"
      and other_same_subst: "\<forall>x\<in>alias_fv r. \<sigma> x = \<sigma>' x \<
          Longrightarrow>
    eval_other CFGs q \<sigma> r = eval_other CFGs q \<sigma>' r"

sublocale LLVM_apreds \<subseteq> LLVM_preds where eval_other=eval_other
  and other_fv="alias_fv::(string, (string, (string, int) LLVM_expr) expr_pattern)
      analysis_pred \<Rightarrow> string set"
apply unfold_locales
apply (erule other_same_subst)
apply (rule alias_fv_finite)
apply (rule infinite_nodes)
done

context mutex_analysis begin

primrec eval_analysis where
"eval_analysis CFGs q \<sigma> (CannotAlias t e1 e2) = (case (\<sigma> t,
    expr_pattern_subst \<sigma> e1, expr_pattern_subst \<sigma> e2) of
 (OThread t', Some e1', Some e2') \<Rightarrow> (case q t' of Some n \<Rightarrow>
     cannot_alias CFGs n t' e1' e2' | _ \<Rightarrow> False) | _ \<Rightarrow> False)" |
"eval_analysis CFGs q \<sigma> (InCritical t e x) = (case (\<sigma> t, expr_pattern_subst
    \<sigma> e, \<sigma> x) of
 (OThread t', Some e', OExpr l) \<Rightarrow> (case q t' of Some n \<Rightarrow>
     in_critical CFGs n t' e' l | _ \<Rightarrow> False) | _ \<Rightarrow> False)" |
"eval_analysis CFGs q \<sigma> (Protected e x) = (case (expr_pattern_subst \<sigma> e, \<
    sigma> x) of
 (Some e', OExpr l) \<Rightarrow> protected CFGs e' l | _ \<Rightarrow> False)"

lemma eval_analysis_same_subst: "\<forall>x\<in>alias_fv a. \<sigma> x = \<sigma>' x \<
    Longrightarrow>
```

```
 eval_analysis CFGs' q \<sigma> a = eval_analysis CFGs' q \<sigma>' a"
apply (cases a, simp_all)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr2 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr in
    expr_pattern_same_subst, simp+)+
done

end

print_locale mutex_analysis
locale LLVM_mutex = mutex_analysis where cannot_alias="cannot_alias::('thread, 'node,
    string) LLVM_tCFG \<Rightarrow>
  'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string,
      int) LLVM_expr \<Rightarrow> bool" for cannot_alias +
  assumes infinite_nodes: "infinite (UNIV::'node set)"

sublocale LLVM_mutex \<subseteq> LLVM_apreds where eval_other=eval_analysis
apply unfold_locales
apply (rule infinite_nodes)
apply (erule eval_analysis_same_subst)
done

(* useful defined predicates *)
abbreviation "Var x \<equiv> EPInj (Local x)"
definition "writes t x \<equiv> let vars = new {t, x} 6 in (SCExs vars (SCPred (Stmt t (Inj
 (Store (MVar (vars ! 0)) (EPInj (Local (vars ! 1))) (MVar (vars ! 2)) (EPInj (Local x)))))
      \<or>sc
 SCPred (Stmt t (Inj (Cmpxchg (vars ! 0) (MVar (vars ! 1)) (EPInj (Local x)) (MVar (vars !
     2))
 (EPInj (Local (vars ! 3))) (MVar (vars ! 4)) (EPInj (Local (vars ! 5))))))))"
abbreviation "def t x \<equiv> writes t x"
definition "stores t x \<equiv> let vars = new ({t} \<union> expr_pattern_fv expr_fv x) 3
    in
 (SCExs vars (SCPred (Stmt t (Inj (Store (MVar (vars ! 0)) (Var (vars ! 1)) (MVar (vars !
     2)) x)))))"
definition "not_touches t x \<equiv> let e = hd (new ({t} \<union> expr_pattern_fv expr_fv
    x) 1) in
```

```
 let vars = new ({t, e} \<union> expr_pattern_fv expr_fv x) 6 in
 (SCAll e ((SCExs vars ((stmt t (Store (MVar (vars ! 0)) (Var (vars ! 1)) (MVar (vars ! 2))
      (Var e))) \<or>sc
  stmt t (Cmpxchg (vars ! 0) (MVar (vars ! 1)) (Var e) (MVar (vars ! 2)) (Var (vars ! 3)) (
      MVar (vars ! 4)) (Var (vars ! 5))) \<or>sc
  stmt t (Load (vars ! 0) (MVar (vars ! 1)) (Var e)))) \<longrightarrow>sc
 SCPred (Other (Ext (CannotAlias t (Var e) x)))))"
definition "not_loads t x \<equiv> let e = hd (new ({t} \<union> expr_pattern_fv expr_fv x)
     1) in
 let vars = new ({t, e} \<union> expr_pattern_fv expr_fv x) 2 in
 (SCAll e ((SCExs vars (stmt t (Load (vars ! 0) (MVar (vars ! 1)) (Var e)))) \<
     longrightarrow>sc
 SCPred (Other (Ext (CannotAlias t (Var e) x)))))"
definition "mods t x \<equiv> SCPred (Other (Base (Def t x)))"
(* These aren't quite inverses, because call and return change all local variables. *)
definition "not_mods t x \<equiv> \<not>sc (mods t x) \<and>sc \<not>sc (SCPred (Other (
    Base (IsCall t)))) \<and>sc \<not>sc (SCPred (Other (Base (IsRet t))))"
definition "returns t \<equiv> let vars = new {t} 2 in SCExs vars (stmt t (Ret (MVar (vars
    ! 0)) (Var (vars ! 1))))"

definition "read t x \<equiv> let vars = new {t, x} 6 in
 SCExs vars (SCPred (Stmt t (Inj (Load (vars ! 0) (vars ! 1) x))) \<or>sc
 SCPred (Stmt t (Inj (Cmpxchg (vars ! 0) (vars ! 1) x (vars ! 2) (vars ! 3) (vars ! 4) (
     vars ! 5)))))"

context TRANS_basics begin

lemma exs_simp [simp]: "models CFGs \<sigma> q (SCExs l (P l)) =
 (\<exists>objs. length objs = length l \<and> models CFGs (update_list \<sigma> (zip l
     objs)) q (P l))"
apply (induct l arbitrary: P \<sigma>, auto)
apply (subgoal_tac "\<exists>objs. length objs = length l \<and>
 models CFGs (update_list (\<sigma>(a := obj)) (zip l objs)) q ((\<lambda>l. P (a # l)) l)
     ", clarify)
apply (rule_tac x="obj # objs" in exI, simp)
apply force
apply (case_tac objs, auto)
apply force
done
```

```
lemma models_ex_disjE [elim]: "\<lbrakk>models CFGs \<sigma> q (SCExs l (P \<or>sc Q));
    models CFGs \<sigma> q (SCExs l P) \<Longrightarrow> R;
 models CFGs \<sigma> q (SCExs l Q) \<Longrightarrow> R\<rbrakk> \<Longrightarrow> R"
by (induct l arbitrary: \<sigma>, auto)

end

context alias_analysis begin
(* Concrete lock analysis example *)

primrec eval_a_only where
"eval_a_only CFGs q \<sigma> (CannotAlias t e1 e2) = (case (\<sigma> t, expr_pattern_subst
    \<sigma> e1, expr_pattern_subst \<sigma> e2) of
 (OThread t', Some e1', Some e2') \<Rightarrow> (case q t' of Some n \<Rightarrow>
     cannot_alias CFGs n t' e1' e2' | _ \<Rightarrow> False) | _ \<Rightarrow> False)" |
"eval_a_only CFGs q \<sigma> (InCritical t e x) = False" |
"eval_a_only CFGs q \<sigma> (Protected e x) = False"

lemma eval_a_only_same_subst: "\<forall>x\<in>alias_fv p. \<sigma> x = \<sigma>' x \<
    Longrightarrow>
 eval_a_only CFGs' q \<sigma> p = eval_a_only CFGs' q \<sigma>' p"
apply (case_tac p, simp_all)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr1 in
    expr_pattern_same_subst, simp+)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'=\<sigma>' and e=expr2 in
    expr_pattern_same_subst, simp+)
done

end

locale LLVM_a_only = alias_analysis where cannot_alias="cannot_alias::('thread, 'node,
    string) LLVM_tCFG \<Rightarrow>
  'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string,
      int) LLVM_expr \<Rightarrow> bool" for cannot_alias +
  assumes infinite_nodes: "infinite (UNIV::'node set)"

sublocale LLVM_a_only \<subseteq> LLVM_apreds where eval_other=eval_a_only
apply unfold_locales
apply (rule infinite_nodes)
apply (erule eval_a_only_same_subst)
```

```
done

context LLVM_apreds begin

abbreviation SCEX' ("EX") where "EX \<equiv> SCEX"
(*abbreviation "EX \<equiv> SCEX" doesn't work, presumably because of conflict with EX x. P
    .
 This is a strange workaround. *)
abbreviation "EP \<equiv> SCEP"
abbreviation "AX \<equiv> SCAX"
abbreviation "AP \<equiv> SCAP"
abbreviation "EXt \<equiv> SCEX_t"

definition "locks t x \<equiv> let vars = new {t, x} 5 in SCExs vars (stmt t (Br_i1 (Var (
    vars ! 0))) \<and>sc
 EXt t (Inj true) (node t (vars ! 4) \<and>sc stmt t (Cmpxchg (vars ! 0) (MVar (vars ! 1))
     (Var x)
 (MVar (vars ! 2)) (EPInj (Const (CInt 0))) (MVar (vars ! 3)) (EPInj (Const (CInt 1))))) \<
     and>sc
 AP t (node t (vars ! 4)))"
definition "locks2 t x \<equiv> let vars = new {t, x} 5 in SCExs vars ((node t (vars ! 4)
    \<and>sc
 stmt t (Cmpxchg (vars ! 0) (MVar (vars ! 1)) (Var x) (MVar (vars ! 2)) (EPInj (Const (CInt
      0)))
 (MVar (vars ! 3)) (EPInj (Const (CInt 1))))) \<and>sc EX t (stmt t (Br_i1 (Var (vars ! 0))
     ) \<and>sc
 SCPred (Out t (Inj true) (MVar (vars ! 4)))))"
(* As of now, locks2 does not imply locks (since it doesn't ensure no other entry. *)
definition "unlocks t x \<equiv> let vars = new {t, x} 2 in SCExs vars
 (stmt t (Store (MVar (vars ! 0)) (EPInj (Const (CInt 0))) (MVar (vars ! 1)) (Var x)))"

(* A point is in the critical section if all paths to it are locked. *)
abbreviation "locked t x \<equiv> A \<not>sc (unlocks t x) \<B> locks t x"
definition "in_critical_lock t x \<equiv> AP t (locked t x)"

(* A lock is good if it isn't used as a regular variable, it is only unlocked
   after being locked, and every locked section is critical (i.e., we can't make
   unsafe jumps into locked sections). *)
definition "good_lock t x \<equiv> SCPred (Other (Base (GVarlit (Var x)))) \<and>sc AG
```

```
 (((\<not>sc (locks2 t x) \<and>sc \<not>sc (unlocks t x)) \<longrightarrow>sc not_touches
     t (Var x)) \<and>sc
 (unlocks t x \<longrightarrow>sc in_critical_lock t x) \<and>sc ((EP t (locked t x)) \<
     longrightarrow>sc in_critical_lock t x))"

definition "protected_l x l \<equiv> let t = SOME y. y \<noteq> x \<and> y \<notin>
    expr_pattern_fv expr_fv l in
 SCAll t (good_lock t x \<and>sc SCPred (Other (Base (GVarlit l))) \<and>sc
 AG (\<not>sc (in_critical_lock t x) \<longrightarrow>sc not_touches t l))"

abbreviation "in_critical_l t x l \<equiv> in_critical_lock t x"

end

lemma tCFG [simp]: "LLVM_tCFG CFGs \<Longrightarrow> tCFG CFGs LLVM_instr_edges seq"
by (simp add: LLVM_tCFG_def)

context LLVM_preds begin

lemma by_thread_EXt [intro!]: "by_thread P t \<Longrightarrow> by_thread (EXt t ty P) t"
apply (clarsimp simp add: SCEX_t_def)
apply (rule by_thread_ex, safe intro!: by_thread_not by_thread_node)
apply (cut_tac A="{t, SOME y. y \<noteq> t \<and> y \<notin> type_fv ty \<and> y \<notin>
    cond_fv pred_fv P} \<union>
 cond_fv pred_fv P" in fresh_new, simp+)
apply (cut_tac A="insert t (type_fv ty) \<union> cond_fv pred_fv P" in fresh_new, simp+)
done

end

context LLVM_apreds begin

lemma locks_by_thread: "by_thread (locks t x) t"
apply (clarsimp simp add: locks_def Let_def)
apply (rule by_thread_exs, auto simp add: new_nodes_are_new2)
apply (cut_tac m=t and S="{t, new {t, x} 5 ! 4}" and n=1 in new_nodes_are_new2, simp+)
done

lemma locks2_by_thread: "by_thread (locks2 t x) t"
apply (clarsimp simp add: locks2_def Let_def)
```

```
apply (rule by_thread_exs, auto simp add: new_nodes_are_new2)
apply (cut_tac A="{t, new {t, x} 5 ! 4, t, new {t, x} 5 ! 0}" in fresh_new, simp+)
done

lemma unlocks_by_thread: "by_thread (unlocks t x) t"
apply (clarsimp simp add: unlocks_def Let_def)
apply (rule by_thread_exs, auto)
apply (cut_tac m=t and S="{t, x}" and n=2 in new_nodes_are_new2, simp+)
done

lemma in_critical_by_thread: "by_thread (in_critical_lock t x) t"
apply (clarsimp simp add: in_critical_lock_def Let_def)
apply (rule by_thread_ex, auto simp add: locks_by_thread unlocks_by_thread)
apply (cut_tac A="insert t (cond_fv pred_fv (unlocks t x)) \<union>
 cond_fv pred_fv (locks t x)" in fresh_new, auto)
done

end

context LLVM_a_only begin

lemma cannot_alias_by_thread [intro!]: "by_thread (SCPred (Other (Ext (CannotAlias t e e'))
    )) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "expr_pattern_subst \<sigma> e", simp+)
apply (case_tac "expr_pattern_subst \<sigma> e'", simp+)
by (metis option.distinct(1) option.inject)

lemma not_touches_by_thread [intro!]: "by_thread (not_touches t x) t"
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x) (expr_pattern_fv expr_fv x))"
 and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac A="insert t (expr_pattern_fv expr_fv x)" in fresh_new, simp+)
apply (auto simp add: not_touches_def Let_def)
done

lemma (in LLVM_mutex) cannot_alias_by_thread [intro!]: "by_thread (SCPred (Other (Ext (
    CannotAlias t e e')))) t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "expr_pattern_subst \<sigma> e", simp+)
```

```
apply (case_tac "expr_pattern_subst \<sigma> e'", simp+)
by (metis option.distinct(1) option.inject)

lemma (in LLVM_mutex) not_touches_by_thread [intro!]: "by_thread (not_touches t x) t"
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x) (expr_pattern_fv expr_fv x))"
 and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac A="insert t (expr_pattern_fv expr_fv x)" in fresh_new, simp+)
apply (auto simp add: not_touches_def Let_def)
done

lemma locks_fv [simp]: "cond_fv pred_fv (locks t x) = {t, x}"
apply (auto simp add: locks_def Let_def SCEX_t_def new_nodes_are_new2)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! 1, new {t, x} 5
     ! 0, t,
 new {t, x} 5 ! 4, t, SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 3 \<and> y \<
     noteq> new {t, x} 5 ! 2 \<and> y \<noteq> x \<and>
 y \<noteq> new {t, x} 5 ! Suc 0 \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t \<
     and> y \<noteq> new {t, x} 5 ! 4}"
 in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! 1, new {t, x} 5
     ! 0, t,
 new {t, x} 5 ! 4}" in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! 1, new {t, x} 5
     ! 0, t,
 new {t, x} 5 ! 4, t, SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 3 \<and> y \<
     noteq> new {t, x} 5 ! 2 \<and> y \<noteq> x \<and>
 y \<noteq> new {t, x} 5 ! Suc 0 \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t \<
     and> y \<noteq> new {t, x} 5 ! 4}"
 in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! 1, new {t, x} 5
     ! 0, t,
 new {t, x} 5 ! 4}" in fresh_new, simp+)
done

lemma locks2_fv [simp]: "cond_fv pred_fv (locks2 t x) = {t, x}"
by (auto simp add: locks2_def Let_def new_nodes_are_new2)

lemma unlocks_fv [simp]: "cond_fv pred_fv (unlocks t x) = {t, x}"
by (auto simp add: unlocks_def Let_def new_nodes_are_new2)
```

```
lemma not_touches_fv [simp]: "cond_fv pred_fv (not_touches t e) =
 insert t (expr_pattern_fv expr_fv e)"
apply (cut_tac A="insert t (expr_pattern_fv expr_fv e)" in fresh_new)
apply (auto simp add: not_touches_def Let_def)
done

lemma in_critical_fv [simp]: "cond_fv pred_fv (in_critical_lock t x) = {t, x}"
apply (cut_tac A="{t, x}" in fresh_new)
apply (auto simp add: in_critical_lock_def Let_def)
done

lemma good_lock_fv [simp]: "cond_fv pred_fv (good_lock t x) = {t, x}"
by (auto simp add: good_lock_def Let_def)

lemma locks_br: "models CFGs \<sigma> q (locks t x) \<Longrightarrow> \<exists>t' e n G. \<
    sigma> t = OThread t' \<and> q t' = Some n \<and>
 CFGs t' = Some G \<and> Label G n = Br_i1 e"
apply (clarsimp simp add: locks_def Let_def)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (erule_tac x=ya in allE, simp)
apply (cut_tac S="{t, x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=t in new_nodes_are_new2, simp+)
apply (clarsimp split: object.splits)
done

lemma locks2_cmpxchg: "models CFGs \<sigma> q (locks2 t x) \<Longrightarrow> \<exists>t' e
    n G v ty1 ty2 ty3. \<sigma> t = OThread t' \<and>
 \<sigma> x = OExpr e \<and> q t' = Some n \<and> CFGs t' = Some G \<and>
 Label G n = Cmpxchg v ty1 e ty2 (Const (CInt 0)) ty3 (Const (CInt 1))"
apply (clarsimp simp add: locks2_def Let_def)
apply (cut_tac A="{t, new {t, x} 5 ! 0, new {t, x} 5 ! 4, SOME y. y \<noteq> t \<and> y \<
    noteq> new {t, x} 5 ! 4 \<and> y \<noteq> t}"
 in fresh_new, simp+)
apply (cut_tac S="{t, x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (clarsimp simp only: id_def)
apply clarsimp
```

```
apply (case_tac "\<sigma> t", simp_all)
apply (case_tac "objs ! 0", simp_all)
apply (case_tac LLVM_expr, simp_all)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "\<sigma> x", simp_all)
apply (case_tac "objs ! 2", simp_all)
apply (case_tac "objs ! 3", simp_all, clarsimp)
by (metis (hide_lams, no_types))

end

locale LLVM_a_only_CFG = LLVM_a_only where cannot_alias="cannot_alias::('thread, 'node,
    string) LLVM_tCFG \<Rightarrow>
  'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string,
      int) LLVM_expr \<Rightarrow> bool" +
  LLVM_tCFG where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" for cannot_alias CFGs
      begin

lemma tCFG [simp]: "tCFG CFGs LLVM_instr_edges seq"
by unfold_locales

lemma locks_gen: "\<lbrakk>models CFGs \<sigma> q (locks t x); \<sigma>' t = \<sigma> t; \<
    sigma>' x = \<sigma> x\<rbrakk> \<Longrightarrow>
  models CFGs \<sigma>' q (locks t x)"
by (erule side_cond_gen, simp_all)

lemma locks2_gen: "\<lbrakk>models CFGs \<sigma> q (locks2 t x); \<sigma>' t = \<sigma> t;
    \<sigma>' x = \<sigma> x\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q (locks2 t x)"
by (erule side_cond_gen, simp_all)

lemma unlocks_gen: "\<lbrakk>models CFGs \<sigma> q (unlocks t x); \<sigma>' t = \<sigma> t
    ; \<sigma>' x = \<sigma> x\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q (unlocks t x)"
by (erule side_cond_gen, simp_all)

lemma locks_in: "\<lbrakk>\<not>models CFGs \<sigma> q (in_critical_lock t x); run_prog
    CFGs (C, mem); q t' = Some n;
 C t' = Some (p0, n, rest); \<sigma> t = OThread t'; CFGs t' = Some G; step t' G mem (p0, n
     , rest) ops (n, n', rest');
```

```
 models CFGs \<sigma> (q(t' \<mapsto> n')) (in_critical_lock t x)\<rbrakk> \<Longrightarrow
     > models CFGs \<sigma> q (locks t x)"
apply (case_tac "n' = n", force simp add: map_upd_triv)
apply (clarsimp simp add: in_critical_lock_def Let_def)
apply (cut_tac A="{t, x}" in fresh_new, simp+, clarsimp)
apply (erule_tac x="ONode n" in allE, clarsimp)
apply (case_tac rest, clarsimp)
apply (frule run_prog_safe, simp)
apply (case_tac "\<exists>i'. l i' t' \<noteq> Some n \<and> (\<forall>j<i'. l j t' = Some
    n)", clarsimp)
apply (erule_tac x=i' in allE, clarsimp, erule disjE, clarsimp)
apply (frule combine_rpaths, simp+)
apply (frule_tac l="l \<Down> i' \<frown> la" and CFGs=CFGs in step_increment_rpath, simp+)
apply (clarsimp simp add: safe_points_def, erule_tac x=t' in allE, clarsimp)
apply (erule_tac x="[q(t' \<mapsto> n')] \<frown> l \<Down> i' \<frown> la" in ballE,
    simp_all, clarsimp)
apply (case_tac obj, simp_all, clarsimp)
apply (case_tac i, clarsimp+)
apply (case_tac nat, clarsimp)
apply (erule_tac x="l \<Down> i' \<frown> la" in ballE, simp_all, clarsimp)
apply (case_tac "i < i'", rule_tac q="l i" in by_threadD [OF locks_by_thread], simp_all,
    simp+)
apply (erule rpath_suffix)
apply unfold_locales
apply (erule locks_gen, simp+)
apply (erule_tac x="i - i'" in allE, erule impE, erule locks_gen, simp+, clarsimp)
apply (drule_tac \<sigma>'="\<sigma>(SOME y. y \<noteq> t \<and> y \<noteq> x := ONode n')"
     in unlocks_gen, simp+)
apply (thin_tac "\<forall>j<i'. l j t' = Some n")
apply (erule_tac x="j + i'" in allE, simp)
apply clarsimp
apply ((erule_tac x=1 in allE)+, simp)
apply metis
apply (case_tac "\<exists>i. l i t' \<noteq> Some n", clarsimp, frule_tac P="\<lambda>i. l
    i t' \<noteq> Some n" in LeastI,
 (erule_tac x="LEAST i. l i t' \<noteq> Some n" in allE)+, clarsimp)
apply (drule_tac P="\<lambda>i. l i t' \<noteq> Some n" and k=j in Least_le, simp)
apply (frule_tac CFGs=CFGs in step_increment_rpath, simp+)
apply (clarsimp simp add: safe_points_def, erule_tac x=t' in allE, clarsimp)
apply (erule_tac x="[q(t' \<mapsto> n')] \<frown> l" in ballE, simp_all, clarsimp)
```

```
apply (case_tac obj, simp_all, clarsimp)
apply (case_tac i, simp+)
apply (case_tac nat, clarsimp)
apply (erule_tac x=l in ballE, simp_all, clarsimp)
apply (rule_tac q="l i" in by_threadD [OF locks_by_thread], simp+)
apply (erule rpath_suffix, simp+)
apply unfold_locales
apply (erule locks_gen, simp+)
apply ((erule_tac x=1 in allE)+, simp)
done

lemma unlocks_store: "models CFGs \<sigma> q (unlocks t x) \<Longrightarrow> \<exists>t' e
    n G ty1 ty2. \<sigma> t = OThread t' \<and>
 \<sigma> x = OExpr e \<and> q t' = Some n \<and> CFGs t' = Some G \<and> Label G n = Store
      ty1 (Const (CInt 0)) ty2 e"
apply (clarsimp simp add: unlocks_def Let_def)
apply (cut_tac S="{t, x}" and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="{t, x}" and n=2 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="{t, x}" and n=2 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (case_tac "objs ! 0", simp_all)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "\<sigma> x", simp_all, clarsimp split: object.splits)
done

lemma locks_locks2: "models CFGs \<sigma> q (locks t x) \<Longrightarrow> models CFGs \<
    sigma> q (AP t (locks2 t x))"
apply (clarsimp simp add: locks_def SCEX_t_def)
apply (cut_tac S="{t, x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 4}" in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! 1, new {t, x} 5
     ! 0, t,
 new {t, x} 5 ! 4}" in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! 1, new {t, x} 5
     ! 0, t,
 new {t, x} 5 ! 4, t, SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 3 \<and> y \<
     noteq> new {t, x} 5 ! 2 \<and> y \<noteq> x \<and>
 y \<noteq> new {t, x} 5 ! Suc 0 \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t \<
     and> y \<noteq> new {t, x} 5 ! 4}"
 in fresh_new, simp+)
```

```
apply (cut_tac S="{t, x}" and n=5 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac A="{t, SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 3 \<and> y \<
    noteq> new {t, x} 5 ! 2 \<and> y \<noteq> x \<and>
 y \<noteq> new {t, x} 5 ! Suc 0 \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t \<
     and> y \<noteq> new {t, x} 5 ! 4}"
 in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 4, t, new {t, x} 5 ! 0}" in fresh_new, simp+)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply clarsimp
apply (case_tac "objs ! 0", simp_all)
apply (case_tac a, simp_all, clarsimp)
apply (rule_tac x="ONode ya" in exI, clarsimp)
apply (cut_tac A="{t, x}" in fresh_new, simp+)
apply (rule conjI, clarsimp+)
apply (case_tac obj, simp_all, case_tac obja, simp_all, case_tac objb, simp_all)
apply (case_tac "\<sigma> t", simp_all)
apply (cut_tac P="\<lambda>y. y \<noteq> t \<and> (y \<noteq> x \<and> y \<noteq> new {t, x
    } 5 ! Suc 0 \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t \<and>
 y \<noteq> new {t, x} 5 ! 4 \<and> y \<noteq> t \<and> (y \<noteq> new {t, x} 5 ! 4 \<and>
      y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 0 \<and>
 y \<noteq> t \<and> y \<noteq> (SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 4 \<
     and> y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t) \<and>
 y \<noteq> t \<or> y = (SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 4 \<and> y
     \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t)) \<or>
 y \<in> set (new {t, x} 5))" and x="SOME y. y \<notin> {t, x, new {t, x} 5 ! 1, new {t, x}
      5 ! 0,
 new {t, x} 5 ! 4, SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 4 \<and> y \<noteq
     > t \<and> y \<noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t}"
 in someI, cut_tac A="{t, x, new {t, x} 5 ! 1, new {t, x} 5 ! 0, new {t, x} 5 ! 4,
 SOME y. y \<noteq> t \<and> y \<noteq> new {t, x} 5 ! 4 \<and> y \<noteq> t \<and> y \<
     noteq> new {t, x} 5 ! 0 \<and> y \<noteq> t}" in fresh_new,
 simp+, clarsimp)
apply (erule_tac x=la in ballE, simp_all, clarsimp)
apply (rule_tac x=ia in exI, clarsimp simp add: locks2_def)
apply (rule subst, rule sym, rule exs_simp, simp)
apply (rule_tac x=objs in exI, clarsimp)
apply (rule_tac x="objs ! 4" in exI, clarsimp)
apply (frule_tac l=la and i=ia in reverse_rpath, clarsimp)
apply (rule_tac x=l' in bexI, simp_all, rule_tac x=1 in exI, clarsimp)
```

```
apply (case_tac ia, simp+)
done

lemma unlocks_out: "\<lbrakk>models CFGs \<sigma> q (unlocks t x); run_prog CFGs (C, mem);
    get_points C = q;
 C t' = Some (p0, n, rest); \<sigma> t = OThread t'; CFGs t' = Some G; step t' G mem (p0, n
     , rest) ops (n, n', rest')\<rbrakk> \<Longrightarrow>
 \<not>models CFGs \<sigma> (q(t' \<mapsto> n')) (in_critical_lock t x)"
apply (clarsimp simp only: in_critical_lock_def Let_def)
apply clarsimp
apply (frule run_prog_safe, unfold_locales)
apply (drule run_prog_path, unfold_locales)
apply (clarsimp, drule_tac i=i in reverse_path, clarsimp)
apply (case_tac rest, frule_tac CFGs=CFGs in step_increment_rpath, simp+)
apply (simp add: safe_points_def, force, clarsimp)
apply (cut_tac A="{t, x}" in fresh_new, simp+)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x="[((\<lambda>C. \<lfloor>case C of (uu_, p, env, stack, allocad) \<
    Rightarrow> p\<rfloor>) \<circ>\<^sub>m C)(t' \<mapsto> n')] \<frown> l'"
  in ballE, simp_all, clarsimp)
apply (case_tac obj, simp_all)
apply (case_tac ia, simp+)
apply (case_tac nat, clarsimp)
apply (erule_tac x=l' in ballE, simp_all, clarsimp)
apply (case_tac ia, clarsimp)
apply (drule locks_br, drule unlocks_store, clarsimp+)
apply (erule_tac x=0 and P="\<lambda>j. j < Suc nat \<longrightarrow> ?P j" in allE, simp)
apply (drule_tac \<sigma>'="\<sigma>(SOME y. y \<noteq> t \<and> y \<noteq> x := ONode n')"
     in unlocks_gen, simp+)
apply (erule_tac x=0 in allE, erule_tac x=1 in allE, clarsimp)
apply (drule unlocks_store, clarsimp)
apply (rule step_cases')
apply (erule CFGs)
apply (simp+, simp_all, clarsimp)
apply (clarsimp simp add: safe_points_def)
apply (erule_tac x=t' in allE, clarsimp)
apply (drule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def flowgraph_axioms_def)
apply (frule pointed_graph.finite_edges, clarsimp simp add: pointed_graph_def)
apply ((erule_tac x=n' in allE)+, clarsimp simp add: out_edges_def)
by (metis (hide_lams, no_types) next_in)
```

```
lemma edge_out_unlocks: "\<lbrakk>models CFGs \<sigma> q (in_critical_lock t x);
 \<not>models CFGs \<sigma> (q(t' \<mapsto> n')) (in_critical_lock t x); run_prog CFGs (C,
     mem);
 models CFGs \<sigma> (start_points CFGs) (good_lock t x); get_points C = q;
 C t' = Some (p0, n, rest); \<sigma> t = OThread t'; CFGs t' = Some G; (n, n', e) \<in>
     Edges G\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q (unlocks t x)"
apply (case_tac "n = n'", force simp add: map_upd_triv)
apply (clarsimp simp add: good_lock_def)
apply (cut_tac q="get_points C(t' \<mapsto> n')" in exists_path, clarsimp)
apply (case_tac rest, clarsimp)
apply (frule path_incremental, force, simp+)
apply (simp add: map_upd_triv)
apply (frule run_prog_path, unfold_locales, clarsimp)
apply (drule_tac i=i in combine_paths, simp+)
apply (erule_tac x="(la \<Down> i @ [get_points C]) \<frown> l" in ballE, simp_all,
    erule_tac x="Suc i" in allE,
 clarsimp)
apply (clarsimp simp add: fun_upd_def Let_def)
apply (erule_tac x="ONode n'" in allE, cut_tac A="{t, x}" in fresh_new, simp+, clarsimp)
apply (frule run_prog_rpath, unfold_locales, clarsimp)
apply (frule rpath_incremental, force, simp+)
apply (simp add: fun_upd_def)
apply (erule_tac x="[\<lambda>x. if x = t' then Some n' else get_points C x] \<frown> lb"
    in ballE, simp_all)
apply (erule_tac x=1 in allE, clarsimp)
apply (thin_tac "\<not>?P", clarsimp simp add: in_critical_lock_def Let_def)
apply (case_tac obj, simp_all, clarsimp)
apply (erule_tac x=lc in ballE, simp_all, clarsimp)
apply (frule_tac l=lc and i=ia in rpath_suffix, erule_tac x="lc \<Up> ia" in ballE,
    simp_all, clarsimp)
apply (erule_tac x="ia + ib" in allE, erule impE)
apply (erule locks_gen, simp+, clarsimp)
apply (case_tac "j < ia", erule_tac x=j in allE, simp)
apply (rule_tac q="lc j" in by_threadD [OF unlocks_by_thread], simp_all)
apply (simp add: map_comp_def)
apply (erule rpath_suffix)
apply unfold_locales
apply (erule unlocks_gen, simp+)
```

```
apply ((erule_tac x="j - ia" in allE)+, simp)
apply (drule_tac \<sigma>'="\<sigma>(SOME y. y \<noteq> t \<and> y \<noteq> x := ONode n)"
    in unlocks_gen, simp+)
done

lemma prev_edge: "\<lbrakk>l \<in> RPaths q; l i t \<noteq> Some n; l i t = Some m; \<
    forall>j<i. l j t = Some n; q t = Some n\<rbrakk> \<Longrightarrow>
 \<exists>G e. CFGs t = Some G \<and> (m, n, e) \<in> Edges G"
apply (case_tac i, auto)
apply (erule_tac x=nat in allE, clarsimp)
apply (drule RPath_prev, simp+)
done

lemma run_back: "\<lbrakk>run_prog CFGs (C, mem); C t = Some (p, n, r); q t = Some n; l \<
    in> RPaths q;
 \<forall>l\<in>RPaths q. \<exists>i. l i t \<noteq> Some n \<and> l i t = Some m \<and>
     (\<forall>j<i. l j t = Some n)\<rbrakk> \<Longrightarrow>
 \<exists>C' mem' p' r'. run_prog CFGs (C', mem') \<and> C' t = Some (p', m, r')"
apply (drule run_prog_path2, unfold_locales, clarsimp)
apply (frule_tac i=i in reverse_path, clarsimp)
apply (drule_tac q="get_points C" and q'=q and t=t in rpath_by_thread, simp_all)
apply clarsimp
apply (erule_tac x=l'a in ballE, simp_all, clarsimp)
apply (frule_tac l=l'a in prev_edge, simp_all)
apply (case_tac ia, simp+)
apply (case_tac "Suc nat \<le> i")
apply (erule_tac x="Suc nat" and P="\<lambda>j. j \<le>i \<longrightarrow> ?P j" in allE,
    simp)
apply (erule_tac x="i - Suc nat" in allE, clarsimp, force simp add: map_comp_def)
apply clarsimp
apply (erule_tac x=i and P="\<lambda>j. j < Suc nat \<longrightarrow> ?P j" in allE,
    clarsimp simp add: start_points_def)
apply (drule CFGs, clarsimp simp add: is_flowgraph_def flowgraph_def, drule pointed_graph.
    start_first,
 clarsimp simp add: in_edges_def)
done

lemma start_not_in_critical [simp]: "\<lbrakk>\<sigma> t = OThread t'; CFGs t' = Some G\<
    rbrakk> \<Longrightarrow>
 \<not>models CFGs \<sigma> [t' \<mapsto> Start G] (in_critical_lock t x)"
```

```
apply (clarsimp simp only: in_critical_lock_def, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (frule start_one_rpath, erule_tac x="(\<lambda>i. [t' \<mapsto> Start G])" in ballE,
    simp_all)
done

lemma start_points_not_in_critical [simp]:
 "\<not>models CFGs \<sigma> (start_points CFGs) (in_critical_lock t x)"
apply (clarsimp simp only: in_critical_lock_def, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (cut_tac start_rpath)
apply (erule_tac x="\<lambda>i. start_points CFGs" in ballE, simp_all)
done

lemma start_not_locks [simp]: "\<lbrakk>\<sigma> t = OThread t'; CFGs t' = Some G\<rbrakk>
    \<Longrightarrow>
 \<not>models CFGs \<sigma> [t' \<mapsto> Start G] (locks t x)"
apply (clarsimp simp add: locks_def Let_def)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (case_tac obj, simp_all, clarsimp)
apply (frule_tac t=t' in start_one_rpath)
apply (erule_tac x="\<lambda>i. [t' \<mapsto> Start G]" in ballE, clarsimp)
apply (simp add: fun_upd_def)
done

lemma start_points_not_locks [simp]: "\<not>models CFGs \<sigma> (start_points CFGs) (locks
     t x)"
apply (clarsimp simp add: locks_def Let_def)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (case_tac obj, simp_all, clarsimp)
apply (cut_tac start_rpath)
apply (erule_tac x="\<lambda>i. start_points CFGs" in ballE, clarsimp+)
done

lemma EXt_bound: "\<lbrakk>models CFGs \<sigma> q (EXt t (Inj true) (node t n \<and>sc stmt
     t (Cmpxchg a (MVar b)
 (Var c) (MVar d) (EPInj (Const k)) (MVar e) (EPInj (Const k')))))); \<sigma> t = \<sigma>'
     t'; \<sigma> n = \<sigma>' n';
```

```
 \<sigma> a = \<sigma>' a'; \<sigma> b = \<sigma>' b'; \<sigma> c = \<sigma>' c'; \<sigma>
     d = \<sigma>' d'; \<sigma> e = \<sigma>' e'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q (EXt t' (Inj true) (node t' n' \<and>sc stmt t' (Cmpxchg a' (MVar
     b')
 (Var c') (MVar d') (EPInj (Const k)) (MVar e') (EPInj (Const k')))))"
apply (cut_tac A="{t, e, d, c, b, a, t, n}" in fresh_new, simp+)
apply (cut_tac A="{t', e', d', c', b', a', t', n'}" in fresh_new, simp+)
apply (cut_tac A="{t, e, d, c, b, a, t, n, t, (SOME y. y \<noteq> t \<and> y \<noteq> e \<
    and> y \<noteq> d \<and> y \<noteq> c \<and> y \<noteq> b \<and>
 y \<noteq> a \<and> y \<noteq> t \<and> y \<noteq> n)}" in fresh_new, simp+)
apply (cut_tac A="{t', e', d', c', b', a', t', n', t', (SOME y. y \<noteq> t' \<and> y \<
    noteq> e' \<and> y \<noteq> d' \<and>
 y \<noteq> c' \<and> y \<noteq> b' \<and> y \<noteq> a' \<and> y \<noteq> t' \<and> y \<
     noteq> n')}" in fresh_new, simp+, clarsimp)
apply (clarsimp simp add: SCEX_t_def Let_def)
done

lemma locks_bound: "\<lbrakk>models CFGs \<sigma> q (locks t x); t \<noteq> x; t' \<noteq>
    x\<rbrakk> \<Longrightarrow>
 models CFGs (\<sigma>(t' := \<sigma> t)) q (locks t' x)"
apply (cut_tac A="{t, x}" in fresh_new, simp+)
apply (cut_tac A="{t', x}" in fresh_new, simp+)
apply (cut_tac S="{t, x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 4}" in fresh_new, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 4, t, new {t, x} 5 ! 0}" in fresh_new, simp+)
apply (cut_tac S="{t', x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{t', x}" and n=5 and m=t' in new_nodes_are_new2, simp+)
apply (cut_tac A="{t', new {t', x} 5 ! 4}" in fresh_new, simp+)
apply (cut_tac A="{t', new {t', x} 5 ! 4, t', new {t', x} 5 ! 0}" in fresh_new, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac S="{t', x}" and n=5 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac A="{t, new {t, x} 5 ! 3, new {t, x} 5 ! 2, x, new {t, x} 5 ! Suc 0, new {t,
    x} 5 ! 0,
 t, new {t, x} 5 ! 4}" in fresh_new, simp+)
apply (cut_tac A="{t', new {t', x} 5 ! 3, new {t', x} 5 ! 2, x, new {t', x} 5 ! Suc 0, new
    {t', x} 5 ! 0,
 t', new {t', x} 5 ! 4}" in fresh_new, simp+)
apply (clarsimp simp add: locks_def Let_def)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
```

```
apply (rule subst, rule sym, rule exs_simp, clarsimp)
apply (rule_tac x=objs in exI, clarsimp)
apply (rule conjI)
apply (erule EXt_bound, simp+)
apply (rule_tac x=obj in exI, clarsimp)
apply (erule_tac x=l in ballE, simp_all, clarsimp)
apply (rule_tac x=i in exI, simp)
done

lemma locks2_bound: "\<lbrakk>models CFGs \<sigma> q (locks2 t x); t \<noteq> x; t' \<noteq
    > x\<rbrakk> \<Longrightarrow>
 models CFGs (\<sigma>(t' := \<sigma> t)) q (locks2 t' x)"
apply (clarsimp simp add: locks2_def)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp)
apply (clarsimp simp only: id_def)
apply (rule_tac x=objs in exI, simp (no_asm))
apply clarsimp
apply (erule_tac x=y in allE, clarsimp)
apply (cut_tac S="{t, x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{t', x}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="{t', x}" and n=5 and m=t' in new_nodes_are_new2, simp+)
apply (cut_tac S="{t, x}" and n=5 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac S="{t', x}" and n=5 and m=x in new_nodes_are_new2, simp+)
apply (case_tac "objs ! 0", simp_all, clarsimp)
apply (case_tac LLVM_expr, simp_all, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "\<sigma> x", simp_all)
apply (case_tac "objs ! 2", simp_all)
apply (case_tac "objs ! 3", simp_all)
apply (case_tac "objs ! 4", simp_all, clarsimp)
apply (rule_tac x=obj in exI, clarsimp)
apply (case_tac obj, simp_all)
apply (cut_tac A="{t, new {t, x} 5 ! 4, t, new {t, x} 5 ! 0}" in fresh_new, simp+)
apply (cut_tac A="{t', new {t', x} 5 ! 4, t', new {t', x} 5 ! 0}" in fresh_new, simp+,
    clarsimp)
apply (rule bexI, simp_all, rule_tac x=i in exI, clarsimp)
done
```

```
lemma unlocks_bound: "\<lbrakk>models CFGs \<sigma> q (unlocks t x); t \<noteq> x; t' \<
    noteq> x\<rbrakk> \<Longrightarrow>
 models CFGs (\<sigma>(t' := \<sigma> t)) q (unlocks t' x)"
apply (cut_tac A="{t, x}" in fresh_new, simp+)
apply (cut_tac A="{t', x}" in fresh_new, simp+)
apply (cut_tac S="{t, x}" and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="{t, x}" and n=2 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="{t', x}" and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="{t', x}" and n=2 and m=t' in new_nodes_are_new2, simp+)
apply (cut_tac S="{t, x}" and n=2 and m=x in new_nodes_are_new2, simp+)
apply (cut_tac S="{t', x}" and n=2 and m=x in new_nodes_are_new2, simp+)
apply (clarsimp simp add: unlocks_def Let_def)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp, clarsimp)
apply (rule_tac x=objs in exI, clarsimp)
done

lemma not_touches_bound: "\<lbrakk>models CFGs \<sigma> q (not_touches t x); t \<notin>
    expr_pattern_fv expr_fv x;
 t' \<notin> expr_pattern_fv expr_fv x\<rbrakk> \<Longrightarrow>
 models CFGs (\<sigma>(t' := \<sigma> t)) q (not_touches t' x)"
apply (cut_tac A="insert t (expr_pattern_fv expr_fv x)" in fresh_new, simp+)
apply (cut_tac A="insert t' (expr_pattern_fv expr_fv x)" in fresh_new, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)
 (expr_pattern_fv expr_fv x))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t' (insert (SOME n. n \<noteq> t' \<and> n \<notin>
    expr_pattern_fv expr_fv x)
 (expr_pattern_fv expr_fv x))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)
 (expr_pattern_fv expr_fv x))" and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t' (insert (SOME n. n \<noteq> t' \<and> n \<notin>
    expr_pattern_fv expr_fv x)
 (expr_pattern_fv expr_fv x))" and n=6 and m=t' in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)
 (expr_pattern_fv expr_fv x))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv x"
```

```
 in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t' (insert (SOME n. n \<noteq> t' \<and> n \<notin>
    expr_pattern_fv expr_fv x)
 (expr_pattern_fv expr_fv x))" and n=6 and m="SOME n. n \<noteq> t' \<and> n \<notin>
     expr_pattern_fv expr_fv x"
 in new_nodes_are_new2, simp+)
apply (clarsimp simp add: not_touches_def Let_def)
apply (erule_tac x=obj in allE, erule impE)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp)
apply (clarsimp simp only: id_def)
apply (rule_tac x=objs in exI, clarsimp+)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (case_tac obj, simp_all)
apply (cut_tac e=x and \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin>
    expr_pattern_fv expr_fv x := OExpr LLVM_expr)"
 and \<sigma>'="\<sigma>(t' := OThread ta, SOME n. n \<noteq> t' \<and> n \<notin>
     expr_pattern_fv expr_fv x := OExpr LLVM_expr)"
 in expr_pattern_same_subst, force)
apply (simp split: option.splits)
apply (rule_tac x=ta in exI, clarsimp)
apply (rule conjI, clarsimp+)
done

lemma in_critical_bound: "\<lbrakk>models CFGs \<sigma> q (in_critical_lock t x); t \<noteq
    > x; t' \<noteq> x\<rbrakk> \<Longrightarrow>
 models CFGs (\<sigma>(t' := \<sigma> t)) q (in_critical_lock t' x)"
apply (cut_tac A="{t, x}" in fresh_new, simp+)
apply (cut_tac A="{t', x}" in fresh_new, simp+)
apply (clarsimp simp add: in_critical_lock_def Let_def)
apply (rule_tac x=obj in exI, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (case_tac obj, simp_all, clarsimp)
apply (erule_tac x=l in ballE, simp_all, clarsimp)
apply (rule_tac x=i in exI, clarsimp)
apply (erule_tac x=la in ballE, simp_all, clarsimp)
apply (rule_tac x=ia in exI, rule conjI)
apply (drule_tac t'=t' in locks_bound, simp+)
apply (erule side_cond_gen, simp)
```

```
apply unfold_locales
apply (clarsimp, (erule_tac x=j in allE)+, drule_tac t'=t in unlocks_bound, simp+)
apply (cut_tac P="unlocks t x" and \<sigma>'="\<sigma>(SOME y. y \<noteq> t \<and> y \<
    noteq> x := ONode ya)" in side_cond_gen,
  simp_all, simp add: LLVM_tCFG_def, simp+)
done

lemma good_lock_bound: "\<lbrakk>models CFGs \<sigma> q (good_lock t x); t \<noteq> x; t'
    \<noteq> x\<rbrakk> \<Longrightarrow>
 models CFGs (\<sigma>(t' := \<sigma> t)) q (good_lock t' x)"
apply (cut_tac A="{t, x}" in fresh_new, simp+)
apply (cut_tac A="{t', x}" in fresh_new, simp+)
apply (clarsimp simp add: good_lock_def Let_def)
apply (erule_tac x=l in ballE, simp_all, erule_tac x=i in allE, clarsimp)
apply (rule conjI, clarsimp)
apply (erule disjE, drule_tac t'=t' in locks2_bound, simp+)
apply (erule disjE, erule unlocks_bound, simp+)
apply (drule_tac t'=t' in not_touches_bound, simp+)
apply (rule conjI, clarsimp)
apply (drule_tac t'=t in unlocks_bound, simp+)
apply (erule impE, erule side_cond_gen, simp+)
apply (erule in_critical_bound, simp+)
apply clarsimp
apply (thin_tac "?P \<or> ?Q \<or> ?R")
apply (case_tac obj, simp_all, clarsimp)
apply (case_tac "\<sigma> t", simp_all)
apply (erule disjE)
apply (erule_tac x="ONode node" in allE, clarsimp)
apply (erule_tac x=la and A="RPaths (l i)" in ballE, simp_all, erule_tac x=ia
 and P="\<lambda>i. ?P i \<or> ?Q i" in allE, clarsimp)
apply (erule disjE, clarsimp)
apply (rule_tac x=lb in bexI, simp_all, clarsimp)
apply (thin_tac "?P \<longrightarrow> ?Q")
apply (drule_tac t'=t in locks_bound, simp+)
apply (erule_tac x=ib in allE, erule impE, erule side_cond_gen, simp+, clarsimp+)
apply ((erule_tac x=j in allE)+, drule_tac t'=t' in unlocks_bound, simp+)
apply (drule_tac P="unlocks t' x" and \<sigma>'="\<sigma>(t' := OThread aa, SOME y. y \<
    noteq> t' \<and> y \<noteq> x := ONode node)"
 in side_cond_gen, simp add: LLVM_tCFG_def, simp+)
apply (rule_tac x=j in exI, simp)
```

```
apply clarsimp+
apply (drule_tac t'=t' in in_critical_bound, simp+)
done

end

context LLVM_preds begin

declare mods_def [simp]

lemma models_EXt: "\<lbrakk>models CFGs \<sigma> q (EXt t ty P); type_subst \<sigma> ty =
    Some et; by_thread P t;
 l \<in> tCFG.RPaths CFGs q; \<sigma> t = OThread t'; CFGs t' = Some G; tCFG CFGs
     LLVM_instr_edges seq\<rbrakk> \<Longrightarrow>
 \<exists>n n'. \<sigma> t = OThread t' \<and> q t' = Some n \<and> (n, n', et) \<in> Edges
      G \<and> models CFGs \<sigma> (q(t' \<mapsto> n')) P"
apply (clarsimp simp add: SCEX_t_def Let_def)
apply (cut_tac A="insert t (type_fv ty) \<union> cond_fv pred_fv P" in fresh_new, simp+,
    clarsimp)
apply (cut_tac \<sigma>=\<sigma> and \<sigma>'="\<sigma>(SOME y. y \<noteq> t \<and> y \<
    notin> type_fv ty \<and> y \<notin> cond_fv pred_fv P := obj)"
 and ty=ty in edge_type_same_subst, simp+)
apply (erule_tac x=t' in allE, simp)
apply (case_tac obj, simp_all, clarsimp)
apply (erule_tac x=et in allE, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (cut_tac A="{t, SOME y. y \<noteq> t \<and> y \<notin> type_fv ty \<and> y \<notin>
    cond_fv pred_fv P} \<union>
 cond_fv pred_fv P" in fresh_new, simp+)
apply (clarsimp split: option.splits)
apply (rule_tac x=yc in exI, simp)
apply (frule_tac i=i in tCFG.path_rpath, simp+, clarsimp)
apply (erule_tac q="la i" in by_threadD, simp_all)
apply force
apply (erule tCFG.rpath_incremental, simp+)
apply (force, simp+)
apply (erule side_cond_gen, auto)
done

end
```

```
context LLVM_a_only begin

lemma not_touches_store1: "\<lbrakk>Label G n = Store a b c e; expr_pattern_subst \<sigma>
    e' = Some d;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_touches t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' e d"
apply (clarsimp simp add: not_touches_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr e" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OType a, OExpr b, OType c, undefined, undefined, undefined]" in exI,
    simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr e)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_touches_load1: "\<lbrakk>Label G n = Load a b e; expr_pattern_subst \<sigma> e' =
     Some d;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_touches t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' e d"
```

```
apply (clarsimp simp add: not_touches_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr e" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OExpr %a, OType b, undefined, undefined, undefined, undefined]" in exI,
     simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr e)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_touches_cmpxchg1: "\<lbrakk>Label G n = Cmpxchg a b h d e f g; expr_pattern_subst
     \<sigma> e' = Some c;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_touches t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' h c"
apply (clarsimp simp add: not_touches_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr h" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
```

```
apply (rule_tac x="[OExpr %a, OType b, OType d, OExpr e, OType f, OExpr g]" in exI, simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr h)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_touches_store: "\<lbrakk>Label G n = Store a b c d; \<sigma> x = OExpr d; CFGs t'
     = Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_touches t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_touches_store1, auto simp add: AA_id)

lemma not_touches_load: "\<lbrakk>Label G n = Load a b c; \<sigma> x = OExpr c; CFGs t' =
    Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_touches t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_touches_load1, auto simp add: AA_id)

lemma not_touches_cmpxchg: "\<lbrakk>Label G n = Cmpxchg a b c d e f g; \<sigma> x = OExpr
    c; CFGs t' = Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_touches t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_touches_cmpxchg1, auto simp add: AA_id)
```

```
lemma not_loads1: "\<lbrakk>Label G n = Load a b e; expr_pattern_subst \<sigma> e' = Some d
    ;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_loads t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' e d"
apply (clarsimp simp add: not_loads_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr e" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OExpr %a, OType b]" in exI, simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=2 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=2 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr e)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_loads: "\<lbrakk>Label G n = Load a b c; \<sigma> x = OExpr c; CFGs t' = Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_loads t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_loads1, auto simp add: AA_id)

lemma not_touches_gen: "\<lbrakk>models CFGs \<sigma> q (not_touches t x); \<sigma> t =
    OThread t'; q t' = q' t';
```

```
 \<sigma>' t = OThread t'; expr_pattern_subst \<sigma> x = expr_pattern_subst \<sigma>' x\<
     rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q' (not_touches t x)"
apply (clarsimp simp only: not_touches_def Let_def models.simps)
apply clarsimp
apply (erule_tac x=obj in allE)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in
     new_nodes_are_new2, simp+)
apply (frule_tac i=0 in some_nth_distinct, simp+)
apply (frule_tac i=1 in some_nth_distinct, simp+)
apply (frule_tac i=2 in some_nth_distinct, simp+)
apply (frule_tac i=3 in some_nth_distinct, simp+)
apply (frule_tac i=4 in some_nth_distinct, simp+)
apply (frule_tac i=5 in some_nth_distinct, simp+)
apply (erule impE)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp)
apply (clarsimp simp only: update_list_of id_def)
apply (rule_tac x=objs in exI, clarify)
apply (case_tac "objs ! 0", simp_all)
apply clarsimp
apply (erule_tac x=y in allE, clarsimp)
apply (case_tac obj, simp_all)
apply (cut_tac \<sigma>'=\<sigma>' and \<sigma>="\<sigma>'(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
```

```
apply (cut_tac \<sigma>'=\<sigma> and \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply force
done

corollary not_touches_thread: "\<lbrakk>models CFGs \<sigma> q (not_touches t x); \<sigma>
    t = OThread t'; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (not_touches t x)"
by (simp add: not_touches_gen)

lemma not_loads_gen: "\<lbrakk>models CFGs \<sigma> q (not_loads t x); \<sigma> t = OThread
     t'; q t' = q' t';
 \<sigma>' t = OThread t'; expr_pattern_subst \<sigma> x = expr_pattern_subst \<sigma>' x\<
     rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q' (not_loads t x)"
apply (clarsimp simp only: not_loads_def Let_def models.simps)
apply clarsimp
apply (erule_tac x=obj in allE)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=2 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=2 and m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in
     new_nodes_are_new2, simp+)
apply (frule_tac i=0 in some_nth_distinct, simp+)
apply (frule_tac i=1 in some_nth_distinct, simp+)
apply (erule impE)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp)
apply (clarsimp simp only: update_list_of id_def)
```

```
apply (rule_tac x=objs in exI, clarsimp)
apply (clarsimp split: option.splits)
apply (case_tac obj, simp_all)
apply (cut_tac \<sigma>'=\<sigma>' and \<sigma>="\<sigma>'(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply (cut_tac \<sigma>'=\<sigma> and \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply force
done

corollary not_loads_thread: "\<lbrakk>models CFGs \<sigma> q (not_loads t x); \<sigma> t =
    OThread t'; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (not_loads t x)"
by (simp add: not_loads_gen)

lemma mods_gen: "\<lbrakk>models CFGs \<sigma> q (mods t x); \<sigma> t = OThread t'; q t'
    = q' t'; \<sigma>' t = OThread t';
 \<sigma> x = \<sigma>' x\<rbrakk> \<Longrightarrow> models CFGs \<sigma>' q' (mods t x)"
by (simp split: object.splits LLVM_expr.splits)

corollary mods_thread: "\<lbrakk>models CFGs \<sigma> q (mods t x); \<sigma> t = OThread t
    '; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (mods t x)"
by (simp add: mods_gen del: mods_def)

lemma not_mods_gen: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t = OThread t
    '; q t' = q' t';
 \<sigma>' t = OThread t'; \<sigma> x = \<sigma>' x\<rbrakk> \<Longrightarrow> models CFGs
     \<sigma>' q' (not_mods t x)"
apply (clarsimp simp add: not_mods_def simp del: mods_def)
apply (drule_tac q'=q and \<sigma>'=\<sigma> in mods_gen, simp+)
done
```

```
corollary not_mods_thread: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t =
    OThread t'; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (not_mods t x)"
by (simp add: not_mods_gen)

lemma mods_CFGs: "\<lbrakk>models CFGs' \<sigma> q (mods t x); \<sigma> t = OThread t';
    CFGs' t' = Some G';
 CFGs t' = Some G; Start G' = Start G; Exit G' = Exit G;
 Label G' (the (q t')) = Label G (the (q t'))\<rbrakk> \<Longrightarrow> models CFGs \<
     sigma> q (mods t x)"
by (clarsimp split: object.splits LLVM_expr.splits, force)

lemma not_mods_CFGs: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t = OThread
    t'; CFGs' t' = Some G'; CFGs t' = Some G;
 Start G' = Start G; Exit G' = Exit G; Label G' (the (q t')) = Label G (the (q t'))\<rbrakk
     > \<Longrightarrow>
 models CFGs' \<sigma> q (not_mods t x)"
apply (clarsimp simp add: not_mods_def simp del: mods_def)
apply (rule conjI, clarsimp simp del: mods_def, erule notE, erule mods_CFGs, simp+)
apply (case_tac "q t'", simp+)
done

end

context LLVM_mutex begin

lemma not_touches_store1: "\<lbrakk>Label G n = Store a b c e; expr_pattern_subst \<sigma>
    e' = Some d;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_touches t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' e d"
apply (clarsimp simp add: not_touches_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr e" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
```

```
apply (rule_tac x="[OType a, OExpr b, OType c, undefined, undefined, undefined]" in exI,
    simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr e)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_touches_load1: "\<lbrakk>Label G n = Load a b e; expr_pattern_subst \<sigma> e' =
     Some d;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_touches t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' e d"
apply (clarsimp simp add: not_touches_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr e" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OExpr %a, OType b, undefined, undefined, undefined, undefined]" in exI,
     simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m=t in new_nodes_are_new2, simp+)
```

```
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr e)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_touches_cmpxchg1: "\<lbrakk>Label G n = Cmpxchg a b h d e f g; expr_pattern_subst
     \<sigma> e' = Some c;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_touches t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' h c"
apply (clarsimp simp add: not_touches_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr h" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OExpr %a, OType b, OType d, OExpr e, OType f, OExpr g]" in exI, simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr h)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
```

```
done

lemma not_touches_store: "\<lbrakk>Label G n = Store a b c d; \<sigma> x = OExpr d; CFGs t'
     = Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_touches t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_touches_store1, auto simp add: AA_id)

lemma not_touches_load: "\<lbrakk>Label G n = Load a b c; \<sigma> x = OExpr c; CFGs t' =
    Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_touches t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_touches_load1, auto simp add: AA_id)

lemma not_touches_cmpxchg: "\<lbrakk>Label G n = Cmpxchg a b c d e f g; \<sigma> x = OExpr
    c; CFGs t' = Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_touches t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_touches_cmpxchg1, auto simp add: AA_id)

lemma not_loads1: "\<lbrakk>Label G n = Load a b e; expr_pattern_subst \<sigma> e' = Some d
    ;
 CFGs t' = Some G; q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs
     \<sigma> q (not_loads t e')\<rbrakk>
 \<Longrightarrow> cannot_alias CFGs n t' e d"
apply (clarsimp simp add: not_loads_def Let_def)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv e')" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv e'" in new_nodes_are_new
     , simp+)
apply (erule_tac x="OExpr e" in allE, simp, erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OExpr %a, OType b]" in exI, simp)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=2 in new_nodes_diff, simp+)
```

```
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=2 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e')
 (expr_pattern_fv expr_fv e'))" and n=2 and m="SOME n. n \<noteq> t \<and> n \<notin>
     expr_pattern_fv expr_fv e'"
 in new_nodes_are_new2, simp+)
apply (cut_tac \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv e' := OExpr e)" and \<sigma>'=\<sigma> and
 e=e' in expr_pattern_same_subst, simp+)
apply force
done

lemma not_loads: "\<lbrakk>Label G n = Load a b c; \<sigma> x = OExpr c; CFGs t' = Some G;
 q t' = Some n; \<sigma> t = OThread t'; n \<noteq> Exit G; models CFGs \<sigma> q (
     not_loads t (Var x))\<rbrakk>
 \<Longrightarrow> False"
by (drule not_loads1, auto simp add: AA_id)

lemma not_touches_gen: "\<lbrakk>models CFGs \<sigma> q (not_touches t x); \<sigma> t =
    OThread t'; q t' = q' t';
 \<sigma>' t = OThread t'; expr_pattern_subst \<sigma> x = expr_pattern_subst \<sigma>' x\<
     rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q' (not_touches t x)"
apply (clarsimp simp only: not_touches_def Let_def models.simps)
apply clarsimp
apply (erule_tac x=obj in allE)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=6 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=6 and m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in
     new_nodes_are_new2, simp+)
```

```
apply (frule_tac i=0 in some_nth_distinct, simp+)
apply (frule_tac i=1 in some_nth_distinct, simp+)
apply (frule_tac i=2 in some_nth_distinct, simp+)
apply (frule_tac i=3 in some_nth_distinct, simp+)
apply (frule_tac i=4 in some_nth_distinct, simp+)
apply (frule_tac i=5 in some_nth_distinct, simp+)
apply (erule impE)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp)
apply (clarsimp simp only: update_list_of id_def)
apply (rule_tac x=objs in exI, clarify)
apply (case_tac "objs ! 0", simp_all)
apply clarsimp
apply (erule_tac x=y in allE, clarsimp)
apply (case_tac obj, simp_all)
apply (cut_tac \<sigma>'=\<sigma>' and \<sigma>="\<sigma>'(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply (cut_tac \<sigma>'=\<sigma> and \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply force
done

corollary not_touches_thread: "\<lbrakk>models CFGs \<sigma> q (not_touches t x); \<sigma>
    t = OThread t'; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (not_touches t x)"
by (simp add: not_touches_gen)

lemma not_loads_gen: "\<lbrakk>models CFGs \<sigma> q (not_loads t x); \<sigma> t = OThread
     t'; q t' = q' t';
 \<sigma>' t = OThread t'; expr_pattern_subst \<sigma> x = expr_pattern_subst \<sigma>' x\<
     rbrakk> \<Longrightarrow>
 models CFGs \<sigma>' q' (not_loads t x)"
```

```
apply (clarsimp simp only: not_loads_def Let_def models.simps)
apply clarsimp
apply (erule_tac x=obj in allE)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and m=t in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=2 and m=t in new_nodes_are_new2, simp+)
apply (cut_tac S="insert t (insert (SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv
    expr_fv x)  (expr_pattern_fv expr_fv x))"
 and n=2 and m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in
     new_nodes_are_new2, simp+)
apply (frule_tac i=0 in some_nth_distinct, simp+)
apply (frule_tac i=1 in some_nth_distinct, simp+)
apply (erule impE)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def)
apply (rule subst, rule sym, rule exs_simp)
apply (clarsimp simp only: update_list_of id_def)
apply (rule_tac x=objs in exI, clarsimp)
apply (clarsimp split: option.splits)
apply (case_tac obj, simp_all)
apply (cut_tac \<sigma>'=\<sigma>' and \<sigma>="\<sigma>'(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply (cut_tac \<sigma>'=\<sigma> and \<sigma>="\<sigma>(SOME n. n \<noteq> t \<and> n \<
    notin> expr_pattern_fv expr_fv x := OExpr y)" and e=x
 in expr_pattern_same_subst, clarsimp)
apply (cut_tac S="insert t (expr_pattern_fv expr_fv x)" and n=1 and
 m="SOME n. n \<noteq> t \<and> n \<notin> expr_pattern_fv expr_fv x" in new_nodes_are_new,
      simp+)
apply force
done
```

```
corollary not_loads_thread: "\<lbrakk>models CFGs \<sigma> q (not_loads t x); \<sigma> t =
    OThread t'; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (not_loads t x)"
by (simp add: not_loads_gen)

lemma mods_gen: "\<lbrakk>models CFGs \<sigma> q (mods t x); \<sigma> t = OThread t'; q t'
    = q' t'; \<sigma>' t = OThread t';
 \<sigma> x = \<sigma>' x\<rbrakk> \<Longrightarrow> models CFGs \<sigma>' q' (mods t x)"
by (simp split: object.splits LLVM_expr.splits)

corollary mods_thread: "\<lbrakk>models CFGs \<sigma> q (mods t x); \<sigma> t = OThread t
    '; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (mods t x)"
by (simp add: mods_gen del: mods_def)

lemma not_mods_gen: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t = OThread t
    '; q t' = q' t';
 \<sigma>' t = OThread t'; \<sigma> x = \<sigma>' x\<rbrakk> \<Longrightarrow> models CFGs
     \<sigma>' q' (not_mods t x)"
apply (clarsimp simp add: not_mods_def simp del: mods_def)
apply (drule_tac q'=q and \<sigma>'=\<sigma> in mods_gen, simp+)
done

corollary not_mods_thread: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t =
    OThread t'; q t' = q' t'\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> q' (not_mods t x)"
by (simp add: not_mods_gen)

lemma mods_CFGs: "\<lbrakk>models CFGs' \<sigma> q (mods t x); \<sigma> t = OThread t';
    CFGs' t' = Some G';
 CFGs t' = Some G; Start G' = Start G; Exit G' = Exit G;
 Label G' (the (q t')) = Label G (the (q t'))\<rbrakk> \<Longrightarrow> models CFGs \<
     sigma> q (mods t x)"
by (clarsimp split: object.splits LLVM_expr.splits, force)

lemma not_mods_CFGs: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t = OThread
    t'; CFGs' t' = Some G'; CFGs t' = Some G;
 Start G' = Start G; Exit G' = Exit G; Label G' (the (q t')) = Label G (the (q t'))\<rbrakk
     > \<Longrightarrow>
 models CFGs' \<sigma> q (not_mods t x)"
```

```
apply (clarsimp simp add: not_mods_def simp del: mods_def)
apply (rule conjI, clarsimp simp del: mods_def, erule notE, erule mods_CFGs, simp+)
apply (case_tac "q t'", simp+)
done

end

(* locale setup *)
locale TRANS_LLVM = LLVM_threads where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
  LLVM_mutex where cannot_alias="cannot_alias::('thread, 'node, string) LLVM_tCFG \<
      Rightarrow>
  'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string,
      int) LLVM_expr \<Rightarrow> bool" for CFGs cannot_alias
begin

lemma not_mods_same [simp]: "\<lbrakk>models CFGs \<sigma> q (not_mods t x); \<sigma> t =
    OThread t'; \<sigma> x = OExpr (Local x');
 get_points (fst C) = q; conc_step CFGs C C'; (fst C) t' = Some (p0, p, env, rest);
 p \<noteq> Exit (the (CFGs t')); (fst C') t' = Some (p, p', env', rest')\<rbrakk> \<
     Longrightarrow>
 env' x' = env x'"
apply (erule conc_step.cases, auto simp add: not_mods_def mods_def map_comp_def split:
    if_splits)
apply (erule step_cases, auto)
done

lemma protected_by_thread [intro!]: "by_thread (SCPred (Other (Ext (Protected x l)))) t"
by (clarsimp intro!: by_threadI2)

lemma in_critical_by_thread [intro!]: "by_thread (SCPred (Other (Ext (InCritical t x l))))
    t"
apply (clarsimp intro!: by_threadI2)
apply (case_tac "expr_pattern_subst \<sigma> x", simp+)
apply (case_tac "\<sigma> l", simp+)
done

end

locale TRANS_LLVM_SC = LLVM_SC where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
```

```
  LLVM_mutex where free_set=free_set and can_read=can_read and update_mem=update_mem and
      start_mem=start_mem and
  cannot_alias="cannot_alias::('thread, 'node, string) LLVM_tCFG \<Rightarrow> 'node \<
      Rightarrow> 'thread \<Rightarrow>
    (string, int) LLVM_expr \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> bool" for
        CFGs cannot_alias

sublocale TRANS_LLVM_SC \<subseteq> TRANS_LLVM where free_set=free_set and can_read=
    can_read and start_mem=start_mem
  and update_mem=update_mem
by unfold_locales

locale TRANS_LLVM_TSO = LLVM_TSO where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
  LLVM_mutex where free_set=free_set and can_read=can_read and update_mem=update_mem and
      start_mem=start_mem and
  cannot_alias="cannot_alias::('thread, 'node, string) LLVM_tCFG \<Rightarrow>
    'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string
        , int) LLVM_expr \<Rightarrow> bool" for CFGs cannot_alias

sublocale TRANS_LLVM_TSO \<subseteq> TRANS_LLVM where free_set=free_set and can_read=
    can_read and start_mem=start_mem
  and update_mem=update_mem
by unfold_locales

locale TRANS_LLVM_PSO = LLVM_PSO where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
  LLVM_mutex where free_set=free_set and can_read=can_read and update_mem=update_mem and
      start_mem=start_mem and
  cannot_alias="cannot_alias::('thread, 'node, string) LLVM_tCFG \<Rightarrow>
    'node \<Rightarrow> 'thread \<Rightarrow> (string, int) LLVM_expr \<Rightarrow> (string
        , int) LLVM_expr \<Rightarrow> bool" for CFGs cannot_alias

sublocale TRANS_LLVM_PSO \<subseteq> TRANS_LLVM where free_set=free_set and can_read=
    can_read and start_mem=start_mem
  and update_mem=update_mem
by unfold_locales

(* mutex proofs *)
locale TRANS_LLVM_SC_mutex = TRANS_LLVM_SC
sublocale TRANS_LLVM_SC_mutex \<subseteq> a_only!: LLVM_a_only_CFG where free_set=free_set
    and can_read=can_read
```

```
  and start_mem=start_mem and update_mem=update_mem and CFGs=CFGs
by (unfold_locales, simp)

context TRANS_LLVM_SC_mutex begin

definition "lock_nz x C mem l \<equiv> mem l \<noteq> Some (CInt 0) \<longrightarrow>
 (\<exists>t. \<forall>t'. ((a_only.models CFGs ((\<lambda>x. OOp add)(''t'' := OThread t',
      ''x'' := OExpr x)) (get_points C)
 (a_only.locks ''t'' ''x'') \<and> (case the (C t') of (p0, p, env, rest) \<Rightarrow> (\<
     exists>e. Label (the (CFGs t')) p = Br_i1 e \<and>
 eval_expr env gt e = CInt 0))) \<or> a_only.models CFGs ((\<lambda>x. OOp add)
 (''t'' := OThread t', ''x'' := OExpr x)) (get_points C) (a_only.in_critical_lock ''t'' ''x
     '')) \<longrightarrow> t' = t)"

definition "lock_z x C mem l \<equiv> mem l = Some (CInt 0) \<longrightarrow>
 (\<forall>t. \<not>((a_only.models CFGs ((\<lambda>x. OOp add)(''t'' := OThread t, ''x''
     := OExpr x)) (get_points C)
 (a_only.locks ''t'' ''x'') \<and> (case the (C t) of (p0, p, env, rest) \<Rightarrow> (\<
     exists>e. Label (the (CFGs t)) p = Br_i1 e \<and>
 eval_expr env gt e = CInt 0))) \<or> a_only.models CFGs ((\<lambda>x. OOp add)
 (''t'' := OThread t, ''x'' := OExpr x)) (get_points C) (a_only.in_critical_lock ''t'' ''x
     '')))"

lemma mutex0: "\<lbrakk>run_prog CFGs (C, mem); a_only.models CFGs \<sigma> (start_points
    CFGs)
 (SCAll ''t'' (a_only.good_lock ''t'' ''x''))\<rbrakk> \<Longrightarrow>
 \<exists>x' l. \<sigma> ''x'' = OExpr x' \<and> eval_expr env gt x' = CPointer l \<and>
     lock_nz x' C mem l \<and>
 lock_z x' C mem l"
apply simp
apply (rule allE, simp, clarsimp simp add: a_only.good_lock_def)
apply (thin_tac "\<forall>l\<in>Paths ?S. ?P l")
apply (case_tac "\<sigma> ''x''", simp_all, case_tac LLVM_expr, simp_all, clarsimp+)
apply (frule run_global, simp, clarsimp)
apply (drule_tac P="\<lambda>(C, mem). \<exists>y. l \<notin> all_alloc_mem C \<and> mem l
    = Some y \<and>
 lock_nz (Global list) C mem l \<and> lock_z (Global list) C mem l" in run_prog_induct,
     simp_all)
(* base case *)
```

```
apply (rule conjI, clarsimp simp add: all_alloc_mem_def ran_def alloc_mem_def split: option
    .splits)
apply (frule global_alloc, simp+)
apply (clarsimp, simp+)
apply (rule conjI, force)
apply (clarsimp simp add: lock_nz_def lock_z_def)
(* inductive case *)
apply (case_tac Ca, case_tac C', clarsimp)
apply (frule run_prog_conc_step, simp)
apply (frule_tac C="(aaa, baa)" in run_global, simp+, clarsimp)
apply (frule run_prog_safe, unfold_locales)
apply (cut_tac A="{''t'',''x''}" in fresh_new, simp+)
apply (case_tac "ya \<noteq> CInt 0", clarsimp simp add: lock_nz_def, rule conjI, clarsimp)
(* someone has already claimed the lock *)
apply (rule_tac x=t in exI, clarsimp)
apply (erule conc_step.cases, clarsimp)
apply (case_tac "t' \<noteq> ta", erule_tac x=t' in allE, clarsimp)
(* the step made isn't for the thread in question *)
apply (drule run_prog_rpath, unfold_locales)+
apply (rule conjI, clarsimp)
apply (cut_tac q="get_points states(ta \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.locks_by_thread], simp_all, simp+)
apply unfold_locales
apply clarsimp+
apply (cut_tac q="get_points states(ta \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all, simp+)
apply unfold_locales
apply clarsimp+
(* the step made is for the thread in question, but it can only proceed if it's the one
   that claimed the lock *)
apply (clarsimp simp add: safe_points_def, erule_tac x=ta in allE, simp)
apply (case_tac "aa = af", simp add: map_upd_triv)
apply (clarsimp, frule a_only.locks_br, clarsimp)
apply (rule step_cases, simp_all, simp_all)
apply (frule step_next, simp+)
apply (frule step_along_edge, simp+)
apply (rule conjI, clarsimp)
(* the thread in question is trying to get the lock *)
apply (thin_tac "\<forall>t'. ?P t'")+
apply (clarsimp simp add: a_only.locks_def Let_def)
```

```
apply (cut_tac a_only.exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (cut_tac S="{''t'', ''x''}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''t''" in new_nodes_are_new2, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''x''" in new_nodes_are_new2, simp+)
apply (cut_tac A="{''t'', new {''t'', ''x''} 5 ! 4}" in fresh_new, simp+, clarsimp)
apply (erule_tac x=yd in allE, clarsimp)
apply (case_tac obj, simp_all, case_tac "objs ! 0", simp_all, clarsimp)
apply (frule_tac C="(states(yd \<mapsto> (aa, yc, ag, ah, ba)), mem')" in run_prog_rpath,
 clarsimp+)
apply (drule_tac t'=yd in a_only.models_EXt, simp_all)
apply (force, simp add: fun_upd_def, clarsimp+)
apply (case_tac ab, simp_all)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "objs ! 2", simp_all)
apply (case_tac "objs ! 3", simp_all, case_tac "objs ! 4", simp_all, clarsimp)
(* so it must have just executed the cmpxchg, and failed *)
apply (drule run_prog_rpath, unfold_locales, clarsimp)
apply (frule_tac l=lb and q="get_points states" in rpath_incremental, simp_all, force)
apply (erule_tac x="[get_points states(yd \<mapsto> yc)] \<frown> lb" in ballE, clarsimp)
apply (case_tac i, simp+)
apply (case_tac "nat > 0", erule_tac x=1 in allE, simp+, clarsimp)
apply (rule step_cases, simp_all, simp_all)
apply clarsimp+
apply (simp add: fun_upd_def)
(* the thread in question is in the critical section, so it must be the one that claimed
    the lock *)
apply clarsimp
apply (erule_tac x=ta in allE, clarsimp)
apply (frule_tac C=states in a_only.locks_in, simp_all)
apply (frule a_only.locks_br, clarsimp)
apply (subgoal_tac "(\<exists>e. (af, aa, e) \<in> Edges G) \<and>
 (\<exists>a b c d e f g. Label G af = Cmpxchg a b c d e f g)", clarsimp)
apply (drule_tac n'=aa and C="states(ta \<mapsto> (aa, af, ag, ah, ba))"
 in a_only.edge_out_unlocks, simp_all)
apply (simp add: map_upd_triv)
apply (subgoal_tac "a_only.models CFGs (\<sigma>(''t'' := OThread ta)) (start_points CFGs)
 (a_only.good_lock ''t'' ''x'')", erule a_only.side_cond_gen)
apply unfold_locales
apply simp
apply (clarsimp simp add: a_only.good_lock_def)
```

```
apply (rule conjI, metis)
apply clarsimp
apply (erule_tac x="OThread ta" in allE, erule_tac x=la in ballE, simp_all)
apply (erule_tac x=i in allE, clarsimp)
apply (erule_tac x=obj in allE, clarsimp)
apply (erule_tac x=lb in ballE, simp_all)
apply (erule_tac P="\<lambda>i. lb i ta = Some ae \<or> ?P i" and x=ia in allE, clarsimp)
apply force
apply (drule a_only.unlocks_store, simp)
apply (thin_tac "\<forall>x. ?P x")
apply (clarsimp simp add: a_only.locks_def Let_def)
apply (cut_tac a_only.exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (cut_tac S="{''t'', ''x''}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''t''" in new_nodes_are_new2, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''x''" in new_nodes_are_new2, simp+)
apply (cut_tac A="{''t'', new {''t'', ''x''} 5 ! 4}" in fresh_new, simp+, clarsimp)
apply (erule_tac x=yd in allE, clarsimp)
apply (case_tac obj, simp_all, case_tac "objs ! 0", simp_all, clarsimp)
apply (frule_tac C="(states(yd \<mapsto> (yc, af, ag, ah, ba)), mem')" in run_prog_rpath,
 clarsimp)
apply (drule run_prog_rpath, unfold_locales, clarsimp)
apply (drule_tac t'=yd in a_only.models_EXt, simp_all)
apply (force, simp add: fun_upd_def, clarsimp+)
apply (case_tac ea, simp_all)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "objs ! 2", simp_all)
apply (case_tac "objs ! 3", simp_all, case_tac "objs ! 4", simp_all, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (frule_tac l=lb in a_only.prev_edge, simp_all, simp+)
apply (rule step_cases, simp_all, simp_all, clarsimp)
apply (drule CFGs, clarsimp simp add: is_flowgraph_def,
 frule_tac u=p in flowgraph.instr_edges_ok, simp_all)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.finite_edges, clarsimp)
apply (simp add: out_edges_def)
apply (subgoal_tac "(aa, true) \<in> {(u, t). (p, u, t) \<in> Edges G}", clarsimp)
apply (rule conjI, metis)
apply (drule_tac t="Label G m1" in sym, force, force)
(* if the lock was set and is now unset, the current thread must have just released it,
   so no one else has claimed it *)
apply (clarsimp simp add: lock_z_def)
```

```
apply (erule conc_step.cases, clarsimp)
apply (frule run_prog_path, unfold_locales, clarsimp)
apply (erule_tac x="OThread tb" in allE, erule_tac x=la in ballE, simp_all, erule_tac x=i
    in allE,
 clarsimp)
apply (erule disjE, drule a_only.locks2_cmpxchg, clarsimp)
apply (rule step_cases, simp_all, simp_all, clarsimp)
apply (erule disjE, clarsimp)
apply (rule_tac x=tb in allE, simp, clarify)
apply (erule impE, erule a_only.side_cond_gen, simp, unfold_locales)
apply (clarify, thin_tac "?P \<longrightarrow> t = t")
apply (frule step_next, simp+)
apply (frule_tac C=states in a_only.unlocks_out, simp_all)
apply (rule conjI, clarsimp)
apply (rule conjI, clarsimp, drule a_only.unlocks_store, drule a_only.locks_locks2,
 clarsimp simp add: Let_def)
apply (drule run_prog_rpath, unfold_locales, clarsimp)
apply (frule_tac l=lb and ps="get_points states" in step_increment_rpath, simp_all)
apply (simp add: safe_points_def, erule_tac x=t in allE, simp)
apply (erule_tac x="[get_points states(t \<mapsto> yb)] \<frown> lb" in ballE, simp_all,
    clarsimp)
apply (case_tac obj, simp_all, clarsimp)
apply (case_tac ia, simp+)
apply (case_tac "nat > 0", (erule_tac x=1 in allE)+, clarsimp+)
apply (drule a_only.locks2_cmpxchg, simp)
apply (clarsimp, drule_tac q="get_points states(t \<mapsto> af)" and \<sigma>'="\<sigma>(''
    t'' := OThread t)"
 in a_only.side_cond_gen, simp_all)
apply clarsimp+
apply (erule_tac x=ta in allE, clarsimp)
apply (drule run_prog_rpath, unfold_locales, clarsimp)+
apply (rule conjI, clarsimp)
apply (cut_tac q="get_points states(t \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.locks_by_thread], simp_all, simp+)
apply clarsimp+
apply (cut_tac q="get_points states(t \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all, simp+)
apply (rule step_cases, simp_all, simp_all, clarsimp+)
apply (clarsimp simp add: restrict_map_def, case_tac "l \<notin> allocad", simp+)+
apply (clarsimp split: if_splits)+
```

```
apply (drule_tac \<sigma>="\<sigma>(''t'' := OThread tc)" and e'="Var ''x''" and t="''t''"
    and CFGs=CFGs in
 a_only.not_touches_store1, simp_all)
apply (drule_tac C="(states, mema)" in AA_correct, simp_all, force, unfold_locales)
apply simp+
apply (clarsimp split: if_splits)+
apply (drule_tac \<sigma>="\<sigma>(''t'' := OThread tc)" and e'="Var ''x''" and t="''t''"
    in
 a_only.not_touches_cmpxchg1, simp_all)
apply (drule_tac C="(states, mema)" in AA_correct, simp_all, force, unfold_locales)
apply simp+
apply (clarsimp split: if_splits)+
(* no one currently has claimed the lock *)
apply (clarsimp simp add: lock_nz_def lock_z_def)
apply (rule conjI, clarsimp)
(* the lock was just claimed by the thread in question *)
apply (erule conc_step.cases, clarsimp)
apply (rule_tac x=t in exI, clarsimp)
apply (erule_tac x=t' in allE, clarsimp)
apply (drule run_prog_rpath, unfold_locales, clarsimp)+
apply (rule conjI, clarsimp)
apply (frule step_next, simp+)
apply (cut_tac q="get_points states(t \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.locks_by_thread], simp_all, simp+)
apply unfold_locales
apply clarsimp+
apply (cut_tac q="get_points states(t \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all, simp+)
apply unfold_locales
apply clarsimp+
(* the thread doesn't set the lock to 1, so neither it nor anyone else claims it *)
apply (erule conc_step.cases, clarsimp)
apply (case_tac "aa = af", clarsimp simp add: map_upd_triv)
apply (erule_tac x=ta in allE, clarsimp)
apply (drule a_only.locks_br, clarsimp)
apply (rule step_cases, simp_all, simp_all)
apply (rule conjI, clarsimp)
(* the thread acts, but doesn't claim the lock *)
apply (erule_tac x=ta in allE, clarsimp)
(* the thread can't be trying to claim the lock, because it would have set the lock bit *)
```

```
apply (rule conjI, clarsimp)
apply (thin_tac "\<forall>x. ?P x")
apply (clarsimp simp add: locks_def Let_def)
apply (cut_tac a_only.exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (cut_tac S="{''t'', ''x''}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''t''" in new_nodes_are_new2, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''x''" in new_nodes_are_new2, simp+)
apply (cut_tac A="{''t'', new {''t'', ''x''} 5 ! 4}" in fresh_new, simp+, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (case_tac obj, simp_all, case_tac "objs ! 0", simp_all, clarsimp)
apply (frule_tac C="(states(yb \<mapsto> (ae, ya, ag, ah, ba)), mem')" in run_prog_rpath,
    unfold_locales,
 clarsimp)
apply (drule_tac t'=yb in a_only.models_EXt, simp_all)
apply (force, simp add: fun_upd_def, clarsimp+)
apply (case_tac ab, simp_all)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "objs ! 2", simp_all)
apply (case_tac "objs ! 3", simp_all, case_tac "objs ! 4", simp_all, clarsimp)
(* it must have just executed the cmpxchg, but this is a contradiction *)
apply (drule run_prog_rpath, unfold_locales, clarsimp)
apply (frule step_next, simp+)
apply (frule step_along_edge, simp+)
apply (clarsimp simp add: safe_points_def, erule_tac x=yb in allE, simp, clarsimp)
apply (frule_tac l=lb and q="get_points states" in rpath_incremental, simp_all)
apply force
apply (erule_tac x="[get_points states(yb \<mapsto> ya)] \<frown> lb" in ballE, clarsimp)
apply (case_tac i, simp+)
apply (case_tac "nat > 0", erule_tac x=1 in allE, simp+, clarsimp)
apply (rule step_cases, simp_all, simp_all, clarsimp+)
apply (simp add: fun_upd_def)
(* the thread enters the critical section, but can't have because it didn't claim the lock
    *)
apply clarsimp
apply (frule step_next, simp+)
apply (frule_tac C=states in a_only.locks_in, simp_all)
apply (clarsimp simp add: locks_def Let_def)
apply (cut_tac a_only.exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (cut_tac S="{''t'', ''x''}" and n=5 in new_nodes_diff, simp+)
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''t''" in new_nodes_are_new2, simp+)
```

```
apply (cut_tac S="{''t'', ''x''}" and n=5 and m="''x''" in new_nodes_are_new2, simp+)
apply (cut_tac A="{''t'', new {''t'', ''x''} 5 ! 4}" in fresh_new, simp+, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (case_tac obj, simp_all, case_tac "objs ! 0", simp_all, clarsimp)
apply (frule_tac C="(states(yb \<mapsto> (ya, af, ag, ah, ba)), mem')" in run_prog_rpath,
    unfold_locales,
 clarsimp)
apply (frule run_prog_rpath, unfold_locales, clarsimp)
apply (drule_tac t'=yb in a_only.models_EXt, simp_all)
apply (force, simp add: fun_upd_def, clarsimp+)
apply (case_tac ab, simp_all)
apply (case_tac "objs ! 1", simp_all)
apply (case_tac "objs ! 2", simp_all)
apply (case_tac "objs ! 3", simp_all, case_tac "objs ! 4", simp_all, clarsimp)
apply (rule step_cases, simp_all, simp_all, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (frule_tac l=lb in a_only.prev_edge, simp_all, force)
apply (subgoal_tac "next_node (Edges G) true p = aa", clarsimp)
apply (drule_tac n'=p and C="\<lambda>x. if x = t then Some (p, next_node (Edges G) true p,
     env, stack, allocad)
 else states x" in a_only.edge_out_unlocks, simp_all)
apply (simp add: map_upd_triv)
apply (simp add: fun_upd_def)
apply (subgoal_tac "a_only.models CFGs (\<sigma>(''t'' := OThread t)) (start_points CFGs)
 (good_lock ''t'' ''x'')", erule a_only.side_cond_gen)
apply clarsimp+
apply (erule_tac x="OThread t" in allE, clarsimp simp add: good_lock_def)
apply (rule conjI, metis)
apply (clarsimp simp add: fun_upd_def locks_def Let_def)
apply (clarsimp intro!: ext simp add: map_comp_def)
apply force
apply (drule a_only.unlocks_store, simp)
apply clarsimp
apply (frule CFGs, clarsimp simp add: is_flowgraph_def, frule_tac u=p in flowgraph.
    instr_edges_ok,
 simp_all)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.edges_ok, force, simp)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.finite_edges, clarsimp)
apply (simp add: out_edges_def, subgoal_tac "(aa, true) \<in> {(u, t). (p, u, t) \<in>
    Edges G}",
```

```
 clarsimp, force)
(* finally, nodes that don't step can't have entered the critical section *)
apply (erule_tac x=t in allE, clarsimp)
apply (drule run_prog_rpath, unfold_locales, clarsimp)+
apply (rule conjI, clarsimp)
apply (cut_tac q="get_points states(ta \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.locks_by_thread], simp_all, simp+)
apply clarsimp
apply (cut_tac q="get_points states(ta \<mapsto> af)" and q'="get_points states"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all, simp+)
done

(* side note: If I was doing SSA, some of these globals wouldn't be necessary. *)
lemma mutex1: "\<lbrakk>run_prog CFGs (C, mem); C t = Some (p0, n, rest); C t' = Some (p0',
    n', rest');
 a_only.models CFGs ((\<lambda>x. OOp add)(''l'' := OExpr l, ''x'' := OExpr x))
 (start_points CFGs) (protected_l ''x'' (Var ''l'')); a_only.models CFGs ((\<lambda>x. OOp
    add)(''t'' :=
 OThread t, ''l'' := OExpr l, ''x'' := OExpr x)) [t \<mapsto> n] (in_critical_lock ''t'' ''
    x'');
 a_only.models CFGs ((\<lambda>x. OOp add)(''t'' := OThread t', ''l'' := OExpr l, ''x'' :=
    OExpr x))
 [t' \<mapsto> n'] (in_critical_lock ''t'' ''x'')\<rbrakk> \<Longrightarrow> t = t'"
apply (frule_tac \<sigma>="(\<lambda>x. OOp add)(''l'' := OExpr l, ''x'' := OExpr x)" in
   mutex0)
apply (clarsimp simp add: protected_l_def Let_def)
apply (erule_tac x=obj in allE, clarsimp)
apply (cut_tac A="{''x'', ''l''}" in fresh_new, simp+)
apply (drule_tac t'="''t''" in a_only.good_lock_bound, simp+)
apply unfold_locales
apply (erule a_only.side_cond_gen)
apply unfold_locales
apply clarsimp+
apply (clarsimp simp add: lock_nz_def lock_z_def)
apply (frule run_prog_rpath, clarsimp)
apply (case_tac rest, case_tac rest', clarsimp)
apply (frule_tac t=t in rpath_one_thread, simp)
apply (frule_tac t=t' in rpath_one_thread, simp)
apply (case_tac "mem la = Some (CInt 0)", clarsimp)
apply (erule_tac x=t' in allE, clarsimp)
```

```
apply (cut_tac q="[t' \<mapsto> n']" and q'="get_points C"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all)
apply (simp add: map_comp_def)
apply unfold_locales
apply (drule_tac q="get_points C" and \<sigma>'="(\<lambda>x. OOp add)(''t'' := OThread t',
     ''x'' := OExpr x)"
 in a_only.side_cond_gen)
apply unfold_locales
apply clarsimp+
apply (cut_tac q="[t \<mapsto> n]" and q'="get_points C"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all, simp add: map_comp_def)
apply (cut_tac q="[t' \<mapsto> n']" and q'="get_points C"
 in a_only.by_threadD [OF a_only.in_critical_by_thread], simp_all, simp add: map_comp_def)
apply (drule_tac q="get_points C" and \<sigma>'="(\<lambda>x. OOp add)(''t'' := OThread t,
    ''x'' := OExpr x)"
 in a_only.side_cond_gen)
apply unfold_locales
apply simp
apply (drule_tac q="get_points C" and \<sigma>'="(\<lambda>x. OOp add)(''t'' := OThread t',
     ''x'' := OExpr x)"
 in a_only.side_cond_gen)
apply unfold_locales
apply simp
apply (rule_tac x=t in allE, assumption, clarify)
apply (erule_tac x=t' in allE, force)
done

end

lemma start_points_struct [simp]: "CFGs t = Some G \<Longrightarrow>
 start_points (CFGs(t \<mapsto> G\<lparr>Label := L'\<rparr>)) = start_points CFGs"
by (rule ext, simp add: start_points_def)

sublocale TRANS_LLVM_SC_mutex \<subseteq> mut1!: TRANS_LLVM_SC_mutex where
 protected="\<lambda>CFGs' x l. a_only.models CFGs' ((\<lambda>x. OOp add)(''l'' := OExpr l
     , ''x'' := OExpr x))
 (start_points CFGs') (protected_l ''x'' (Var ''l''))" and in_critical="\<lambda>CFGs' n t
     x l. a_only.models CFGs'
 ((\<lambda>x. OOp add)(''t'' := OThread t, ''l'' := OExpr l, ''x'' := OExpr x)) [t \<
     mapsto> n] (in_critical_l ''t'' ''x'' %''l'')"
```

```
apply unfold_locales
(* protected_l and in_critical_l provide mutual exclusion. *)
apply (case_tac C, clarsimp)
apply (rule_tac C=ae in TRANS_LLVM_SC_mutex.mutex1, simp_all)
apply (clarsimp simp add: LLVM_threads_def LLVM_tCFG_def, unfold_locales)
apply (erule tCFG.CFGs, simp)
apply (erule_tac t'=t'a in tCFG.disjoint, simp+)
apply (erule tCFG.finite_threads)
apply force
apply force
(* protected_l guarantees that l is protected by x. *)
apply (clarsimp simp add: protected_l_def Let_def)
apply (cut_tac A="{''x'',''l''}" in fresh_new, simp+)
apply (erule_tac x="OThread t" in allE, clarsimp)
apply (frule run_prog_path, simp add: LLVM_threads_def LLVM_tCFG_def, clarsimp)
apply (erule_tac x=l in ballE, simp_all)
apply (erule_tac x=i in allE, simp)
apply (simp add: fun_upd_twist)
apply (frule run_prog_rpath, simp add: LLVM_threads_def LLVM_tCFG_def, clarsimp)
apply (cut_tac t=t in tCFG.rpath_one_thread, clarsimp simp add: LLVM_threads_def
    LLVM_tCFG_def, simp+)
apply (erule disjE, rule_tac q="get_points a" in a_only.by_threadD [OF a_only.
    in_critical_by_thread],
 simp_all, simp+)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (cut_tac t'="''t''" and CFGs=CFGs in LLVM_a_only_CFG.in_critical_bound)
apply (simp add: LLVM_a_only_CFG_def LLVM_threads_def)
apply unfold_locales
apply simp+
apply (erule a_only.side_cond_gen, simp+)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (clarsimp simp add: good_lock_def)
apply (erule_tac x=l in ballE, simp_all, erule_tac x=i in allE, clarsimp)
apply (case_tac y, simp_all, clarsimp)
apply (frule LLVM_threads.run_global, simp+, clarsimp simp add: all_alloc_mem_def ran_def
    alloc_mem_def)
apply (erule_tac x=p0 in allE, erule_tac x=p in allE, erule_tac x=env in allE, erule_tac x=
    aa in allE,
 erule_tac x=ba in allE, erule disjE, erule_tac x=t in allE, simp+)
apply (rule LLVM_threads.step_cases, simp+, simp_all, clarsimp+)
```

```
apply (frule_tac \<sigma>="((\<lambda>x. OOp add)(''l'' := OExpr (Global list), ''t'' :=
    OThread ta,
 ''x'' := OExpr x))" and q="get_points a" and e'="Var ''l''" and t="''t''" and CFGs=CFGs
 in a_only.not_touches_load1, simp_all)
apply (cut_tac t'="''t''" and CFGs=CFGs in LLVM_a_only_CFG.not_touches_bound)
apply (simp add: LLVM_a_only_CFG_def LLVM_threads_def)
apply unfold_locales
apply simp+
apply (erule a_only.side_cond_gen, simp+)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (drule AA_correct, simp_all, force, unfold_locales)
apply simp+
apply (frule_tac \<sigma>="((\<lambda>x. OOp add)(''l'' := OExpr (Global list), ''t'' :=
    OThread ta,
 ''x'' := OExpr x))" and q="get_points a" and e'="Var ''l''" and t="''t''" and CFGs=CFGs
 in a_only.not_touches_store1, simp_all)
apply (cut_tac t'="''t''" and CFGs=CFGs and q="(\<lambda>C. \<lfloor>case C of (uu_, p, env
    , stack, allocad) \<Rightarrow> p\<rfloor>) \<circ>\<^sub>m a"
  in LLVM_a_only_CFG.not_touches_bound)
apply (simp add: LLVM_a_only_CFG_def LLVM_threads_def)
apply unfold_locales
apply (assumption, simp, simp)
apply (erule a_only.side_cond_gen, simp+)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (drule AA_correct, simp_all, force, unfold_locales)
apply simp+
apply (frule_tac \<sigma>="((\<lambda>x. OOp add)(''l'' := OExpr (Global list), ''t'' :=
    OThread ta,
 ''x'' := OExpr x))" and q="get_points a" and e'="Var ''l''" and t="''t''" and CFGs=CFGs
 in a_only.not_touches_cmpxchg1, simp_all)
apply (cut_tac t'="''t''" and CFGs=CFGs in LLVM_a_only_CFG.not_touches_bound, simp+)
apply (simp add: LLVM_a_only_CFG_def LLVM_threads_def)
apply (unfold_locales, assumption, simp, simp)
apply (erule a_only.side_cond_gen, simp+)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (drule AA_correct, simp_all, force, unfold_locales)
apply simp+
apply (frule_tac \<sigma>="((\<lambda>x. OOp add)(''l'' := OExpr (Global list), ''t'' :=
    OThread ta,
 ''x'' := OExpr x))" and q="get_points a" and e'="Var ''l''" and t="''t''" and CFGs=CFGs
```

```
 in a_only.not_touches_cmpxchg1, simp_all)
apply (cut_tac t'="''t''" and CFGs=CFGs in LLVM_a_only_CFG.not_touches_bound, simp+)
apply (simp add: LLVM_a_only_CFG_def LLVM_threads_def)
apply (unfold_locales, assumption, simp, simp)
apply (erule a_only.side_cond_gen, simp+)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (drule AA_correct, simp_all, force, unfold_locales)
apply simp+
done

end

(* LLVM_sim.thy *)
(* Simulation relations specialized to the case of MiniLLVM tCFGs. *)
(* William Mansky *)

theory LLVM_sim
imports LLVM_preds trans_sim (*"../../Isabelle/TemporalLogic/sequence"*)
begin

(* Some basic transformation facts. *)
context TRANS_LLVM begin

lemma insert_node: "\<lbrakk>n \<in> Nodes G; n \<noteq> Exit G; CFGs t = Some G\<rbrakk>
    \<Longrightarrow>
  is_flowgraph (G\<lparr>Nodes := insert (SOME n. n \<notin> nodes CFGs) (Nodes G),
                  Edges := rep_edges (Edges G) [n, SOME n. n \<notin> nodes CFGs],
                  Label := (Label G)(n := Load a b c, SOME n. n \<notin> nodes CFGs :=
                      Label G n)\<rparr>)
               seq LLVM_instr_edges"
apply (frule CFGs)
apply (subgoal_tac "(SOME n. n \<notin> nodes CFGs) \<notin> Nodes G")
apply (simp add: is_flowgraph_def, unfold_locales)
apply (clarsimp simp add: flowgraph_def, erule pointed_graph.finite_nodes)
apply (clarsimp simp add: flowgraph_def, erule pointed_graph.finite_edge_types)
apply (clarsimp simp add: flowgraph_def rep_edges_def remap_succ_def, erule disjE, clarsimp
    +)
apply (drule pointed_graph.edges_ok, simp+)
apply (clarsimp split: if_splits)
apply (clarsimp simp add: flowgraph_def rep_edges_def remap_succ_def, drule pointed_graph.
    has_start, simp+)
```

```
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.has_exit, simp+)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.start_not_exit, simp+)
apply (clarsimp simp add: flowgraph_def rep_edges_def remap_succ_def, frule pointed_graph.
    start_first,
  clarsimp simp add: in_edges_def)
apply (drule pointed_graph.has_start, cut_tac A="nodes CFGs" in fresh_new, simp+)
apply (rule nodes_finite)
apply (rule conjI, force simp add: nodes_def ran_def)
apply clarsimp
apply (erule disjE, clarsimp+)
apply (clarsimp simp add: flowgraph_def rep_edges_def remap_succ_def, frule pointed_graph.
    exit_last,
  clarsimp simp add: out_edges_def)
apply (drule pointed_graph.has_exit, cut_tac A="nodes CFGs" in fresh_new, simp+)
apply (rule nodes_finite)
apply (rule conjI, force simp add: nodes_def ran_def)
apply clarsimp+
apply (subgoal_tac "out_edges (TRANS_basics.rep_edges seq (Edges G) [n, SOME n. n \<notin>
    nodes CFGs]) =
  (out_edges (Edges G)) (n := {(SOME n. n \<notin> nodes CFGs, seq)}, SOME n. n \<notin>
      nodes CFGs := out_edges (Edges G) n)")
apply (erule disjE, rule conjI, clarsimp+)
apply (drule flowgraph.instr_edges_ok, simp+)
apply (drule_tac u=u in flowgraph.instr_edges_ok, simp+)
apply (rule conjI, clarsimp)
apply (rule conjI, clarsimp+)
apply (rule ext, simp add: edge_types_def)
apply (clarsimp+)
apply (rule ext, simp)
apply (rule conjI, clarsimp, rule conjI, clarsimp+)
apply (clarsimp simp add: out_edges_def rep_edges_def remap_succ_def)
apply (clarsimp simp add: out_edges_def rep_edges_def remap_succ_def)
apply (rule set_eqI, rule iffI, clarsimp)
apply (case_tac "n = aa", clarsimp+)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.edges_ok, simp_all, clarsimp+)
apply metis
apply (clarsimp simp add: rep_edges_def remap_succ_def, frule_tac u=u in flowgraph.no_loop)
apply (rule conjI, clarsimp+)
apply (rule conjI, clarsimp)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.edges_ok, simp_all, clarsimp+)
```

```
apply (cut_tac A="nodes CFGs" in fresh_new, simp, rule nodes_finite)
apply force
done

end

(* modular simulation *)
lemma (in TRANS_LLVM) not_mods_same1 [simp]: "\<lbrakk>models CFGs \<sigma> q (not_mods t x
    ); \<sigma> t = OThread t';
 CFGs t' = Some G; \<sigma> x = OExpr (Local x'); q t' = Some p; p \<noteq> Exit G;
 one_step t' G ((p0, p, env, r), mem) ((p, p', env', r'), mem')\<rbrakk> \<Longrightarrow>
 env' x' = env x'"
apply (erule one_step.cases, clarsimp simp add: not_mods_def mods_def map_comp_def split:
    if_splits)
apply (case_tac "\<exists>ty e. Label G p = Ret ty e")
apply (cut_tac S="{t}" and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="{t}" and n=2 and m=t in new_nodes_are_new2, simp+)
apply (erule step_cases, auto)
done

context LLVM_threads begin

lemma LLVM_step_mem: "\<lbrakk>step t G mem C ops (p0, p, env, st, al); CFGs t = Some G;
 free_set mem = free_set mem'; \<forall>l v. v \<in> can_read mem t l \<longrightarrow> (\<
     exists>v'. v' \<in> can_read mem' t l)\<rbrakk> \<Longrightarrow>
 \<exists>ops' env'. step t G mem' C ops' (p0, p, env', st, al) \<and> get_loc ` ops' =
     get_loc ` ops"
apply (rule step_cases, simp_all, simp_all)
apply (rule_tac x=env in exI, clarsimp, rule LLVM.assign, rule LLVM_graph, simp+)
apply (rule_tac x=ops in exI, simp, rule_tac x=env in exI, clarsimp, rule LLVM.ret_pop,
    rule LLVM_graph, simp+)
apply (rule_tac x=ops in exI, simp, rule_tac x=env in exI, clarsimp, rule LLVM.ret_main,
    rule LLVM_graph, simp_all, force)
apply (rule conjI)
apply (clarsimp, rule_tac x=env in exI, rule LLVM.branch_false, rule LLVM_graph, simp+)
apply (clarsimp, rule_tac x=env in exI, rule LLVM.branch_true, rule LLVM_graph, simp+)
apply (rule_tac x=env in exI, clarsimp, rule LLVM.branch_u, rule LLVM_graph, simp+)
apply (rule_tac x=ops in exI, simp, rule_tac x=env in exI, clarsimp, rule LLVM.alloca, rule
     LLVM_graph, simp_all)
apply (erule allE, erule impE, force, clarsimp)
```

```
apply (rule_tac x="{Read t l v'}" in exI, simp, rule_tac x="env(x := v')" in exI, rule LLVM
    .load,
 rule LLVM_graph, simp+, force, simp+)
apply (rule_tac x=ops in exI, simp, rule_tac x=env in exI, clarsimp, rule LLVM.store, rule
    LLVM_graph, simp_all, force)
apply (erule allE, erule impE, force, clarsimp)
apply (case_tac "eval_expr env gt e2 = v'")
apply (rule_tac x="{ARW t l (eval_expr env gt e2) (eval_expr env gt e3)}" in exI, simp,
    rule_tac
 x="env(x := eval_expr env gt e2)" in exI, cut_tac stack=st and allocad=al and mem=mem'
 and can_read=can_read in LLVM.cmpxchg_eq, rule LLVM_graph, simp, force, simp+)
apply (rule_tac x="{ARW t l v' v'}" in exI, simp, rule_tac x="env(x := v')" in exI, rule
    LLVM.cmpxchg_no,
 rule LLVM_graph, simp, force, simp+)
apply (erule allE, erule impE, force, clarsimp)
apply (case_tac "eval_expr env gt e2 = v'")
apply (rule_tac x="{ARW t l (eval_expr env gt e2) (eval_expr env gt e3)}" in exI, simp,
 rule_tac x="env(x := eval_expr env gt e2)" in exI, cut_tac stack=st and allocad=al and mem
     =mem'
 and can_read=can_read in LLVM.cmpxchg_eq, rule LLVM_graph, simp, force, simp+)
apply (rule_tac x="{ARW t l v' v'}" in exI, simp, rule_tac x="env(x := v')" in exI, rule
    LLVM.cmpxchg_no,
 rule LLVM_graph, simp, force, simp+)
apply (rule_tac x=env in exI, clarsimp, rule LLVM.icmp, rule LLVM_graph, simp+)
apply (rule_tac x=env in exI, clarsimp, rule LLVM.phi, rule LLVM_graph, simp, force, simp+)
apply (rule_tac x=env in exI, clarsimp, rule LLVM.call, rule LLVM_graph, simp+)
apply (rule_tac x=env in exI, clarsimp, rule LLVM.ispointer, rule LLVM_graph, simp+)
done

lemma LLVM_step_read_ops: "\<lbrakk>step t G mem C ops C'; CFGs t = Some G;
 \<forall>l\<in>get_loc ` ops. can_read mem t l = can_read mem' t l; free_set mem =
     free_set mem';
 L' (fst (snd C)) = Label G (fst (snd C)); is_flowgraph (G\<lparr>Label := L'\<rparr>) seq
     LLVM_instr_edges\<rbrakk> \<Longrightarrow>
 step t (G\<lparr>Label := L'\<rparr>) mem' C ops C'"
apply (drule LLVM_graph')
apply (rule step_cases, simp_all, simp_all)
apply (metis LLVM.assign)
apply (erule LLVM.ret_pop, simp+)
apply (erule LLVM.ret_main, simp_all, force)
```

```
apply (rule conjI)
apply (clarsimp, rule LLVM.branch_false, simp+)
apply (clarsimp, rule LLVM.branch_true, simp+)
apply (erule LLVM.branch_u, simp+)
apply (erule LLVM.alloca, simp_all, clarsimp)
apply (erule LLVM.load, simp, force, simp+, clarsimp)
apply (erule LLVM.store, simp_all, force)
apply (erule LLVM.cmpxchg_eq, simp, force, simp+, clarsimp)
apply (erule LLVM.cmpxchg_no, simp, force, simp+, clarsimp)
apply (erule LLVM.icmp, simp+)
apply (erule LLVM.phi, simp, force, simp+)
apply (erule LLVM.call, simp+)
apply (erule LLVM.ispointer, auto)
done

lemma LLVM_step_read: "\<lbrakk>step t G mem C ops C'; CFGs t = Some G; can_read mem t =
    can_read mem' t;
  free_set mem = free_set mem'; L' (fst (snd C)) = Label G (fst (snd C)); is_flowgraph (G\<
      lparr>Label := L'\<rparr>) seq LLVM_instr_edges\<rbrakk> \<Longrightarrow>
 step t (G\<lparr>Label := L'\<rparr>) mem' C ops C'"
by (metis LLVM_step_read_ops)

end

locale LLVM_CFGs = LLVM_threads where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
  tCFG'!: tCFG where CFGs="CFGs'::('thread, 'node, string) LLVM_tCFG" and instr_edges=
      LLVM_instr_edges
  and Seq=seq for CFGs CFGs'

sublocale LLVM_CFGs \<subseteq> sim?: sim_base where step_rel=step and get_point="\<lambda
    >(_, p, env, stack, allocad). p"
  and instr_edges=LLVM_instr_edges and Seq=seq and start_state="\<lambda>CFGs C0 mem0.
  declare_globals (start_env, start_mem) decls (gt, mem0) \<and> C0 = (\<lambda>t. case
      CFGs t of Some G \<Rightarrow>
  Some (Start G, Start G, start_env, [], {}) | None \<Rightarrow> None)"
apply unfold_locales
apply (drule_tac mem=mem and mem'=mem' in LLVM_step_read_ops, simp+)
apply (erule CFGs, simp)
apply (rule ops_thread, simp+)
done
```

```
locale LLVM_CFGs_SC = TRANS_LLVM_SC where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG" +
  tCFG'!: tCFG where CFGs="CFGs'::('thread, 'node, string) LLVM_tCFG" and instr_edges=
      LLVM_instr_edges
  and Seq=seq for CFGs CFGs'

sublocale LLVM_CFGs_SC \<subseteq> LLVM_CFGs where start_mem=start_mem and free_set=
    free_set
  and can_read=can_read and update_mem=update_mem
by unfold_locales

locale LLVM_CFGs_TSO = TRANS_LLVM_TSO where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG"
     +
  tCFG'!: tCFG where CFGs="CFGs'::('thread, 'node, string) LLVM_tCFG" and instr_edges=
      LLVM_instr_edges
  and Seq=seq for CFGs CFGs'

sublocale LLVM_CFGs_TSO \<subseteq> LLVM_CFGs where start_mem=start_mem and free_set=
    free_set
  and can_read=can_read and update_mem=update_mem
by unfold_locales

locale LLVM_CFGs_PSO = TRANS_LLVM_PSO where CFGs="CFGs::('thread, 'node, string) LLVM_tCFG"
     +
  tCFG'!: tCFG where CFGs="CFGs'::('thread, 'node, string) LLVM_tCFG" and instr_edges=
      LLVM_instr_edges
  and Seq=seq for CFGs CFGs'

sublocale LLVM_CFGs_PSO \<subseteq> LLVM_CFGs where start_mem=start_mem and free_set=
    free_set
  and can_read=can_read and update_mem=update_mem
by unfold_locales

(* assorted useful list lemmas*)
lemma map_upt_zip_Suc [simp]: "l' = map (\<lambda>(x, n). (x, Suc n)) l \<Longrightarrow>
    map (\<lambda>((l, v), n). (l, v) # map (Pair l) (f n)) l' =
 map (\<lambda>((l, v), n). (l, v) # map (Pair l) (f (Suc n))) l"
by auto

declare map_Suc_upt [simp]
```

```
lemma zip_Suc [simp]: "zip l [Suc i..<Suc j] = map (\<lambda>(x, n). (x, Suc n)) (zip l [i
    ..<j])"
by (simp only: zip_map2 [THEN sym], simp)

(* For TSO *)
definition "lift_sim_rel_bufs sim_rel CFGs CFGs' t C C' \<equiv> case (C, C') of ((states,
    mem), (states', mem')) \<Rightarrow>
 (\<exists>s s' G G'. states t = Some s \<and> states' t = Some s' \<and> CFGs t = Some G
     \<and> CFGs' t = Some G' \<and>
 sim_rel G G' (mem, s) (mem', s')) \<and> fst mem = fst mem' \<and>
 (\<forall>t'. t' \<noteq> t \<longrightarrow> states t' = states' t' \<and> snd mem t' =
     snd mem' t')"

context LLVM_TSO begin

lemma can_read_same [intro, cong]: "bufs t = bufs' t \<Longrightarrow>
 can_read (m, bufs) t = can_read (m, bufs') t"
by (clarsimp intro!: ext simp add: can_read_def)

end

lemma ex_spec: "\<lbrakk>\<And>x y. P x y \<Longrightarrow> x = c; \<exists>y. P c y\<
    rbrakk> \<Longrightarrow> \<exists>x y. P x y"
by auto

lemma ex_spec2: "\<lbrakk>\<And>x y. P x y \<Longrightarrow> y = c; \<exists>y. P c y\<
    rbrakk> \<Longrightarrow> \<exists>x y. P x y"
by auto

lemma ex_spec4: "\<lbrakk>\<And>v w x y z. P v w x y z \<Longrightarrow> w = a \<and> x = b
     \<and> y = c \<and> z = d; \<exists>v. P v a b c d\<rbrakk> \<Longrightarrow>
 \<exists>v w x y z. P v w x y z"
by metis

end

(* LLVM_RSE_pre.thy *)
(* Setup for verification of the Redundant Store Elimination optimization. *)
(* William Mansky *)
```

```
theory LLVM_RSE_pre
imports LLVM_sim
begin

context TRANS_LLVM begin

definition "RSE cond \<equiv> TIf [AReplace (MVar ''n'') [Inj (IsPointer (Var ''l''))]]
 (EF (SCPred (Node ''t'' (MVar ''n'')) \<and>sc
 stmt ''t'' (Store (MVar ''ty1'') (Var ''e1'') (MVar ''ty2'') (Var ''l'')) \<and>sc cond))"

lemma RSE_flowgraph [simp, intro!]: "\<lbrakk>Label G n = Store ty1 e1 ty2 e2; CFGs t =
    Some G\<rbrakk> \<Longrightarrow>
 LLVM_threads free_set (CFGs(t \<mapsto> G\<lparr>Label := (Label G)(n := IsPointer e2)\<
     rparr>)) update_mem"
apply (frule CFGs, unfold_locales, auto)
apply (clarsimp split: if_splits)
apply (clarsimp simp add: is_flowgraph_def flowgraph_def)
apply (auto simp add: flowgraph_axioms_def intro!: ext)
apply (erule CFGs)
apply (simp split: if_splits)
apply (cut_tac t=t in disjoint, auto)
apply (cut_tac t=t and t'=ta in disjoint, simp_all, force)
apply (cut_tac t=ta and t'=t' in disjoint, simp_all, force)
done

lemma RSE_paths: "\<lbrakk>Label G n = Store ty1 e1 ty2 e2; CFGs t = Some G\<rbrakk> \<
    Longrightarrow>
 tCFG.Paths (CFGs(t \<mapsto> G\<lparr>Label := (Label G)(n := IsPointer e2)\<rparr>)) =
     Paths"
apply (rule ext, rule set_eqI)
apply (drule RSE_flowgraph, simp, cut_tac CFGs="CFGs(t \<mapsto> G\<lparr>Label := (Label G
    )(n := IsPointer e2)\<rparr>)"
 and Seq=seq and instr_edges=LLVM_instr_edges and q=x in tCFG.Paths_def)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (simp only: Paths_def, clarsimp, rule iffI, clarsimp)
apply (metis (hide_lams, no_types))
apply clarsimp
by (metis (hide_lams, no_types) option.inject)
```

```
lemma RSE_rpaths: "\<lbrakk>Label G n = Store ty1 e1 ty2 e2; CFGs t = Some G\<rbrakk> \<
    Longrightarrow>
 tCFG.RPaths (CFGs(t \<mapsto> G\<lparr>Label := (Label G)(n := IsPointer e2)\<rparr>)) =
     RPaths"
apply (rule ext, rule set_eqI)
apply (drule RSE_flowgraph, simp, cut_tac CFGs="CFGs(t \<mapsto> G\<lparr>Label := (Label G
    )(n := IsPointer e2)\<rparr>)"
 and Seq=seq and instr_edges=LLVM_instr_edges and q=x in tCFG.RPaths_def)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (simp only: RPaths_def, clarsimp, rule iffI, clarsimp, rule conjI, clarsimp)
apply (erule_tac x=i in allE, (erule_tac x=ta in allE)+, clarsimp)
apply (case_tac "ta = t", clarsimp+)
apply (case_tac "ta = t", clarsimp+)
apply (rule conjI, clarsimp)
apply force
by (metis (hide_lams, no_types) the.simps)

lemma LLVM_step_untouched: "\<lbrakk>models CFGs \<sigma> q (not_touches t e); \<sigma> t =
     OThread t';
 expr_pattern_subst \<sigma> e = Some e'; step t' (G\<lparr>Label := L\<rparr>) mem (p0, p,
      env, rest) ops C';
 CFGs t' = Some G; q t' = Some p; Label G p = L p; eval_expr env gt e' = CPointer l;
 is_flowgraph (G\<lparr>Label := L\<rparr>) seq LLVM_instr_edges;
 \<forall>l'. l' \<noteq> l \<longrightarrow> can_read mem t' l' = can_read mem' t' l';
     free_set mem = free_set mem';
 run_prog CFGs C; fst C t' = Some (p0, p, env, rest)\<rbrakk> \<Longrightarrow>
 step t' G mem' (p0, p, env, rest) ops C'"
apply (drule_tac q'="get_points (fst C)" in not_touches_thread, simp+)
apply (rule LLVM.LLVM_step.cases)
apply (rule LLVM_graph', simp_all, simp_all)
apply clarsimp
apply (rule LLVM.assign, rule LLVM_graph, simp+)
apply (rule LLVM.ret_pop, rule LLVM_graph, simp+)
apply (rule LLVM.ret_main, rule LLVM_graph, simp_all, force)
apply (rule conjI)
apply (clarsimp, rule LLVM.branch_false, rule LLVM_graph, simp+)
apply (clarsimp, rule LLVM.branch_true, rule LLVM_graph, simp+)
apply (rule LLVM.branch_u, rule LLVM_graph, simp+)
apply (rule LLVM.alloca, rule LLVM_graph, simp_all, clarsimp)
apply (rule LLVM.load, rule LLVM_graph, simp_all)
```

```
apply (drule not_touches_load1, simp_all)
apply (drule AA_correct, simp+)
apply (unfold_locales, simp+)
apply (rule LLVM.store, rule LLVM_graph, simp_all, force)
apply (rule LLVM.cmpxchg_eq, rule LLVM_graph, simp_all, force)
apply (drule not_touches_cmpxchg1, simp_all)
apply (drule AA_correct, simp+)
apply (unfold_locales, simp+)
apply (rule LLVM.cmpxchg_no, rule LLVM_graph, simp_all, force)
apply (drule not_touches_cmpxchg1, simp_all)
apply (drule AA_correct, simp+)
apply (unfold_locales, simp+)
apply (rule LLVM.icmp, rule LLVM_graph, simp_all)
apply (rule LLVM.phi, rule LLVM_graph, simp, force, simp+)
apply (rule LLVM.call, rule LLVM_graph, simp_all)
apply (rule_tac l=la in LLVM.ispointer, rule LLVM_graph, simp_all)
done

end

declare [[goals_limit=2]]

end

(* LLVM_RSE_SC.thy *)
(* A sample LLVM optimization in PTRANS under SC. *)
(* William Mansky *)

theory LLVM_RSE_SC
imports LLVM_RSE_pre
begin

lemma eval_top: "\<lbrakk>eval_expr start_env gt e = v; v \<noteq> CUndef\<rbrakk> \<
    Longrightarrow> eval_expr env gt e = v"
by (case_tac e, auto)

context TRANS_LLVM_SC begin

definition "RSE_SC_cond \<equiv> SCPred (Other (Ext (Protected (Var ''x'') ''l''))) \<and>
    sc
```

```
 SCPred (Other (Base (GVarlit (Var ''l'')))) \<and>sc SCPred (Other (Base (GVarlit (Var ''x
     '')))) \<and>sc \<not>sc (SCPred (Is ''x'' ''l'')) \<and>sc
 A SCPred (Other (Ext (InCritical ''t'' (Var ''x'') ''l''))) \<and>sc (node ''t'' ''n'' \<
     or>sc not_touches ''t'' (Var ''l'')) \<U>
 (SCPred (Other (Ext (InCritical ''t'' (Var ''x'') ''l''))) \<and>sc (\<not>sc (SCPred (
     Node ''t'' (MVar ''n'')))) \<and>sc
 SCExs [''ty3'', ''e2'', ''ty4''] (stmt ''t'' (Store (MVar ''ty3'') (EPInj (Local ''e2''))
     (MVar ''ty4'') (Var ''l''))))"

lemma RSE_cond_global: "models CFGs \<sigma> q RSE_SC_cond \<Longrightarrow> models CFGs \<
    sigma> q (SCPred (Other (Base (GVarlit (Var ''l'')))))"
by (clarsimp simp add: RSE_SC_cond_def)

lemma RSE_cond_by_thread: "by_thread RSE_SC_cond ''t''"
by (auto simp add: RSE_SC_cond_def)

lemma RSE_cond_step_gen: "\<lbrakk>step t G m C a (fst (snd C), p', rest); models CFGs \<
    sigma> q RSE_SC_cond;
  \<sigma> ''t'' = OThread t; \<sigma> ''n'' = ONode n; q t = Some (fst (snd C)); CFGs t =
      Some G;
  fst (snd C) \<in> Nodes G; fst (snd C) \<noteq> Exit G; \<sigma> ''l'' = OExpr e2; fst (
      snd C) = n \<or>
  (\<forall>ty1 e1 ty2. Label G (fst (snd C)) \<noteq> Store ty1 e1 ty2 e2)\<rbrakk> \<
      Longrightarrow>
 models CFGs \<sigma> (q(t \<mapsto> p')) RSE_SC_cond"
apply (frule RSE_cond_global)
apply (simp only: RSE_SC_cond_def, case_tac C, clarsimp)
apply (drule_tac CFGs=CFGs in step_increment_path, unfold_locales, simp+)
apply (simp add: map_upd_triv, erule_tac x="[q] \<frown> l" in ballE, simp_all, clarsimp)
apply (case_tac e2, simp_all)
apply (case_tac i, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (erule_tac x=yb in allE, force)
apply (rule_tac x=nat in exI, clarsimp, rule conjI, rule_tac x=obj in exI, clarsimp)
apply (rule_tac x=obja in exI, rule_tac x=objb in exI, clarsimp)
apply (metis option.distinct(1))
apply clarsimp
apply (erule_tac x="Suc j" in allE, clarsimp)
done
```

```
corollary RSE_cond_step: "\<lbrakk>step t G m C a (fst (snd C), p', rest); models CFGs \<
    sigma> q RSE_SC_cond; \<sigma> ''t'' = OThread t;
 q t = Some (fst (snd C)); CFGs t = Some G; fst (snd C) \<in> Nodes G; fst (snd C) \<noteq>
      Exit G; \<sigma> ''n'' = ONode n;
 \<sigma> ''l'' = OExpr e2; fst (snd C) = n \<or> (\<forall>ty1 e1 ty2. Label G (fst (snd C
     )) \<noteq> Store ty1 e1 ty2 e2)\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> (q(t \<mapsto> p')) RSE_SC_cond"
by (rule RSE_cond_step_gen, auto)

lemma RSE_fv [simp]: "cond_fv pred_fv RSE_SC_cond = {''t'', ''x'', ''n'', ''l''}"
by (auto simp add: RSE_SC_cond_def not_touches_def)


definition "RSE_SC_sim_rel t G G' C C' \<equiv> case (C, C') of ((s, m), (s', m')) \<
    Rightarrow>
 s = s' \<and> (m = m' \<or> (\<exists>n e2 p ty1 e1 ty2 l v x. n \<in> Nodes G \<and>
     Label G n \<noteq> Label G' n \<and> fst (snd s) = p \<and>
 Label G' n = Store ty1 e1 ty2 e2 \<and> eval_expr start_env gt e2 = CPointer l \<and> m' =
      m(l \<mapsto> v) \<and>
 models CFGs ((\<lambda>x. undefined)(''t'' := OThread t, ''x'' := x, ''l'' := OExpr e2, ''
     n'' := ONode n)) [t \<mapsto> p] RSE_SC_cond))"

end

context LLVM_CFGs_SC begin

theorem RSE_SC_sim: "\<lbrakk>CFGs' \<in> trans_sf (RSE RSE_SC_cond) \<tau> CFGs; CFGs t =
    Some G; CFGs' t = Some G';
 Label G n \<noteq> Label G' n; Label G n = Store ty1 e1 ty2 e2; eval_expr start_env gt e2
     = CPointer l\<rbrakk> \<Longrightarrow>
 tCFG_sim (lift_reach_sim_rel (RSE_SC_sim_rel t) CFGs' CFGs t) (op =) CFGs' CFGs conc_step
     (UNIV - {l}) snd"
apply (subgoal_tac "tCFG CFGs' LLVM_instr_edges seq")
apply (cut_tac get_mem=id in sim_by_reachable_thread_mem_obs, simp_all)
apply (cut_tac A="{''t'', ''l'', ''ty4'', ''e2'', ''ty3'', ''t''}" in fresh_mvars, simp+)
apply (clarsimp simp only: RSE_def trans_sf.simps)
apply clarsimp
apply (case_tac "\<sigma> ''t''", simp_all)
apply (clarsimp simp add: action_list_sf_def split: if_splits)
apply (frule_tac i=i in Path_safe, simp, simp add: safe_points_def)
```

```
apply (erule_tac x=y in allE, clarsimp simp add: thread_of_correct)
apply (frule RSE_paths, simp+)
apply (frule RSE_rpaths, simp+)
apply (cut_tac RSE_flowgraph, simp_all)
apply (frule RSE_cond_global, clarsimp split: LLVM_expr.splits)

apply (unfold_locales, clarsimp simp add: trsys_of_tCFG_def RSE_SC_sim_rel_def split:
    if_splits)
apply (cut_tac CFGs="CFGs(y \<mapsto> G\<lparr>Label := (Label G)(n := IsPointer (Global
    list))\<rparr>)" and t=y and n=n
 in tCFG.label_correct, simp+)
apply (rule one_step.cases, simp, clarsimp, clarsimp simp add: add_reach_def
    RSE_SC_sim_rel_def)
apply (frule_tac t=t in LLVM_threads.LLVM_graph, simp+)
apply (frule run_global, simp+)
apply (frule LLVM_threads.run_global, simp+)
apply (rule ex_spec4, clarsimp, force)
apply (erule disjE, clarsimp)
apply (case_tac "ae = n", clarsimp)
apply (rule LLVM.LLVM_step.cases, simp_all, simp_all, clarsimp)
apply (rule exI, rule_tac x="mem(l \<mapsto> eval_expr env gt e1)" in exI, clarsimp, rule
    context_conjI)
apply (rule_tac ops="{Write ta l (eval_expr env gt e1)}" in step_single)
apply (rule LLVM.store, rule LLVM_graph, simp_all, force, simp)
apply (drule run_prog_one_step, simp+)
apply (drule run_prog_one_step, simp+)
apply (rule_tac x="S(ta \<mapsto> (n, next_node (Edges G) seq n, env, stack, allocad))" in
    exI, clarsimp)
apply (rule_tac x="S'(ta \<mapsto> (n, next_node (Edges G) seq n, env, stack, allocad))" in
     exI, simp)
apply (rule disjI2)
apply (rule conjI, metis)
apply ((erule one_step.cases)+, clarsimp)
apply (cut_tac G=G and C="(a, n, ab, ac, b)" in RSE_cond_step_gen, simp+)
apply (frule_tac i=i in reverse_path, clarsimp)
apply (cut_tac t=t and l=l' and ps="la i" and G=G in step_increment_rpath, simp_all)
apply (unfold_locales, simp+)
apply (frule_tac t=t and l="[la i(t \<mapsto> next_node (Edges G) seq n)] \<frown> l'" in
    tCFG.rpath_one_thread,
  simp+)
```

```
apply (cut_tac q="la i(t \<mapsto> next_node (Edges G) seq n)" and q'="[t \<mapsto>
    next_node (Edges G) seq n]"
 in by_threadD [OF RSE_cond_by_thread], simp_all, simp+)
apply unfold_locales
apply (drule_tac q="[t \<mapsto> next_node (Edges G) seq n]" and \<sigma>'="(\<lambda>x.
    undefined)(''t'' := OThread t,
 ''x'' := \<sigma> ''x'', ''l'' := OExpr (Global list), ''n'' := ONode n)" in side_cond_gen
     , simp+)
apply (unfold_locales, metis)
apply (rule exI, rule_tac x=mem' in exI, clarsimp, rule context_conjI)
apply (rule_tac ops=ops in step_single, simp_all)
apply (drule_tac t=t and G="G\<lparr>Label := (Label G)(n := IsPointer (Global list))\<
    rparr>" and mem=bc
 and mem'=bc in LLVM_threads.LLVM_step_read, simp_all)
apply (erule CFGs)
apply (drule run_prog_one_step, simp+)
apply (drule run_prog_one_step, simp+, force)
apply (clarsimp split: if_splits)
apply (case_tac "ae = n", clarsimp)
apply (rule LLVM.LLVM_step.cases, simp_all, simp_all, clarsimp)
apply (rule exI, rule_tac x="mema(l \<mapsto> eval_expr env gt e1)" in exI, clarsimp, rule
    context_conjI)
apply (rule_tac ops="{Write ta l (eval_expr env gt e1)}" in step_single)
apply (rule LLVM.store, rule LLVM_graph, simp_all, force, simp)
apply (drule run_prog_one_step, simp+)
apply (drule run_prog_one_step, simp+)
apply (rule_tac x="S(ta \<mapsto> (n, next_node (Edges G) seq n, env, stack, allocad))" in
    exI, clarsimp)
apply (rule_tac x="S'(ta \<mapsto> (n, next_node (Edges G) seq n, env, stack, allocad))" in
     exI, simp)
apply (rule disjI2)
apply (rule conjI, metis)
apply ((erule one_step.cases)+, clarsimp)
apply (cut_tac G=G and C="(?vf, n, ?env, ?stack, ?allocad)" in RSE_cond_step_gen, simp+)
apply (frule_tac i=i in reverse_path, clarsimp)
apply (frule_tac t=t and l=l' and ps="la i" and G=G in step_increment_rpath, simp_all)
apply unfold_locales
apply (frule_tac t=t and l="[la i(t \<mapsto> next_node (Edges G) seq n)] \<frown> l'" in
    tCFG.rpath_one_thread, simp+)
```

```
apply (cut_tac q="la i(t \<mapsto> next_node (Edges G) seq n)" and q'="[t \<mapsto>
    next_node (Edges G) seq n]"
 in by_threadD [OF RSE_cond_by_thread], simp_all, simp+)
apply unfold_locales
apply (drule_tac q="[t \<mapsto> next_node (Edges G) seq n]" and \<sigma>'="(\<lambda>x.
    undefined)(''t'' := OThread t,
 ''x'' := \<sigma> ''x'', ''l'' := OExpr (Global list), ''n'' := ONode n)" in side_cond_gen
     , simp+)
apply (unfold_locales, metis)
apply (case_tac "\<forall>t v v'. Write t l v \<notin> ops \<and> ARW t l v v' \<notin> ops
     \<and> Alloc t l \<notin> ops \<and> Free t l \<notin> ops")
apply (rule exI, rule_tac x="mem'(l \<mapsto> y)" in exI, clarsimp, rule context_conjI)
apply (rule_tac ops=ops in step_single, simp_all)
apply (thin_tac "models CFGs \<sigma> (la i) RSE_SC_cond", clarsimp simp add:
    RSE_SC_cond_def)
apply (cut_tac q="[t \<mapsto> ae]" in exists_path, clarsimp)
apply (erule_tac x=laa in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (erule_tac x=yc in allE, clarsimp, erule_tac x=yd in allE, clarsimp)
apply (rule LLVM.LLVM_step.cases, simp_all, simp_all, clarsimp)
apply metis
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (rule_tac L="(Label G)(n := IsPointer (Global list))" in LLVM_step_untouched, simp+)
apply (erule_tac t=t in tCFG.CFGs, simp+)
apply (rule_tac f="op - UNIV" in arg_cong)
apply (rule set_eqI, simp add: dom_def, simp+)
apply (rule update_past, simp+)
apply (drule run_prog_one_step, simp+)
apply (frule run_prog_one_step, simp+)
apply (rule_tac x="S(t \<mapsto> (ap, aq, ar, as, bg))" in exI, clarsimp)
apply (rule_tac x="S'(t \<mapsto> (ap, aq, ar, as, bg))" in exI, simp)
apply (rule disjI2)
apply (rule conjI, metis)
apply (drule one_step_next, erule_tac t=t in tCFG.CFGs, simp+)
apply (erule one_step.cases, clarsimp)
apply (cut_tac G=G and q="[ta \<mapsto> aa]" and p'=ai and C="(a, aa, ab, ac, b)" in
    RSE_cond_step_gen,
  simp+)
apply (frule_tac S=S' in run_prog_safe)
apply (unfold_locales)
```

```
apply (simp add: safe_points_def, (erule_tac x=ta in allE)+, clarsimp, simp+)
apply (erule LLVM.not_exit, simp+)
apply (clarsimp, rule LLVM.LLVM_step.cases, simp_all, simp_all, clarsimp, metis)
apply metis
apply (rule exI, rule_tac x=mem' in exI, clarsimp, rule context_conjI)
apply (rule LLVM.LLVM_step.cases, simp_all, simp_all, clarsimp+)
apply (rule_tac ops="Free tb ` allocad" in step_single, rule LLVM.ret_pop, rule LLVM_graph,
     simp+)
apply (clarsimp simp add: all_alloc_mem_def alloc_mem_def ran_def)
apply (smt access.distinct(13) access.distinct(17) access.distinct(19) access.inject(5)
    image_iff)
apply (cut_tac ops="Free tb ` allocad" in step_single, rule LLVM.ret_main, rule_tac t=tb in
     LLVM_graph,
  simp+)
apply (clarsimp simp add: all_alloc_mem_def alloc_mem_def ran_def)
apply (smt access.distinct(13) access.distinct(17) access.distinct(19) access.inject(5)
    image_iff)
apply clarsimp+
apply (rule_tac ops="{Write tb l (eval_expr env gt e1a)}" in step_single, rule LLVM.store,
 rule LLVM_graph, simp+, force, simp+)
apply (simp add: fun_upd_def)
apply (thin_tac "models CFGs \<sigma> (la i) RSE_SC_cond", clarsimp simp add:
    RSE_SC_cond_def)
apply (cut_tac q="[tb \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=laa in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis domI domIff)
apply ((erule_tac x="0::nat" in allE)+, clarsimp)
apply (drule not_touches_cmpxchg1, simp_all)
apply (cut_tac C="(S', mema(l \<mapsto> y))" in AA_correct, simp_all)
apply (unfold_locales, simp+)
apply (thin_tac "models CFGs \<sigma> (la i) RSE_SC_cond", clarsimp simp add:
    RSE_SC_cond_def)
apply (cut_tac q="[tb \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=laa in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis domI domIff)
apply ((erule_tac x="0::nat" in allE)+, clarsimp)
apply (drule not_touches_cmpxchg1, simp_all)
apply (cut_tac C="(S', mema(l \<mapsto> y))" in AA_correct, simp_all)
```

```
apply (unfold_locales, simp+)
apply (drule_tac CFGs="CFGs(t \<mapsto> G\<lparr>Label := (Label G)(n := IsPointer (Global
    list))\<rparr>)"
  in run_prog_one_step, simp+)
apply (drule run_prog_one_step, simp+)
apply (simp add: fun_upd_def)
apply (rule_tac x="S(t \<mapsto> (ap, aq, ar, as, bg))" in exI, clarsimp)
apply (rule_tac x="S'(t \<mapsto> (ap, aq, ar, as, bg))" in exI, clarsimp)
(* side conditions for other threads *)
apply (clarsimp simp only: RSE_def trans_sf.simps, clarsimp simp add: action_list_sf_def)
apply (case_tac "ab \<notin> nodes CFGs", simp, clarsimp)
apply (clarsimp simp only: RSE_def trans_sf.simps, clarsimp simp add: action_list_sf_def)
apply (case_tac "aj \<notin> nodes CFGs", simp, clarsimp)
apply (clarsimp simp add: RSE_SC_sim_rel_def)
apply (erule disjE, clarsimp+)
apply (rule_tac x=mem2 in exI, simp)
apply (clarsimp split: if_splits)
apply (frule RSE_cond_global, simp)
apply (case_tac e2, clarsimp+)
apply (cut_tac CFGs="CFGs(thread_of n CFGs \<mapsto> G\<lparr>Label := (Label G)(n :=
    IsPointer ap)\<rparr>)"
  in LLVM_threads.run_global)
apply (simp add: LLVM_threads_def LLVM_tCFG_def)
apply (unfold_locales, simp+)
apply clarsimp
apply (rule conjI)
apply (metis insert_dom)
apply clarsimp
apply (thin_tac "models CFGs \<sigma> (la i) RSE_SC_cond")
apply (rule conjI, clarsimp)
apply (clarsimp simp add: RSE_SC_cond_def)
apply (cut_tac t=t' and mem'="mem(get_loc la \<mapsto> v)" in LLVM_step_mem, simp+)
apply (subgoal_tac "get_loc la \<in> dom mem", simp add: insert_absorb, erule domI)
apply clarsimp+
apply (frule_tac t=t' in critical_protects, simp_all, simp_all)
apply unfold_locales
apply force
apply (cut_tac q="[thread_of n CFGs \<mapsto> ae]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (frule_tac t="thread_of n CFGs" and t'=t' and n=ae in mutex, simp_all, simp_all)
```

```
apply (case_tac ia, simp+)
apply (erule_tac x=0 in allE, simp)
apply (force, force)
apply unfold_locales
apply clarsimp
apply (erule_tac update_mem.cases, clarsimp)
apply (case_tac "writes l")
apply (case_tac "\<exists>t. Alloc t l \<in> opsa")
apply (rule_tac x="mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     ++ writes" in exI, simp)
apply (cut_tac mem="mema(l \<mapsto> v)" and ops=opsa and writes=writes in no_atomic, simp
    +)
apply (case_tac "\<forall>t. Free t l \<notin> opsa", simp_all)
apply (subgoal_tac "(mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa} - {l}))(l \<
    mapsto> v) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     = (mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa})
 ++ (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else
     None))", simp, rule ext)
apply (simp add: map_add_def, rule conjI, clarsimp+)
apply (simp add: restrict_map_def)
apply (case_tac "\<exists>t. Free t l \<in> opsa")
apply (rule_tac x="mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     ++ writes" in exI, simp)
apply (cut_tac mem="mema(l \<mapsto> v)" and ops=opsa and writes=writes in no_atomic, simp
    +)
apply (rule_tac x="(mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     ++ writes)(l \<mapsto> v)" in exI, simp, rule conjI)
apply (cut_tac mem="mema(l \<mapsto> v)" and ops=opsa and writes=writes in no_atomic, simp
    +)
apply (subgoal_tac "(mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa} - {l}))(l \<
    mapsto> v) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     ++ writes = (mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa})
 ++ (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else
     None) ++ writes)(l \<mapsto> v)", simp, rule ext, simp)
apply (clarsimp simp add: map_add_def restrict_map_def)
```

```
apply (rule disjI2)
apply smt
apply (rule_tac x="(mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa}) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     ++ writes)" in exI, simp)
apply (cut_tac mem="mema(l \<mapsto> v)" and ops=opsa and writes=writes in no_atomic, simp
    +)
apply (case_tac "\<forall>t. Free t l \<notin> opsa", simp_all)
apply (subgoal_tac "((mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa} - {l}))(l \<
    mapsto> v) ++
 (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else None)
     ++ writes) = (mema |` (UNIV - {l. \<exists>t. Free t l \<in> opsa}))
 ++ (\<lambda>l. if \<exists>t. Alloc t l \<in> opsa then \<lfloor>CUndef\<rfloor> else
     None) ++ writes", simp, rule ext)
apply (clarsimp simp add: map_add_def restrict_map_def)
apply (rule_tac x="mema(l \<mapsto> v, laa \<mapsto> v')" in exI, simp, rule conjI,
    clarsimp)
apply (case_tac "laa = l", simp+)
apply (rule disjI2)
apply (rule conjI)
apply (rule_tac x=v in exI, simp add: fun_upd_twist)
by metis

end

end

(* LLVM_RSE_TSO.thy *)
(* A sample LLVM optimization in PTRANS under TSO. *)
(* William Mansky *)

theory LLVM_RSE_TSO
imports LLVM_RSE_pre
begin

(* The redundant store produces buffers with redundant elements.
   add_red is a general characterization of such buffers. *)
definition "add_red l f =
 concat (map (\<lambda>((l, v), n). (l, v) # map (\<lambda>v. (l, v)) (f n)) (zip l [0..<
     length l]))"
```

```
lemma add_red_nil [simp]: "add_red [] f = []"
by (simp add: add_red_def)

lemma add_red_cons [simp]: "add_red (x # l) f = x # map (\<lambda>v. (fst x, v)) (f 0) @
    add_red l (\<lambda>n. f (Suc n))"
apply (auto simp add: add_red_def)
apply (case_tac "map (\<lambda>((l, v), n). (l, v) # map (Pair l) (f n)) (zip (x # l)
    ([0..<length l] @ [length l]))", auto)
apply (case_tac "[0..<length l] @ [length l]", auto)
apply (case_tac "[0..<length l] @ [length l]", auto)
apply (case_tac "[0..<length l] @ [length l]", auto)
apply (cut_tac i=0 and j="Suc (length l)" and x=ba and xs=list in upt_eq_Cons_conv, auto)
apply (rule_tac f=concat in arg_cong)
apply (rule map_upt_zip_Suc)
by (metis upt_Suc_append zip_Suc)

lemma add_red_app [simp]: "add_red (l @ l') f = add_red l f @ add_red l' (\<lambda>n. f (n
    + length l))"
by (induct l arbitrary: f, auto)

lemma add_red_id [simp]: "(\<And>n. n < length l \<Longrightarrow> f n = []) \<
    Longrightarrow> add_red l f = l"
by (induct l arbitrary: f, auto)

lemma find_append: "(\<And>x. x \<in> set l \<Longrightarrow> \<not>P x) \<Longrightarrow>
    List.find P (l @ l') = List.find P l'"
by (induct l, auto)

lemma find_append2 [simp]: "List.find P l = None \<Longrightarrow> List.find P (l @ l') =
    List.find P l'"
by (induct l, auto)

lemma find_append3 [simp]: "List.find P l = Some x \<Longrightarrow> List.find P (l @ l') =
     Some x"
by (induct l, auto)

lemma add_red_find [simp]: "List.find (\<lambda>(l, v). l = x) (add_red buf f) =
 List.find (\<lambda>(l, v). l = x) buf"
apply (induct buf arbitrary: f, auto)
apply (rule trans, rule find_append, auto)
```

```
done

context TSO begin

lemma update_red: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs'); \<And>t. red t =
    add_red (bufs t) (f t)\<rbrakk> \<Longrightarrow>
 \<exists>red' f'. update_mem (mem, red) ops (mem', red') \<and> (\<forall>t. red' t =
     add_red (bufs' t) (f' t))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). (\<forall>t. red t = add_red (
    bufs t) (f t)) \<longrightarrow>
 (\<exists>red' f'. update_mem (mem, red) ops (mem', red') \<and> (\<forall>t. red' t =
     add_red (bufs' t) (f' t)))"
 in update_mem.induct, auto)
apply (rule_tac x="\<lambda>t. (SOME up. bufs' t = up @ bufsa t \<and> set up = {(l, v).
    Write t l v \<in> ops} \<and>
 distinct up) @ red t" in exI, auto)
apply (rule no_atomic, simp_all)
apply (cut_tac P="\<lambda>up. bufs' t = up @ bufsa t \<and> set up = {(l, v). Write t l v
    \<in> ops} \<and>
 distinct up" in someI_ex, simp+)
apply (rule_tac x="\<lambda>t n. if n < length (SOME up. bufs' t = up @ bufsa t \<and>
 set up = {(l, v). Write t l v \<in> ops} \<and> distinct up) then [] else f t (n - length
     (SOME up.
 bufs' t = up @ bufsa t \<and> set up = {(l, v). Write t l v \<in> ops} \<and> distinct up)
     )" in exI, clarsimp)
apply (subgoal_tac "\<exists>up. bufs' t = up @ bufsa t \<and> set up = {(l, v). Write t l
    v \<in> ops} \<and> distinct up",
 clarify, clarsimp)
apply (cut_tac P="\<lambda>upa. up = upa \<and> set upa = {(l, v). Write t l v \<in> ops}
    \<and> distinct upa" in someI,
 force, clarsimp)
apply metis
apply (drule_tac bufs=red and t=t and a="add_red buf (f' t)" in process_buffer, simp,
    clarsimp)
apply (subgoal_tac "dom (map_of (map (Pair l) (f' t (length buf)))) = {l} \<or> f' t (
    length buf) = []",
 erule disjE, simp+)
apply (rule_tac x="red'(t := add_red buf (f' t))" in exI, clarsimp)
apply (rule_tac x=f' in exI, simp)
apply (rule_tac x="red'(t := add_red buf (f' t))" in exI, clarsimp)
```

```
apply (rule_tac x=f' in exI, simp)
apply (auto simp add: dom_map_of_conv_image_fst intro!: set_eqI)
apply (case_tac "f' t (length buf)", simp+)
apply (rule_tac x=red in exI, auto)+
done

lemma update_red1: "update_mem (mem, bufs) ops (mem', bufs') \<Longrightarrow>
 \<exists>f'. update_mem (mem, bufs(t := add_red (bufs t) f)) ops (mem', bufs'(t := add_red
      (bufs' t) f'))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). \<exists>f'. update_mem (mem,
    bufs(t := add_red (bufs t) f))
 ops (mem', bufs'(t := add_red (bufs' t) f'))" in update_mem.induct, auto)
apply (rule_tac x="\<lambda>n. if n < length (SOME x. bufs' t = x @ bufs t \<and> set x =
    {(l, v). Write t l v \<in> ops} \<and> distinct x)
 then [] else f (n - length (SOME x. bufs' t = x @ bufs t \<and> set x = {(l, v). Write t l
      v \<in> ops} \<and> distinct x))"
 in exI, rule no_atomic, auto)
apply (cut_tac P="\<lambda>up. bufs' t = up @ bufs t \<and> set up = {(l, v). Write t l v
    \<in> ops} \<and>
 distinct up" in someI_ex, force, clarsimp)
apply (rule_tac x="SOME x. bufs' t = x @ bufs t \<and> set x = {(l, v). Write t l v \<in>
    ops} \<and> distinct x"
 in exI, clarsimp)
apply (rule_tac s="add_red ((SOME x. bufs' t = x @ bufs t \<and> set x = {(l, v). Write t l
     v \<in> ops} \<and> distinct x) @ bufs t)
        (\<lambda>n. if n < length (SOME x. bufs' t = x @ bufs t \<and> set x = {(l, v).
            Write t l v \<in> ops} \<and> distinct x) then []
             else f (n - length (SOME x. bufs' t = x @ bufs t \<and> set x = {(l, v). Write
                  t l v \<in> ops} \<and> distinct x)))"
 in trans, simp, simp (no_asm))
apply (drule_tac bufs="bufs(t := add_red (bufs t) f)" and t=t and a="add_red buf f'" in
    process_buffer,
 simp, clarsimp)
apply (subgoal_tac "dom (map_of (map (Pair l) (f' (length buf)))) = {l} \<or> f' (length
    buf) = []",
 auto simp add: dom_map_of_conv_image_fst intro!: set_eqI)
apply (metis (full_types) append_Nil fst_conv imageI in_set_conv_decomp list.exhaust)
apply (drule_tac bufs="bufs(t := add_red (bufs t) f)" in update, simp, force,
 force simp add: fun_upd_twist)
apply force
```

```
done

end

context TRANS_LLVM_TSO begin

lemma step_thread_buf: "\<lbrakk>step t G (mem, b) C ops C'; CFGs t = Some G; b t = b' t\<
    rbrakk> \<Longrightarrow>
 LLVM.LLVM_step (Exit G) (Edges G) free_set can_read (Label G) t (mem, b') gt C ops C'"
apply (rule step_cases, simp_all, simp_all)
apply (rule LLVM.assign, rule LLVM_graph, simp+)
apply (rule LLVM.ret_pop, rule LLVM_graph, simp+)
apply (rule LLVM.ret_main, rule LLVM_graph, simp_all, force)
apply (rule conjI)
apply (clarsimp, rule LLVM.branch_false, rule LLVM_graph, simp+)
apply (clarsimp, rule LLVM.branch_true, rule LLVM_graph, simp+)
apply (rule LLVM.branch_u, rule LLVM_graph, simp+)
apply (rule LLVM.alloca, rule LLVM_graph, simp_all, clarsimp)
apply (rule LLVM.load, rule LLVM_graph, simp, force, simp+, clarsimp, simp)
apply (rule LLVM.store, rule LLVM_graph, simp_all, force)
apply (rule LLVM.cmpxchg_eq, rule LLVM_graph, simp, force, simp+, clarsimp, simp+)
apply (rule LLVM.cmpxchg_no, rule LLVM_graph, simp, force, simp+, clarsimp, simp+)
apply (rule LLVM.icmp, rule LLVM_graph, simp+)
apply (rule LLVM.phi, rule LLVM_graph, simp, force, simp+)
apply (rule LLVM.call, rule LLVM_graph, simp+)
apply (rule LLVM.ispointer, rule LLVM_graph, auto)
done

lemma can_read_red: "b' t = add_red (b t) f \<Longrightarrow> can_read (mem, b') t =
    can_read (mem, b) t"
by (clarsimp intro!: ext simp add: can_read_def)

lemma can_read_red_loc: "\<lbrakk>b' t = (l', v) # add_red (b t) f; l' \<noteq> l\<rbrakk>
    \<Longrightarrow>
 can_read (mem, b') t l = can_read (mem, b) t l"
by (clarsimp intro!: ext simp add: can_read_def)

end

(* Redundant Store Elimination: with alias analysis *)
```

```
definition "RSE_cond \<equiv> A (not_mods ''t'' ''l'' \<and>sc not_loads ''t'' (Var ''l'')
    \<and>sc
 \<not>sc (SCExs [''x'', ''ty1'', ''e1'', ''ty2'', ''e2'', ''ty3'', ''e3'']
 ((stmt ''t'' (Cmpxchg ''x'' (MVar ''ty1'') (Var ''e1'') (MVar ''ty2'') (Var ''e2'') (MVar
     ''ty3'') (Var ''e3''))) \<or>sc
   stmt ''t'' (Store (MVar ''ty1'') (Var ''e1'') (MVar ''ty2'') (Var ''e2''))))) \<U>
 ((\<not>sc (SCPred (Node ''t'' (MVar ''n'')))) \<and>sc SCExs [''ty3'', ''e2'', ''ty4'']
 (stmt ''t'' (Store (MVar ''ty3'') (EPInj (Local ''e2'')) (MVar ''ty4'') (Var ''l'')))))"

context TRANS_LLVM_TSO begin

lemma RSE_cond_gen: "\<lbrakk>models CFGs \<sigma> q RSE_cond; q t = q' t; \<sigma> ''t'' =
     OThread t; \<sigma>' ''t'' = OThread t;
 \<sigma> ''l'' = \<sigma>' ''l''; \<sigma> ''n'' = ONode n; \<sigma>' ''n'' = ONode n\<
     rbrakk> \<Longrightarrow> models CFGs \<sigma>' q' RSE_cond"
apply (clarsimp simp add: RSE_cond_def)
apply (drule_tac q'=q and t=t in path_by_thread, simp, clarsimp)
apply (erule_tac x=l' in ballE, simp_all, clarsimp)
apply (rule_tac x=i in exI, rule conjI, clarsimp)
apply (rule conjI, rule_tac x=obj in exI, clarsimp)
apply (rule_tac x=obja in exI, rule_tac x=objb in exI, clarsimp)
apply (smt object.simps(84) option.distinct(1))
apply clarsimp
apply (erule_tac x=j in allE, clarsimp)
apply (rule conjI, erule not_mods_gen, simp+, rule conjI, erule not_loads_gen, simp+)
apply (clarsimp, case_tac objc, simp_all)
apply (case_tac LLVM_expr, simp_all)
done

corollary RSE_cond_thread: "\<lbrakk>models CFGs \<sigma> q RSE_cond; \<sigma> ''t'' =
    OThread t; \<sigma> ''n'' = ONode n;
 q t = q' t\<rbrakk> \<Longrightarrow> models CFGs \<sigma> q' RSE_cond"
by (rule_tac q=q and \<sigma>=\<sigma> in RSE_cond_gen, simp_all)

lemma RSE_cond_step_gen: "\<lbrakk>step t G m C a (fst (snd C), p', rest); models CFGs \<
    sigma> q RSE_cond;
  \<sigma> ''t'' = OThread t; \<sigma> ''n'' = ONode n; q t = Some (fst (snd C)); fst (snd
      C) \<in> Nodes G; CFGs t = Some G;
 fst (snd C) \<noteq> Exit G; \<sigma> ''l'' = OExpr e2; fst (snd C) = n \<or> (\<forall>
     ty1 e1 ty2. Label G (fst (snd C)) \<noteq> Store ty1 e1 ty2 e2)\<rbrakk> \<
```

```
    Longrightarrow>
 models CFGs \<sigma> (q(t \<mapsto> p')) RSE_cond"
apply (simp only: RSE_cond_def, case_tac C, clarsimp)
apply (drule step_increment_path, unfold_locales, simp+)
apply (simp add: map_upd_triv, erule_tac x="[q] \<frown> l" in ballE, simp_all, clarsimp)
apply (case_tac i, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=ya in allE, force)
apply (rule_tac x=nat in exI, clarsimp, rule conjI, rule_tac x=obj in exI, clarsimp)
apply (rule_tac x=obja in exI, rule_tac x=objb in exI, clarsimp)
apply (metis option.distinct(1))
apply clarsimp
apply (erule_tac x="Suc j" in allE, clarsimp)
apply (case_tac objc, simp_all, case_tac LLVM_expr, simp_all)
done

corollary RSE_cond_step: "\<lbrakk>step t G m C a (fst (snd C), p', rest); models CFGs \<
    sigma> q RSE_cond; \<sigma> ''t'' = OThread t;
 q t = Some (fst (snd C)); CFGs t = Some G; fst (snd C) \<in> Nodes G; fst (snd C) \<noteq>
      Exit G; \<sigma> ''n'' = ONode n;
 \<sigma> ''l'' = OExpr e2; fst (snd C) = n \<or> (\<forall>ty1 e1 ty2. Label G (fst (snd C
     )) \<noteq> Store ty1 e1 ty2 e2)\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> (q(t \<mapsto> p')) RSE_cond"
by (rule RSE_cond_step_gen, auto)

lemma RSE_fv [simp]: "cond_fv pred_fv RSE_cond = {''t'', ''n'', ''l''}"
by (auto simp add: RSE_cond_def not_mods_def returns_def not_loads_def mods_def)

lemma cond_not_touches: "\<lbrakk>models CFGs \<sigma> q (not_loads ''t'' e); models CFGs
    \<sigma> q
 (\<not>sc (SCExs [''x'', ''ty1'', ''e1'', ''ty2'', ''e2'', ''ty3'', ''e3'']
 ((stmt ''t'' (Cmpxchg ''x'' (MVar ''ty1'') (Var ''e1'') (MVar ''ty2'') (Var ''e2'') (MVar
     ''ty3'') (Var ''e3''))) \<or>sc
   stmt ''t'' (Store (MVar ''ty1'') (Var ''e1'') (MVar ''ty2'') (Var ''e2'')))))\<rbrakk>
       \<Longrightarrow>
 models CFGs \<sigma> q (not_touches ''t'' e)"
apply (clarsimp simp only: not_touches_def Let_def models.simps)
apply clarsimp
apply (cut_tac S="insert ''t'' (insert (SOME n. n \<noteq> ''t'' \<and> n \<notin>
    expr_pattern_fv expr_fv e) (expr_pattern_fv expr_fv e))"
```

```
 and n=6 in new_nodes_diff, simp+)
apply (cut_tac S="insert ''t'' (insert (SOME n. n \<noteq> ''t'' \<and> n \<notin>
    expr_pattern_fv expr_fv e) (expr_pattern_fv expr_fv e))"
 and n=2 in new_nodes_diff, simp+)
apply (cut_tac S="insert ''t'' (expr_pattern_fv expr_fv e)" and n=1 and m="''t''" in
    new_nodes_are_new2, simp+)
apply (cut_tac S="insert ''t'' (insert (SOME n. n \<noteq> ''t'' \<and> n \<notin>
    expr_pattern_fv expr_fv e) (expr_pattern_fv expr_fv e))"
 and n=6 and m="''t''" in new_nodes_are_new2, simp+)
apply (cut_tac S="insert ''t'' (insert (SOME n. n \<noteq> ''t'' \<and> n \<notin>
    expr_pattern_fv expr_fv e) (expr_pattern_fv expr_fv e))"
 and n=2 and m="''t''" in new_nodes_are_new2, simp+)
apply (cut_tac S="insert ''t'' (insert (SOME n. n \<noteq> ''t'' \<and> n \<notin>
    expr_pattern_fv expr_fv e) (expr_pattern_fv expr_fv e))"
 and n=6 and m="SOME n. n \<noteq> ''t'' \<and> n \<notin> expr_pattern_fv expr_fv e" in
     new_nodes_are_new2, simp+)
apply (cut_tac S="insert ''t'' (insert (SOME n. n \<noteq> ''t'' \<and> n \<notin>
    expr_pattern_fv expr_fv e) (expr_pattern_fv expr_fv e))"
 and n=2 and m="SOME n. n \<noteq> ''t'' \<and> n \<notin> expr_pattern_fv expr_fv e" in
     new_nodes_are_new2, simp+)
apply (cut_tac S="insert ''t'' (expr_pattern_fv expr_fv e)" and n=1 in new_nodes_are_new,
    simp+)
apply (erule models_ex_disjE)+
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (case_tac "\<sigma> ''t''", simp_all)
apply (erule_tac x=undefined in allE, erule_tac x="OType a" in allE, erule_tac x="OExpr aa"
     in allE,
 erule_tac x="OType ab" in allE, erule_tac x="OExpr ac" in allE, clarsimp, metis)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (case_tac "objs ! 0", simp_all)
apply (case_tac LLVM_expr, simp_all)
apply (erule_tac x="OExpr %var" in allE, erule_tac x="OType a" in allE, erule_tac x="OExpr
    aa" in allE,
 erule_tac x="OType ab" in allE, erule_tac x="OExpr ac" in allE, clarsimp)
apply (erule_tac x="OType ad" in allE, erule_tac x="OExpr ae" in allE, clarsimp)
apply (cut_tac exs_simp, drule_tac P=id in subst, simp only: id_def, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (case_tac "objs ! 0", simp_all)
apply (case_tac LLVM_expr, simp_all)
```

```
apply (clarsimp simp add: not_loads_def)
apply (erule_tac x="OExpr aa" in allE, clarsimp)
apply (erule impE)
apply (rule subst, rule sym, rule exs_simp)
apply (rule_tac x="[OExpr %var, OType a]" in exI, clarsimp+)
apply (case_tac obj, simp_all)
by smt

definition "RSE_tsim_rel2 t G G' C C' \<equiv> case (C, C') of ((s, m, b), (s', m', b')) \<
    Rightarrow>
 s = s' \<and> m = m' \<and> ((\<exists>f. b' = b(t := add_red (b t) f)) \<or>
 (\<exists>n p0 v e2 p env r l ty1 e1 ty2. n \<in> Nodes G \<and> Label G n \<noteq> Label
     G' n \<and> s = (p0, p, env, r) \<and>
 Label G' n = Store ty1 e1 ty2 e2 \<and> eval_expr env gt e2 = CPointer l \<and>
 models CFGs ((\<lambda>x. undefined)(''t'' := OThread t, ''l'' := OExpr e2, ''n'' := ONode
      n)) (empty(t \<mapsto> p)) RSE_cond \<and>
 (\<exists>f. b' = b(t := (l, v) # add_red (b t) f))))"

end

context LLVM_CFGs_TSO begin

theorem RSE2_sim: "\<lbrakk>CFGs' \<in> trans_sf (RSE (AX ''t'' RSE_cond)) \<tau> CFGs;
    CFGs t = Some G; CFGs' t = Some G'; G' \<noteq> G\<rbrakk> \<Longrightarrow>
 tCFG_sim (lift_reach_sim_rel (RSE_tsim_rel2 t) CFGs' CFGs t) (op =) CFGs' CFGs conc_step
     UNIV (fst o snd)"
apply (subgoal_tac "tCFG CFGs' LLVM_instr_edges seq")
apply (rule sim_by_reachable_thread, simp_all)
apply (cut_tac A="{''t'', ''l'', ''ty4'', ''e2'', ''ty3'', ''t''}" in fresh_new, simp+)
apply (clarsimp simp only: RSE_def trans_sf.simps)
apply clarsimp
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (cut_tac A="{''t'', ''n'', ''l''}" in fresh_new, simp+)
apply (case_tac obj, simp_all)
apply (case_tac "\<sigma> ''t''", simp_all)
apply (clarsimp simp add: action_list_sf_def split: if_splits)
apply (frule_tac i=i in Path_safe, simp, simp add: safe_points_def)
apply (erule_tac x=yb in allE, clarsimp simp add: thread_of_correct)
apply (frule RSE_paths, simp+)
```

```
apply (cut_tac RSE_flowgraph, simp_all)

apply (unfold_locales, clarsimp simp add: trsys_of_tCFG_def RSE_tsim_rel2_def split:
    if_splits)
apply (cut_tac CFGs="CFGs(yb \<mapsto> G\<lparr>Label := (Label G)(ya := IsPointer ac)\<
    rparr>)" and t=yb and n=ya
 in tCFG.label_correct)
apply (simp add: LLVM_threads_def LLVM_tCFG_def, force, simp+)
apply (rule one_step.cases, assumption, clarsimp simp add: add_reach_def RSE_tsim_rel2_def)
apply (rule ex_spec, clarsimp, force)
apply (rule ex_spec4, clarsimp, force)
(* The actual modified thread. *)
apply (erule_tac P="\<exists>f. bc = bf(t := add_red (bf t) f)" in disjE, clarsimp)
apply (case_tac "aj = ya", clarsimp)
apply (drule_tac t=t and f=f in update_red1, clarsimp)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, simp_all, clarsimp)
apply (rule exI, rule_tac x="bk(ta := (la, eval_expr env gt aa) # add_red (bk ta) f')" in
    exI,
  rule context_conjI)
apply (rule_tac ops="{Write ta la (eval_expr env gt aa)}" in step_single)
apply (rule LLVM.LLVM_step.store, rule LLVM_graph, simp_all, force)
apply (drule_tac t=ta and l=la and v="eval_expr env gt aa" in update_later, simp)
apply (drule run_prog_one_step, simp+)
apply (frule_tac run_prog_one_step, simp+)
apply (rule_tac x="S(ta \<mapsto> (p, next_node (Edges G) seq p, env, stack, allocad))" in
    exI,
 clarsimp)
apply (rule_tac x="S'(ta \<mapsto> (p, next_node (Edges G) seq p, env, stack, allocad))" in
     exI,
 clarsimp)
apply (erule impE)
apply (frule CFGs, simp add: is_flowgraph_def, frule_tac u=p in flowgraph.no_loop,
 frule_tac u=p in flowgraph.instr_edges_ok, simp+)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.finite_edges, clarsimp)
apply (cut_tac q="(l i)(ta \<mapsto> m)" in exists_path, clarsimp)
apply (cut_tac t=ta and ps="(l i)(ta \<mapsto> m)" and G="G\<lparr>Label := (Label G)(p :=
    IsPointer e)\<rparr>"
  in step_increment_path, simp+)
```

```
apply (simp add: map_upd_triv)
apply (erule_tac x="[l i] \<frown> lb" in ballE, simp_all, erule_tac x=1 in allE, clarsimp)
apply (erule disjE, subgoal_tac "(p, seq) \<in> {(u, t). (p, u, t) \<in> Edges G}", simp,
 simp add: out_edges_def, clarsimp)
apply (erule RSE_cond_gen, simp_all)
apply metis
apply (drule_tac t=t and f=f in update_red1, clarsimp)
apply (rule exI, rule_tac x="bk(t := add_red (bk t) f')" in exI, rule context_conjI)
apply (rule_tac ops=ops in step_single, simp_all)
apply (drule_tac t=t and G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" and
    mem="(am, bf)" and
 mem'="(am, bf(t := add_red (bf t) f))" in LLVM_threads.LLVM_step_read, simp_all)
apply (rule sym, rule can_read_red, simp+)
apply (rule CFGs, simp+)
apply (drule run_prog_one_step, simp+)
apply (frule run_prog_one_step, simp+)
apply (rule_tac x="S(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp)
apply (rule_tac x="S'(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp, metis)
apply (clarsimp split: if_splits)
apply (thin_tac "\<forall>la\<in>Paths (l i). ?P la")
apply (case_tac "aj = ya", clarsimp)
apply (erule notE, clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[t \<mapsto> ya]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (erule_tac x="OExpr ac" in allE, erule_tac x="OType a" in allE, erule_tac x="OExpr aa
    " in allE,
 erule_tac x="OType ab" in allE, erule_tac x="OExpr ac" in allE, clarsimp, force)
apply (case_tac "\<forall>ty1 e1 ty2. Label G aj \<noteq> Store ty1 e1 ty2 ac")
apply (drule_tac t=t and f=f in update_red1, clarsimp)
apply (rule exI, rule_tac x="bk(t := (la, v) # add_red (bk t) f')" in exI, rule
    context_conjI)
apply (rule_tac ops=ops in step_single, simp_all)
apply (clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[t \<mapsto> aj]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
```

```
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (rule_tac L="(Label G)(ya := IsPointer ac)" in LLVM_step_untouched, rule
    cond_not_touches,
 simp+, clarsimp)
apply (case_tac objc, simp_all)
apply (case_tac LLVM_expr, simp_all)
apply clarify
apply (erule_tac x="OExpr %var" in allE, erule_tac x="OType ae" in allE, erule_tac x="OExpr
     af" in allE,
 erule_tac x="OType ag" in allE, erule_tac x="OExpr ah" in allE, clarsimp split: object.
     splits)
apply (drule_tac t=t in tCFG.CFGs, simp+)
apply (clarsimp, rule sym, rule can_read_red_loc, simp+, force, simp+)
apply (drule_tac t=t and b="[(la, v)]" in update_past2, simp+)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, clarsimp+)
apply (clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[ta \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (erule_tac x="OExpr ac" in allE, erule_tac x="OType ty1" in allE, erule_tac x="OExpr
    e1" in allE,
 erule_tac x="OType ty2" in allE, erule_tac x="OExpr e2" in allE, clarsimp, force)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, clarsimp+)
apply (clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[ta \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (erule_tac x="OExpr %x" in allE, erule_tac x="OType ty1" in allE, erule_tac x="OExpr
    e1" in allE,
 erule_tac x="OType ty2" in allE, erule_tac x="OExpr e2" in allE, clarsimp)
apply (erule_tac x="OType ty3" in allE, clarsimp, force)
```

```
apply (clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[ta \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (erule_tac x="OExpr %x" in allE, erule_tac x="OType ty1" in allE, erule_tac x="OExpr
    e1" in allE,
 erule_tac x="OType ty2" in allE, erule_tac x="OExpr e2" in allE, clarsimp)
apply (erule_tac x="OType ty3" in allE, clarsimp, force)
apply (drule run_prog_one_step, simp+)
apply (frule run_prog_one_step, simp+)
apply (rule_tac x="S(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp)
apply (rule_tac x="S'(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp, rule disjI2)
apply (frule_tac t=t in LLVM_threads.LLVM_graph, simp+)
apply (cut_tac Nodes="Nodes G" and Edges="Edges G" and Start="Start G" and Exit="Exit G"
 in pointed_graph.start_not_exit, simp add: LLVM_def LLVM_flowgraph_def flowgraph_def)
apply (case_tac "aj = Exit G", clarsimp)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, clarsimp+)
apply (rule_tac x=v in exI, rule_tac x=la in exI)
apply (rule conjI)
apply (clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[t \<mapsto> aj]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (case_tac ac, simp_all)
apply (drule_tac C="([t \<mapsto> (ai, aj, ak, al, bb)], am, bf(t := (la, v) # add_red (bf
    t) f))"
  in not_mods_same, simp+)
apply (rule ext, simp+)
apply (drule_tac CFGs=CFGs and G=G and states="[t \<mapsto> (ai, aj, ak, al, bb)]"
  in one_step_conc_step, simp+)
apply (force, simp+)
apply (drule CFGs)
```

```
apply (drule one_step_next, simp+, force, simp+)
apply (drule one_step_next, erule_tac t=t in tCFG.CFGs, simp+)
apply (erule one_step.cases, clarsimp)
apply (rule conjI, cut_tac G=G and q="[ta \<mapsto> af]" and C="(ae, af, ag, ah, ba)" and
  \<sigma>="(\<lambda>x. undefined)(''t'' := OThread ta, ''l'' := OExpr ac, ''n'' := ONode
      ya)"
  in RSE_cond_step_gen, simp+)
apply (drule run_prog_safe, unfold_locales)
apply (simp add: safe_points_def, (erule_tac x=ta in allE)+, clarsimp, simp+)
apply (metis (hide_lams, mono_tags))
apply clarsimp
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and G="G\<lparr>Label := (Label G)(ya :=
    IsPointer ac)\<rparr>"
 in LLVM_threads.LLVM_graph, simp_all, simp_all, clarsimp)
apply (drule update_write)
apply (drule_tac t=ta and f="\<lambda>n. case n of 0 \<Rightarrow> [v] | Suc n' \<
    Rightarrow> f n'" in update_red1, clarsimp)
apply (drule_tac ops="{Write ta la (eval_expr env gt e1a)}" and mem=ad and
 bufs="b(ta := (la, v) # add_red (b ta) f)" in update_trans_rev)
apply (rule update_later2, simp_all)
apply (rule exI, rule_tac x="bk(ta := add_red (bk ta) f')" in exI, rule context_conjI)
apply (rule_tac ops="{Write ta la (eval_expr env gt e1a)}" in step_single, simp_all)
apply (rule LLVM.LLVM_step.store, rule LLVM_graph, simp_all, force)
apply (drule run_prog_one_step, simp+)
apply (frule run_prog_one_step, simp+)
apply (smt map_upd_Some_unfold)
(* side conditions for other threads *)
apply (clarsimp simp only: RSE_def trans_sf.simps, clarsimp simp add: action_list_sf_def)
apply (case_tac "ab \<notin> nodes CFGs", simp, clarsimp)
apply (clarsimp simp only: RSE_def trans_sf.simps, clarsimp simp add: action_list_sf_def)
apply (case_tac "al \<notin> nodes CFGs", simp, clarsimp)
apply (clarsimp simp add: RSE_tsim_rel2_def)
apply (rule conjI, clarsimp)
apply (rule sym, rule_tac f="\<lambda>f. []" in can_read_red, simp+)
apply (metis (hide_lams, no_types) fun_upd_apply)
apply clarsimp
apply (cut_tac A="{''t'', ''n'', ''l''}" in fresh_new, simp+)
apply (case_tac "t \<noteq> thread_of al CFGs", clarsimp simp only: if_not_P if_False, simp
    )
apply (erule disjE, clarsimp)
```

```
apply (drule_tac t="thread_of al CFGs" and f=f in update_red1, clarsimp)
apply (rule_tac x="ba(thread_of al CFGs := add_red (ba (thread_of al CFGs)) f')" in exI,
    metis)
apply clarsimp
apply (drule_tac t="thread_of al CFGs" and f=f in update_red1, clarsimp)
apply (rule_tac x="ba(thread_of al CFGs := (la, v) # add_red (ba (thread_of al CFGs)) f')"
    in exI,
  clarsimp split: if_splits)
apply (rule conjI)
apply (drule_tac t="thread_of al CFGs" and b="[(la, v)]" in update_past, simp+)
by metis

end

end

(* LLVM_RSE_PSO.thy *)
(* A sample LLVM optimization in PTRANS under PSO. *)
(* William Mansky *)

theory LLVM_RSE_PSO
imports LLVM_RSE_pre
begin

definition (in TRANS_LLVM_PSO) "RSE_PSO_cond \<equiv> A (not_mods ''t'' ''l'' \<and>sc
    not_touches ''t'' (Var ''l'') \<and>sc
 \<not>sc (SCExs [''x'', ''ty1'', ''e1'', ''ty2'', ''e2'', ''ty3'', ''e3'']
 (stmt ''t'' (Cmpxchg ''x'' (MVar ''ty1'') (Var ''e1'') (MVar ''ty2'') (Var ''e2'') (MVar
     ''ty3'') (Var ''e3''))))) \<U>
 ((\<not>sc (SCPred (Node ''t'' (MVar ''n'')))) \<and>sc SCExs [''ty3'', ''e2'', ''ty4'']
 (stmt ''t'' (Store (MVar ''ty3'') (EPInj (Local ''e2'')) (MVar ''ty4'') (Var ''l'')))))"

(* In PSO, redundant elements are added to the buffers for individual locations. *)
definition "add_red2 l f = concat (map (\<lambda>(v, n). v # f n) (zip l [0..<length l]))"

lemma add_red2_nil [simp]: "add_red2 [] f = []"
by (simp add: add_red2_def)

lemma upt_0 [simp]: "j > i \<Longrightarrow> [i..<j] ! 0 = i"
by (induct i, auto)
```

```
lemma add_red2_nil2 [simp]: "(add_red2 l f = []) = (l = [])"
apply (auto simp add: add_red2_def)
apply (case_tac l, auto)
apply (erule_tac x="(a, 0)" in ballE, auto simp add: set_conv_nth)
apply (erule_tac x=0 in allE, auto)
by (metis gr_implies_not0 le0 upt_0 upt_Suc zero_less_Suc)

lemma map_upt_zip_Suc2 [simp]: "l' = map (\<lambda>(x, n). (x, Suc n)) l \<Longrightarrow>
    map (\<lambda>(v, n). v # f n) l' =
 map (\<lambda>(v, n). v # f (Suc n)) l"
by auto

lemma add_red2_cons [simp]: "add_red2 (x # l) f = x # f 0 @ add_red2 l (\<lambda>n. f (Suc
    n))"
apply (auto simp add: add_red2_def)
apply (case_tac "map (\<lambda>(v, n). v # f n) (zip (x # l) ([0..<length l] @ [length l]))
    ", auto)
apply (case_tac "[0..<length l] @ [length l]", auto)
apply (case_tac "[0..<length l] @ [length l]", auto)
apply (case_tac "[0..<length l] @ [length l]", auto)
apply (cut_tac i=0 and j="Suc (length l)" and x=b and xs=list in upt_eq_Cons_conv, auto)
apply (rule_tac f=concat in arg_cong)
apply (rule map_upt_zip_Suc2)
by (metis upt_Suc_append zip_Suc)

lemma add_red2_app [simp]: "add_red2 (l @ l') f = add_red2 l f @ add_red2 l' (\<lambda>n. f
     (n + length l))"
by (induct l arbitrary: f, auto)

lemma add_red2_id [simp]: "(\<And>n. n < length l \<Longrightarrow> f n = []) \<
    Longrightarrow> add_red2 l f = l"
by (induct l arbitrary: f, auto)

context PSO begin

lemma update_red2: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs');
 \<And>t l. red t l = add_red2 (bufs t l) (f t l)\<rbrakk> \<Longrightarrow>
 \<exists>red' f'. update_mem (mem, red) ops (mem', red') \<and>
 (\<forall>t l. red' t l = add_red2 (bufs' t l) (f' t l))"
```

```
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). (\<forall>t l. red t l =
    add_red2 (bufs t l) (f t l)) \<longrightarrow>
 (\<exists>red' f'. update_mem (mem, red) ops (mem', red') \<and> (\<forall>t l. red' t l =
      add_red2 (bufs' t l) (f' t l)))"
 in update_mem.induct, auto)
apply (rule_tac x="\<lambda>t l. (SOME up. bufs' t l = up @ (bufsa t l) \<and> set up = {v.
     Write t l v \<in> ops} \<and>
 distinct up) @ red t l" in exI, auto)
apply (rule no_atomic, simp_all)
apply (cut_tac P="\<lambda>up. bufs' t l = up @ bufsa t l \<and> set up = {v. Write t l v
    \<in> ops} \<and>
 distinct up" in someI_ex, simp+)
apply (rule_tac x="\<lambda>t l n. if n < length (SOME up. bufs' t l = up @ bufsa t l \<and
    > set up =
 {v. Write t l v \<in> ops} \<and> distinct up) then [] else f t l (n - length (SOME up.
     bufs' t l =
 up @ bufsa t l \<and> set up = {v. Write t l v \<in> ops} \<and> distinct up))" in exI,
     clarsimp)
apply (subgoal_tac "\<exists>up. bufs' t l = up @ bufsa t l \<and> set up = {v. Write t l v
     \<in> ops} \<and> distinct up",
 clarify, clarsimp)
apply (cut_tac P="\<lambda>upa. up = upa \<and> set upa = {v. Write t l v \<in> ops} \<and>
     distinct upa" in someI,
 force, clarsimp)
apply metis
apply (drule_tac bufs=red and t=t and l=l and a="add_red2 buf (f' t l)" in process_buffer,
    force)
apply (rule_tac x="red'(t := (red' t)(l := add_red2 buf (f' t l)))" in exI, clarsimp, metis
    )
apply (rule_tac x=red in exI, auto)
done

lemma update_red2_one_buf: "\<lbrakk>update_mem (mem, bufs) ops (mem', bufs');
 \<And>t'. t' \<noteq> t \<Longrightarrow> red t' = bufs t'; \<And>l. red t l = add_red2 (
     bufs t l) (f l)\<rbrakk> \<Longrightarrow>
 \<exists>red' f'. update_mem (mem, red) ops (mem', red') \<and> (\<forall>t'. t' \<noteq>
     t \<longrightarrow> red' t' = bufs' t') \<and>
 (\<forall>l. red' t l = add_red2 (bufs' t l) (f' l))"
apply (drule_tac P="\<lambda>(mem, bufs) ops (mem', bufs'). ((\<forall>t'. t' \<noteq> t \<
    longrightarrow> red t' = bufs t') \<and>
```

```
(\<forall>l. red t l = add_red2 (bufs t l) (f l))) \<longrightarrow> (\<exists>red' f'.
    update_mem (mem, red) ops (mem', red') \<and>
(\<forall>t'. t' \<noteq> t \<longrightarrow> red' t' = bufs' t') \<and> (\<forall>l. red'
     t l = add_red2 (bufs' t l) (f' l)))"
in update_mem.induct, simp_all, clarsimp)
apply (rule_tac x="bufs'(t := \<lambda>l. (SOME up. bufs' t l = up @ (bufsa t l) \<and> set
    up = {v. Write t l v \<in> ops} \<and>
distinct up) @ red t l)" in exI, clarsimp)
apply (rule conjI, rule no_atomic, simp_all)
apply (cut_tac P="\<lambda>up. bufs' t l = up @ bufsa t l \<and> set up = {v. Write t l v
   \<in> ops} \<and>
distinct up" in someI_ex, simp+)
apply (rule_tac x="\<lambda>l n. if n < length (SOME up. bufs' t l = up @ bufsa t l \<and>
   set up =
{v. Write t l v \<in> ops} \<and> distinct up) then [] else f l (n - length (SOME up. bufs
    ' t l =
up @ bufsa t l \<and> set up = {v. Write t l v \<in> ops} \<and> distinct up))" in exI,
    clarsimp)
apply (subgoal_tac "\<exists>up. bufs' t l = up @ bufsa t l \<and> set up = {v. Write t l v
    \<in> ops} \<and> distinct up",
clarify, clarsimp)
apply (cut_tac P="\<lambda>upa. up = upa \<and> set upa = {v. Write t l v \<in> ops} \<and>
    distinct upa" in someI,
force, clarsimp)
apply metis
apply auto
apply (drule_tac bufs=red and t=t and l=l in process_buffer, force)
apply (rule_tac x="red'(t := (red' t)(l := add_red2 buf (f' l)))" in exI, clarsimp, metis)
apply (drule_tac bufs=red and t=ta and l=l in process_buffer, force)
apply (rule_tac x="red'(ta := (red' ta)(l := buf))" in exI, clarsimp, metis)
apply (rule_tac x=red in exI, auto)
apply (case_tac "ta = t", auto)
done

end

context TRANS_LLVM_PSO begin

definition "RSE_PSO_sim_rel t G G' C C' \<equiv> case (C, C') of ((s, m, b), (s', m', b'))
   \<Rightarrow>
```

```
s = s' \<and> m = m' \<and> (\<forall>t'. t' \<noteq> t \<longrightarrow> b t' = b' t') \<
    and> ((\<exists>f. \<forall>l. b' t l = add_red2 (b t l) (f l)) \<or>
(\<exists>n p0 v e2 p env r l ty1 e1 ty2. n \<in> Nodes G \<and> Label G n \<noteq> Label
    G' n \<and> s = (p0, p, env, r) \<and>
Label G' n = Store ty1 e1 ty2 e2 \<and> eval_expr env gt e2 = CPointer l \<and>
models CFGs ((\<lambda>x. undefined)(''t'' := OThread t, ''l'' := OExpr e2, ''n'' := ONode
     n)) (empty(t \<mapsto> p)) RSE_PSO_cond \<and>
(\<exists>f. b' t l = v # add_red2 (b t l) (f l) \<and> (\<forall>l'. l' \<noteq> l \<
    longrightarrow> b' t l' = add_red2 (b t l') (f l')))))"

lemma RSE_cond_gen: "\<lbrakk>models CFGs \<sigma> q RSE_PSO_cond; q t = q' t; \<sigma> ''t
   '' = OThread t; \<sigma>' ''t'' = OThread t;
\<sigma> ''l'' = \<sigma>' ''l''; \<sigma> ''n'' = ONode n; \<sigma>' ''n'' = ONode n\<
    rbrakk> \<Longrightarrow> models CFGs \<sigma>' q' RSE_PSO_cond"
apply (clarsimp simp add: RSE_PSO_cond_def)
apply (drule_tac q'=q and t=t in path_by_thread, simp, clarsimp)
apply (erule_tac x=l' in ballE, simp_all, clarsimp)
apply (rule_tac x=i in exI, rule conjI, clarsimp)
apply (rule conjI, rule_tac x=obj in exI, clarsimp)
apply (rule_tac x=obja in exI, rule_tac x=objb in exI, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply clarsimp
apply (erule_tac x=j in allE, clarsimp)
apply (rule conjI, erule not_mods_gen, simp+, erule not_touches_gen, simp+)
done

corollary RSE_cond_thread: "\<lbrakk>models CFGs \<sigma> q RSE_PSO_cond; \<sigma> ''t'' =
   OThread t; \<sigma> ''n'' = ONode n;
q t = q' t\<rbrakk> \<Longrightarrow> models CFGs \<sigma> q' RSE_PSO_cond"
by (rule_tac t=t and \<sigma>=\<sigma> and q=q in RSE_cond_gen, simp_all)

lemma RSE_cond_step_gen: "\<lbrakk>step t G m C a (fst (snd C), p', rest); models CFGs \<
   sigma> q RSE_PSO_cond;
 \<sigma> ''t'' = OThread t; CFGs t = Some G; \<sigma> ''n'' = ONode n; q t = Some (fst (
     snd C)); fst (snd C) \<in> Nodes G;
fst (snd C) \<noteq> Exit G; \<sigma> ''l'' = OExpr e2; fst (snd C) = n \<or> (\<forall>
    ty1 e1 ty2. Label G (fst (snd C)) \<noteq> Store ty1 e1 ty2 e2)\<rbrakk> \<
    Longrightarrow>
```

```
 models CFGs \<sigma> (q(t \<mapsto> p')) RSE_PSO_cond"
apply (simp only: RSE_PSO_cond_def, case_tac C, clarsimp)
apply (drule step_increment_path)
apply (unfold_locales, simp+)
apply (simp add: map_upd_triv, erule_tac x="[q] \<frown> l" in ballE, simp_all, clarsimp)
apply (case_tac i, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=ya in allE, force)
apply (rule_tac x=nat in exI, clarsimp, rule conjI, rule_tac x=obj in exI, clarsimp)
apply (rule_tac x=obja in exI, rule_tac x=objb in exI, clarsimp)
apply (metis option.distinct(1))
apply clarsimp
apply (erule_tac x="Suc j" in allE, clarsimp)
done

corollary RSE_cond_step: "\<lbrakk>step t G m C a (fst (snd C), p', rest); models CFGs \<
    sigma> q RSE_PSO_cond; \<sigma> ''t'' = OThread t;
 q t = Some (fst (snd C)); CFGs t = Some G; fst (snd C) \<in> Nodes G; fst (snd C) \<noteq>
      Exit G; \<sigma> ''n'' = ONode n;
 \<sigma> ''l'' = OExpr e2; fst (snd C) = n \<or> (\<forall>ty1 e1 ty2. Label G (fst (snd C
     )) \<noteq> Store ty1 e1 ty2 e2)\<rbrakk> \<Longrightarrow>
 models CFGs \<sigma> (q(t \<mapsto> p')) RSE_PSO_cond"
by (rule RSE_cond_step_gen, auto)

lemma RSE_fv [simp]: "cond_fv pred_fv RSE_PSO_cond = {''t'', ''n'', ''l''}"
by (auto simp add: RSE_PSO_cond_def not_mods_def returns_def not_touches_def mods_def)

lemma can_read_red [simp]: "b' t l = add_red2 (b t l) f \<Longrightarrow>
 can_read (mem, b') t l = can_read (mem, b) t l"
by (clarsimp intro!: ext simp add: can_read_def split: list.splits)

lemma can_read_loc: "b' t l = b t l \<Longrightarrow>
 can_read (mem, b') t l = can_read (mem, b) t l"
by (clarsimp intro!: ext simp add: can_read_def split: list.splits)

lemma two_part_buf [simp]: "bufs t l = buf \<Longrightarrow> bufs(t := (bufs t)(l := buf))
    = bufs"
by (clarsimp intro!: ext)

end
```

```
context LLVM_CFGs_PSO begin

theorem RSE_PSO_sim: "\<lbrakk>CFGs' \<in> trans_sf (RSE (AX ''t'' RSE_PSO_cond)) \<tau>
    CFGs; CFGs t = Some G; CFGs' t = Some G'; G' \<noteq> G\<rbrakk> \<Longrightarrow>
 tCFG_sim (lift_reach_sim_rel (RSE_PSO_sim_rel t) CFGs' CFGs t) (op =) CFGs' CFGs conc_step
      UNIV (fst o snd)"
apply (subgoal_tac "tCFG CFGs' LLVM_instr_edges seq")
apply (rule sim_by_reachable_thread, simp_all)
apply (cut_tac A="{''t'', ''l'', ''ty4'', ''e2'', ''ty3'', ''t''}" in fresh_new, simp+)
apply (clarsimp simp only: RSE_def trans_sf.simps)
apply clarsimp
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (cut_tac A="{''t'', ''n'', ''l''}" in fresh_new, simp+)
apply (case_tac obj, simp_all)
apply (case_tac "\<sigma> ''t''", simp_all)
apply (clarsimp simp add: action_list_sf_def split: if_splits)
apply (frule_tac i=i in Path_safe, simp, simp add: safe_points_def)
apply (erule_tac x=yb in allE, clarsimp simp add: thread_of_correct)
apply (frule RSE_paths, simp+)
apply (cut_tac RSE_flowgraph, simp_all)

apply (unfold_locales, clarsimp simp add: trsys_of_tCFG_def RSE_PSO_sim_rel_def split:
    if_splits)
apply (cut_tac CFGs="CFGs(yb \<mapsto> G\<lparr>Label := (Label G)(ya := IsPointer ac)\<
    rparr>)" and t=yb and n=ya
 in tCFG.label_correct)
apply (simp add: LLVM_threads_def LLVM_tCFG_def, force, simp+)
apply (rule one_step.cases, assumption, clarsimp simp add: add_reach_def
    RSE_PSO_sim_rel_def)
apply (rule ex_spec, clarsimp, force)
apply (rule ex_spec4, clarsimp, force)
(* The actual modified thread. *)
apply (erule_tac P="\<exists>f. \<forall>l. bc t l = add_red2 (bf t l) (f l)" in disjE,
    clarsimp)
apply (case_tac "aj = ya", clarsimp)
apply (drule_tac t=t and f=f in update_red2_one_buf, simp+, clarsimp)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
```

```
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, simp_all, clarsimp)
apply (rule exI, rule_tac x="red'(ta := (red' ta)(la := eval_expr env gt aa # red' ta la))"
     in exI,
  clarsimp, rule context_conjI)
apply (rule_tac ops="{Write ta la (eval_expr env gt aa)}" in step_single)
apply (rule LLVM.LLVM_step.store, rule LLVM_graph, simp_all, force)
apply (drule_tac t=ta and l=la and v="eval_expr env gt aa" in update_later, simp)
apply (drule run_prog_one_step, simp+)
apply (frule_tac run_prog_one_step, simp+)
apply (rule_tac x="S(ta \<mapsto> (p, next_node (Edges G) seq p, env, stack, allocad))" in
    exI,
 clarsimp)
apply (rule_tac x="S'(ta \<mapsto> (p, next_node (Edges G) seq p, env, stack, allocad))" in
     exI,
 clarsimp)
apply (erule impE)
apply (frule CFGs, simp add: is_flowgraph_def, frule_tac u=p in flowgraph.no_loop,
 frule_tac u=p in flowgraph.instr_edges_ok, simp+)
apply (clarsimp simp add: flowgraph_def, drule pointed_graph.finite_edges, clarsimp)
apply (cut_tac q="(l i)(ta \<mapsto> m)" in exists_path, clarsimp)
apply (cut_tac t=ta and ps="(l i)(ta \<mapsto> m)" and G="G\<lparr>Label := (Label G)(p :=
    IsPointer e)\<rparr>"
  in step_increment_path, simp+)
apply (simp add: map_upd_triv)
apply (erule_tac x="[l i] \<frown> lb" in ballE, simp_all, erule_tac x=1 in allE, clarsimp)
apply (erule disjE, subgoal_tac "(p, seq) \<in> {(u, t). (p, u, t) \<in> Edges G}", simp,
 simp add: out_edges_def, clarsimp)
apply (erule RSE_cond_gen, simp_all)
apply (erule_tac x=f' in allE, simp)
apply (drule_tac t=t and f=f in update_red2_one_buf, simp+, clarsimp)
apply (rule exI, rule_tac x=red' in exI, rule context_conjI)
apply (rule_tac ops=ops in step_single, simp_all)
apply (drule_tac t=t and G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" and
    mem="(am, bf)" and
 mem'="(am, bc)" in LLVM_threads.LLVM_step_read, simp_all)
apply (rule ext, simp)
apply (rule CFGs, simp+)
apply (drule run_prog_one_step, simp+)
apply (frule_tac run_prog_one_step, simp+)
```

```
apply (rule_tac x="S(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp)
apply (rule_tac x="S'(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp)
apply (rule disjI1, force)
apply (clarsimp split: if_splits)
apply (thin_tac "\<forall>la\<in>Paths (l i). ?P la")
apply (case_tac "aj = ya", clarsimp)
apply (erule notE, clarsimp simp add: RSE_PSO_cond_def)
apply (cut_tac q="[t \<mapsto> ya]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (drule not_touches_store, simp_all)
apply (case_tac "\<forall>ty1 e1 ty2. Label G aj \<noteq> Store ty1 e1 ty2 ac")
apply (drule_tac t=t and f=f and red="bc(t := (bc t)(la := add_red2 (bf t la) (f la)))"
 in update_red2_one_buf, simp+, clarsimp)
apply (erule_tac x=lb in allE, simp, clarsimp)
apply (rule exI, rule_tac x="red'(t := (red' t)(la := v # red' t la))" in exI, clarsimp,
    rule context_conjI)
apply (rule_tac ops=ops in step_single, simp_all)
apply (clarsimp simp add: RSE_PSO_cond_def)
apply (cut_tac q="[t \<mapsto> aj]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x="0::nat" in allE)+, clarsimp)
apply (erule_tac x=y in allE, clarsimp)
apply (erule_tac x=yb in allE, clarsimp)
apply (rule_tac L="(Label G)(ya := IsPointer ac)" in LLVM_step_untouched, simp+)
apply (erule_tac t=t in tCFG.CFGs, simp+)
apply (clarsimp, erule_tac x=l' in allE, rule sym, rule can_read_red, simp+)
apply (drule_tac t=t and l=la and b="[v]" in update_past2, clarsimp)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, clarsimp+)
apply (clarsimp simp add: RSE_PSO_cond_def)
apply (cut_tac q="[ta \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=la in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
```

```
apply (drule not_touches_store1, simp_all)
apply (drule_tac C="(S', ad, bc)" in AA_correct, simp_all, force)
apply unfold_locales
apply simp+
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, clarsimp+)
apply (clarsimp simp add: RSE_PSO_cond_def)
apply (cut_tac q="[ta \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (erule_tac x="OExpr %x" in allE, erule_tac x="OType ty1" in allE, erule_tac x="OExpr
    e1" in allE,
 erule_tac x="OType ty2" in allE, erule_tac x="OExpr e2" in allE, clarsimp)
apply (erule_tac x="OType ty3" in allE, clarsimp, force)
apply (clarsimp simp add: RSE_PSO_cond_def)
apply (cut_tac q="[ta \<mapsto> p]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (erule_tac x="OExpr %x" in allE, erule_tac x="OType ty1" in allE, erule_tac x="OExpr
    e1" in allE,
 erule_tac x="OType ty2" in allE, erule_tac x="OExpr e2" in allE, clarsimp)
apply (erule_tac x="OType ty3" in allE, clarsimp, force)
apply (drule run_prog_one_step, simp+)
apply (frule run_prog_one_step, simp+)
apply (rule_tac x="S(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp)
apply (rule_tac x="S'(t \<mapsto> (ax, ay, az, bh, bi))" in exI, simp, rule disjI2)
apply (frule_tac t=t in LLVM_threads.LLVM_graph, simp+)
apply (cut_tac Nodes="Nodes G" and Edges="Edges G" and Start="Start G" and Exit="Exit G"
 in pointed_graph.start_not_exit, simp add: LLVM_def LLVM_flowgraph_def flowgraph_def)
apply (case_tac "aj = Exit G", clarsimp)
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and
 G="G\<lparr>Label := (Label G)(ya := IsPointer ac)\<rparr>" in LLVM_threads.LLVM_graph,
     simp_all, clarsimp+)
apply (rule_tac x=v in exI, rule_tac x=la in exI, clarsimp)
apply (rule conjI)
```

```
apply (clarsimp simp add: RSE_PSO_cond_def)
apply (cut_tac q="[t \<mapsto> aj]" in exists_path, clarsimp)
apply (erule_tac x=lb in ballE, simp_all, clarsimp)
apply (case_tac ia, simp)
apply (metis option.distinct(1))
apply ((erule_tac x=0 in allE)+, clarsimp)
apply (case_tac ac, simp_all)
apply (drule_tac C="([t \<mapsto> (ai, aj, ak, al, bb)], am, bc)" in not_mods_same, simp+)
apply (rule ext, simp+)
apply (drule_tac CFGs=CFGs and G=G and states="[t \<mapsto> (ai, aj, ak, al, bb)]"
  in one_step_conc_step, simp+)
apply (force, simp+)
apply (drule CFGs)
apply (drule one_step_next, simp+, force, simp+)
apply (drule one_step_next, erule_tac t=t in tCFG.CFGs, simp+)
apply (erule one_step.cases, clarsimp)
apply (rule conjI, cut_tac G=G and q="[ta \<mapsto> af]" and C="(ae, af, ag, ah, ba)" and
  \<sigma>="(\<lambda>x. undefined)(''t'' := OThread ta, ''l'' := OExpr ac, ''n'' := ONode
      ya)"
  in RSE_cond_step_gen, simp+)
apply (drule run_prog_safe, unfold_locales)
apply (simp add: safe_points_def, (erule_tac x=ta in allE)+, clarsimp, simp+)
apply (metis (hide_lams, mono_tags))
apply clarsimp
apply (rule LLVM.LLVM_step.cases, erule_tac t=t and G="G\<lparr>Label := (Label G)(ya :=
    IsPointer ac)\<rparr>"
 in LLVM_threads.LLVM_graph, simp_all, simp_all, clarsimp)
apply (drule update_write)
apply (drule_tac t=ta and f="\<lambda>l n. if l = la then case n of 0 \<Rightarrow> [v] |
    Suc n' \<Rightarrow> f l n' else f l n"
 and red="bc(ta := (bc ta)(la := eval_expr env gt e1a # bc ta la))" in update_red2_one_buf,
      simp+, metis,
 clarsimp)
apply (drule_tac ops="{Write ta la (eval_expr env gt e1a)}" and mem=ad and
 bufs="bc(ta := (bc ta)(la := v # add_red2 (b ta la) (f la)))" in update_trans_rev)
apply (rule update_later2, simp_all)
apply (rule exI, rule_tac x=red' in exI, rule context_conjI)
apply (rule_tac ops="{Write ta la (eval_expr env gt e1a)}" in step_single, simp_all)
apply (rule LLVM.LLVM_step.store, rule LLVM_graph, simp_all, force)
apply (drule run_prog_one_step, simp+)
```

```
apply (frule run_prog_one_step, simp+)
apply (smt map_upd_Some_unfold)
(* side conditions for other threads *)
apply (clarsimp simp only: RSE_def trans_sf.simps, clarsimp simp add: action_list_sf_def)
apply (case_tac "ab \<notin> nodes CFGs", simp, clarsimp)
apply (clarsimp simp only: RSE_def trans_sf.simps, clarsimp simp add: action_list_sf_def)
apply (case_tac "al \<notin> nodes CFGs", simp, clarsimp)
apply (clarsimp simp add: RSE_PSO_sim_rel_def)
apply (rule conjI, clarsimp, rule ext)
apply (rule can_read_loc, simp+)
apply clarsimp
apply (erule disjE, clarsimp)
apply (drule_tac t=t and f=f in update_red2_one_buf, simp+, clarsimp)
apply (rule_tac x=red' in exI, force)
apply clarsimp
apply (drule_tac t=t and f=f and red="bb(t := (bb t)(la := add_red2 (b t la) (f la)))"
 in update_red2_one_buf, simp+, clarsimp)
apply (rule_tac x="red'(t := (red' t)(la := v # add_red2 (ba t la) (f' la)))" in exI,
    clarsimp)
apply (rule conjI)
apply (drule_tac t=t and b="[v]" and l=la in update_past2)
apply (metis (mono_tags) get_thread.simps(2) imageI)
apply (metis (mono_tags) get_thread.simps(3) imageI)
apply clarsimp
apply (case_tac "t \<noteq> thread_of al CFGs", clarsimp simp only: if_not_P if_False, simp
    )
apply (rule disjI2, clarsimp)
apply (erule_tac x=ya in allE, clarsimp)
apply (case_tac "n \<noteq> al", clarsimp simp only: if_not_P if_False, simp)
by (metis (full_types))

end

end

(* BIL.thy *)
(* William Mansky *)
(* Baby IL for VeriF-OPT. *)

theory BIL
imports trans_flowgraph
```

```
begin

(* syntax *)

datatype 'class BIL_type = tvoid | int32 | tclass 'class ("class")
  | valueclass 'class | tpointer "'class BIL_type" ("_&")

definition "pointerFree a \<equiv> \<not>(\<exists>b. a = b&)"

datatype ('class, 'mname) sig = Sig "'class BIL_type" 'mname "'class BIL_type list"

datatype ('class, 'mname) method_ref = MRef "'class BIL_type" 'class 'mname "'class
    BIL_type list"
  ("_ _::_'(_')" [1000])

datatype 'class constr_ref = KRef 'class "'class BIL_type list" ("void _::.ctor'(_')")

datatype ('class, 'mname, 'field) body = Load int ("ldc.i4 _")
  | Cond "('class, 'mname, 'field) body" "('class, 'mname, 'field) body"
      "('class, 'mname, 'field) body" ("_ _ _ cond")
  | While "('class, 'mname, 'field) body" "('class, 'mname, 'field) body" ("_ _ while")
  | Seq "('class, 'mname, 'field) body" "('class, 'mname, 'field) body"
  | Loadind "('class, 'mname, 'field) body" ("_ ldind" 20)
  | Storeind "('class, 'mname, 'field) body" "('class, 'mname, 'field) body" ("_ _ stind")
  | Loadarga nat ("ldarga _" [1000] 30)
  | Storearg "('class, 'mname, 'field) body" nat ("_ starg _")
  | Newobj "('class, 'mname, 'field) body list" "'class constr_ref" ("_ newobj _")
  | Callboxed "('class, 'mname, 'field) body" "('class, 'mname, 'field) body list"
      "('class, 'mname) method_ref" ("_ _ callvirt _")
  | Callunboxed "('class, 'mname, 'field) body" "('class, 'mname, 'field) body list"
      "('class, 'mname) method_ref" ("_ _ call instance _")
  | Loadflda "('class, 'mname, 'field) body" "'class BIL_type" 'class 'field
      ("_ ldflda _ _::_" [0, 0, 0, 1000] 30)
  | Storefld "('class, 'mname, 'field) body" "('class, 'mname, 'field) body" "'class
      BIL_type"
    'class 'field ("_ _ stfld _ _::_")
  | Box "('class, 'mname, 'field) body" 'class ("_ box _")
  | Unbox "('class, 'mname, 'field) body" 'class ("_ unbox _")

definition Loadfld ("_ ldfld _ _::_") where "a ldfld t c::f \<equiv> a ldflda t c::f ldind"
```

```
definition Loadarg ("ldarg _") where "ldarg j \<equiv> ldarga j ldind"

definition "fdom fs \<equiv> fst ` set fs"
definition "get fs f \<equiv> case List.find (\<lambda>x. fst x = f) fs of Some (a, b) \<
    Rightarrow> Some b | None \<Rightarrow> None"

locale BIL = fixes ValueClass::"'class set" and Object::'class
  and fields::"'class \<Rightarrow> ('field \<times> 'class BIL_type) list"
  and methods::"'class \<Rightarrow> ('class, 'mname) sig \<rightharpoonup> ('class, 'mname
      , 'field) body"
  and inherits::"'class rel"
  assumes Hi_Refl: "refl inherits" and Hi_Trans: "trans inherits" and Hi_Antisym: "antisym
      inherits"
  and Hi_Root: "(c, Object) \<in> inherits"
  and Hi_Fields: "\<lbrakk>(c, d) \<in> inherits; f \<in> fdom (fields d)\<rbrakk> \<
      Longrightarrow>
    f \<in> fdom (fields c) \<and> get (fields c) f = get (fields d) f"
  and Hi_methods: "(c, d) \<in> inherits \<Longrightarrow> dom (methods d) \<subseteq> dom
      (methods c)"
  and Hi_val: "\<lbrakk>(c, vc) \<in> inherits; vc \<in> ValueClass\<rbrakk> \<
      Longrightarrow> c = vc"
  and Good_fields: "get (fields c) f = Some a \<Longrightarrow> pointerFree a"
  and Good_methods: "Sig b l al \<in> dom (methods c) \<Longrightarrow> pointerFree b"
      begin

abbreviation subtype (infix "<:" 100) where "a <: b \<equiv> (a, b) \<in> inherits"

end

(* memory model (layout) *)
datatype ('href, 'field) pointer = PBoxed 'href | FrameArg nat nat
  | FieldRef "('href, 'field) pointer" 'field ("_._" [900, 1000] 900)

datatype ('href, 'field) result = IntResult int | PtrResult "('href, 'field) pointer"
  | ObjResult "'field \<rightharpoonup> ('href, 'field) result"

type_synonym ('href, 'field) unboxed_obj = "'field \<rightharpoonup> ('href, 'field) result
    "

datatype ('class, 'href, 'field) boxed_obj = BoxedObj 'class "('href, 'field) unboxed_obj"
```

```
type_synonym ('class, 'href, 'field) heap = "'href \<rightharpoonup> ('class, 'href, 'field
    ) boxed_obj"

datatype ('href, 'field) frame = Frame "('href, 'field) result list" (".args'(_')")

type_synonym ('href, 'field) stack = "('href, 'field) frame list"

type_synonym ('class, 'href, 'field) store = "('class, 'href, 'field) heap \<times> ('href,
     'field) stack"

term "[p \<mapsto> BoxedObj c [f1 \<mapsto> IntResult 0, f2 \<mapsto> ObjResult [g \<mapsto
    > IntResult 1]]]"
term "[.args([PtrResult (PBoxed p), PtrResult ((PBoxed p).(f2).g)]),
       .args([PtrResult (PBoxed p), PtrResult (FrameArg 1 1)])]"

fun lookup0::"('class, 'href, 'field) store \<Rightarrow> ('href, 'field) pointer \<
    rightharpoonup> ('href, 'field) result" where
"lookup0 (h, s) (PBoxed p) = (case h p of Some (BoxedObj c u) \<Rightarrow> Some (ObjResult
     u) | _ \<Rightarrow> None)" |
"lookup0 (h, s) (FieldRef p f) = (case lookup0 (h, s) p of Some (ObjResult u) \<Rightarrow>
     u f | _ \<Rightarrow> None)" |
"lookup0 (h, s) (FrameArg i j) = (case s ! (i - 1) of .args(vs) \<Rightarrow> Some (vs ! (j
     - 1)))"

fun get_base::"('href, 'field) pointer \<Rightarrow> ('href, 'field) pointer" where
"get_base (FieldRef p f) = get_base p" |
"get_base p = p"

lemma get_base_cases: "(\<exists>p. get_base ptr = PBoxed p) \<or> (\<exists>i j. get_base
    ptr = FrameArg i j)"
by (induct ptr, auto)

fun is_heap where
"is_heap p = (case get_base p of PBoxed _ \<Rightarrow> True | _ \<Rightarrow> False)"

fun get_fields::"('href, 'field) pointer \<Rightarrow> 'field list" where
"get_fields (FieldRef p f) = get_fields p @ [f]" |
"get_fields _ = []"
```

```
fun rlookup::"('href, 'field) result \<Rightarrow> 'field list \<rightharpoonup> ('href, '
    field) result" where
"rlookup v [] = Some v" |
"rlookup (ObjResult u) (f # fs) = (case u f of Some v \<Rightarrow> rlookup v fs | _ \<
    Rightarrow> None)" |
"rlookup _ _ = None"

fun lookup::"('class, 'href, 'field) store \<Rightarrow> ('href, 'field) pointer \<
    rightharpoonup> ('href, 'field) result" where
"lookup (h, s) p = (let (p0, fs) = (get_base p, get_fields p) in
  case p0 of PBoxed pb \<Rightarrow> (case h pb of Some (BoxedObj c u) \<Rightarrow>
      rlookup (ObjResult u) fs | _ \<Rightarrow> None)
  | FrameArg i j \<Rightarrow> case s ! (i - 1) of .args(vs) \<Rightarrow> rlookup (vs ! (j
      - 1)) fs)"

lemma rlookup_None_add_field [simp]: "rlookup v fs = None \<Longrightarrow> rlookup v (fs @
     [f]) = None"
apply (induct fs arbitrary: v, simp+)
apply (case_tac v, simp_all split: option.splits)
done

lemma rlookup_Some_add_field [simp]: "rlookup v fs = Some v' \<Longrightarrow> rlookup v (
    fs @ fs') = rlookup v' fs'"
apply (induct fs arbitrary: v fs', simp+)
apply (case_tac v, simp_all split: option.splits)
done

lemma "lookup \<sigma> p = lookup0 \<sigma> p"
apply (case_tac \<sigma>, clarsimp)
apply (induct p, simp_all split: option.splits boxed_obj.splits frame.splits)
apply (cut_tac ptr=p in get_base_cases, auto split: option.splits boxed_obj.splits frame.
    splits)
apply (case_tac aa, simp_all split: option.splits)+
done

fun rupdate::"('href, 'field) result \<Rightarrow> 'field list \<Rightarrow> ('href, 'field
    ) result \<rightharpoonup>
  ('href, 'field) result" where
"rupdate v [] v' = Some v'" |
```

```
"rupdate (ObjResult u) (f # fs) v' = (case u f of Some v \<Rightarrow> (case rupdate v fs v
    ' of Some v'' \<Rightarrow>
  Some (ObjResult (u(f \<mapsto> v''))) | _ \<Rightarrow> None) | _ \<Rightarrow> None)" |
"rupdate _ _ _ = None"

fun update::"('class, 'href, 'field) store \<Rightarrow> ('href, 'field) pointer \<
    Rightarrow> ('href, 'field) result \<rightharpoonup>
  ('class, 'href, 'field) store" where
"update (h, s) p v' = (let (p0, fs) = (get_base p, get_fields p) in
  case p0 of PBoxed pb \<Rightarrow> (case h pb of Some (BoxedObj c u) \<Rightarrow>
    (case rupdate (ObjResult u) fs v' of Some (ObjResult u') \<Rightarrow> Some (h(pb \<
        mapsto> BoxedObj c u'), s) | _ \<Rightarrow> None) | _ \<Rightarrow> None)
  | FrameArg i j \<Rightarrow> case s ! (i - 1) of .args(vs) \<Rightarrow>
    (let j' = if j - 1 < length vs then j - 1 else length vs - 1 in
      case rupdate (vs ! j') fs v' of Some v'' \<Rightarrow>
        Some (h, s[i - 1 := .args(vs[j' := v''])]) | _ \<Rightarrow> None))"

(* semantics *)
context BIL begin

inductive eval::"('class, 'href, 'field) store \<Rightarrow> ('class, 'mname, 'field) body
    \<Rightarrow>
  ('href, 'field) result list \<Rightarrow> ('class, 'href, 'field) store \<Rightarrow>
      bool" ("_ \<turnstile> _ \<leadsto> _ \<cdot> _") where
(* control flow *)
eval_ldc [intro!]: "\<sigma> \<turnstile> ldc.i4 i \<leadsto> [IntResult i] \<cdot> \<sigma
    >" |
eval_seq [intro]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> u \<cdot> \<sigma>'; \<sigma
    >' \<turnstile> b \<leadsto> v \<cdot> \<sigma>''\<rbrakk> \<Longrightarrow> \<sigma> \<
    turnstile> Seq a b \<leadsto> v \<cdot> \<sigma>''" |
eval_cond [intro]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [i] \<cdot> \<sigma>'; \<
    sigma>' \<turnstile> (if i = IntResult 0 then b else c) \<leadsto> v \<cdot> \<sigma
    >''\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a b c cond \<leadsto> v \<cdot> \<sigma>''" |
eval_while0 [intro]: "\<sigma> \<turnstile> a \<leadsto> [IntResult 0] \<cdot> \<sigma>' \<
    Longrightarrow> \<sigma> \<turnstile> a b while \<leadsto> [] \<cdot> \<sigma>'" |
eval_while1 [intro]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [i] \<cdot> \<sigma>'; i
    \<noteq> IntResult 0; \<sigma>' \<turnstile> b \<leadsto> v \<cdot> \<sigma>''; \<sigma
    >'' \<turnstile> a b while \<leadsto> u \<cdot> \<sigma>'''\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a b while \<leadsto> u \<cdot> \<sigma>'''" |
```

```
(* pointer types *)
eval_ldind [intro!]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [PtrResult ptr] \<cdot>
    \<sigma>'; lookup \<sigma>' ptr = Some v\<rbrakk> \<Longrightarrow> \<sigma> \<turnstile
    > a ldind \<leadsto> [v] \<cdot> \<sigma>'" |
eval_stind [intro]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [PtrResult ptr] \<cdot> \<
    sigma>'; \<sigma>' \<turnstile> b \<leadsto> [v] \<cdot> \<sigma>''; update \<sigma>''
    ptr v = Some \<sigma>'''\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a b stind \<leadsto> [] \<cdot> \<sigma>'''" |
(* arguments *)
eval_ldarga [intro!]: "length (snd \<sigma>) = i \<Longrightarrow> \<sigma> \<turnstile>
    ldarga j \<leadsto> [PtrResult (FrameArg i j)] \<cdot> \<sigma>" |
eval_starg [intro]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [u] \<cdot> \<sigma>';
    length (snd \<sigma>') = i; update \<sigma>' (FrameArg i j) u = Some \<sigma>''\<rbrakk>
     \<Longrightarrow>
  \<sigma> \<turnstile> a starg j \<leadsto> [] \<cdot> \<sigma>''" |
(* for reference types only *)
eval_newobj [intro]: "\<lbrakk>K = void c::.ctor(As); c \<notin> ValueClass; n = length
    args;
  length (fields c) = n; length \<sigma>s = Suc n; length vs = n;
  \<forall>i<n. (\<sigma>s ! i) \<turnstile> args ! i \<leadsto> [vs ! i] \<cdot> \<sigma>s
       ! (Suc i); fst (\<sigma>s ! n) = h; p \<notin> dom h;
  h' = h(p \<mapsto> BoxedObj c (map_of (zip (map fst (fields c)) vs))); \<sigma> = \<sigma
      >s ! 0\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> args newobj K \<leadsto> [PtrResult (PBoxed p)] \<cdot> (h', snd
      (\<sigma>s ! n))" |
eval_callvirt [intro]: "\<lbrakk>M = B c::l(As); length args = n; length stores = Suc n;
    length vs = n;
  \<sigma> \<turnstile> a \<leadsto> [PtrResult (PBoxed p)] \<cdot> (stores ! 0); fst (
      stores ! 0) p = Some (BoxedObj c' u);
  \<forall>i<n. ((stores ! i) \<turnstile> args ! i \<leadsto> [vs ! i] \<cdot> (stores ! (
      Suc i))); methods c' (Sig B l As) = Some b;
  stores ! n = (h, s); (h, s @ [.args(PtrResult (PBoxed p) # vs)]) \<turnstile> b \<leadsto
      > v' \<cdot> (h', s' @ [fr'])\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a args callvirt M \<leadsto> v' \<cdot> (h', s')" |
(* for reference and value types *)
eval_ldflda [intro!]: "\<sigma> \<turnstile> a \<leadsto> [PtrResult ptr] \<cdot> \<sigma>'
     \<Longrightarrow> \<sigma> \<turnstile> a ldflda t c::f \<leadsto> [PtrResult ((ptr).f)
    ] \<cdot> \<sigma>'" |
eval_stfld [intro]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [PtrResult ptr] \<cdot> \<
    sigma>'; \<sigma>' \<turnstile> b \<leadsto> [v] \<cdot> \<sigma>''; update \<sigma>''
```

```
    ((ptr).f) v = Some \<sigma>'''\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a b stfld t c::f \<leadsto> [] \<cdot> \<sigma>'''" |
(* for value types only *)
eval_newobjv [intro]: "\<lbrakk>K = void vc::.ctor(As); vc \<in> ValueClass; n = length
    args;
  length (fields c) = n; length \<sigma>s = Suc n; length vs = n;
  \<forall>i<n. (\<sigma>s ! i) \<turnstile> args ! i \<leadsto> [vs ! i] \<cdot> (\<sigma>
      s ! Suc i); \<sigma> = \<sigma>s ! 0\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> args newobj K \<leadsto> [ObjResult (map_of (zip (map fst (fields c
      )) vs))] \<cdot> \<sigma>s ! n" |
eval_call [intro]: "\<lbrakk>M = B vc::l(As); vc \<in> ValueClass; length args = n; length
    stores = Suc n;
  length vs = n; \<sigma> \<turnstile> a \<leadsto> [PtrResult ptr] \<cdot> (stores ! 0);
  \<forall>i<n. ((stores ! i) \<turnstile> args ! i \<leadsto> [vs ! i] \<cdot> (stores ! (
      Suc i))); methods vc (Sig B l As) = Some b;
  stores ! n = (h, s); (h, s @ [.args(PtrResult ptr # vs)]) \<turnstile> b \<leadsto> v' \<
      cdot> (h', s' @ [fr'])\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a args call instance M \<leadsto> v' \<cdot> (h', s')" |
(* boxing and unboxing *)
eval_box [intro]: "\<lbrakk>p \<notin> dom h'; \<sigma> \<turnstile> a \<leadsto> [
    PtrResult ptr] \<cdot> (h', s');
  lookup (h', s') ptr = Some (ObjResult u); vc \<in> ValueClass\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a box vc \<leadsto> [PtrResult (PBoxed p)] \<cdot> (h'(p \<mapsto>
      BoxedObj vc u), s')" |
eval_unbox [intro!]: "\<lbrakk>\<sigma> \<turnstile> a \<leadsto> [PtrResult (PBoxed p)] \<
    cdot> \<sigma>'; vc \<in> ValueClass\<rbrakk> \<Longrightarrow>
  \<sigma> \<turnstile> a unbox vc \<leadsto> [PtrResult (PBoxed p)] \<cdot> \<sigma>'"

end

end

theory BIL_graph
imports BIL trans_semantics trans_preds step_rel
begin

no_notation Load ("ldc.i4 _")
no_notation Loadind ("_ ldind" 20)
no_notation Storeind ("_ _ stind")
no_notation Loadarga ("ldarga _" [1000] 30)
no_notation Storearg ("_ starg _")
```

```
no_notation Newobj ("_ newobj _")
no_notation Callboxed ("_ _ callvirt _")
no_notation Callunboxed ("_ _ call instance _")
no_notation Loadflda ("_ ldflda _ _::_" [0, 0, 0, 1000] 30)
no_notation Storefld ("_ _ stfld _ _::_")
no_notation Box ("_ box _")
no_notation Unbox ("_ unbox _")

(* Is there a general method for this translation? *)
datatype ('class, 'type, 'field, 'int_val, 'constr, 'method) instr =
    ldc_i4 'int_val ("ldc.i4 _")
  | br
  | brtrue
  | brfalse
  | ldind
  | stind
  | ldarga 'int_val
  | starg 'int_val
  | newobj 'constr
  | callvirt 'method
  | callinstance 'method ("call instance")
  | ret
  | Ldflda 'type 'class 'field ("ldflda _ _::_" [0, 0, 1000] 30)
  | Stfld 'type 'class 'field ("stfld _ _::_")
  | box 'class
  | unbox 'class
  | dup
  | pop

type_synonym ('class, 'mname, 'field) c_instr = "('class, 'class BIL_type, 'field, int,
  'class constr_ref, ('class, 'mname) method_ref) instr"

datatype 'class edge_type = seq | branch | mcall 'class

lemma finite_edge_types [simp]: "finite (UNIV::'class edge_type set) = finite (UNIV::'class
     set)"
apply auto
apply (drule_tac A="range mcall" in rev_finite_subset, simp)
apply (drule finite_imageD, auto simp add: inj_on_def)
apply (rule_tac s="{seq, branch} \<union> range mcall" in subst, auto)
```

```
apply (case_tac x, auto)
done

record ('node, 'edge_type, 'instr) graph_part = "('node, 'edge_type) doubly_pointed_graph"
    +
 pLabel::"'node \<rightharpoonup> 'instr"
definition "finish_CFG G \<equiv> \<lparr>Nodes = Nodes G, Edges = Edges G, Start = Start G
    , Exit = Exit G,
  Label = \<lambda>n. case pLabel G n of Some i \<Rightarrow> i | None \<Rightarrow> br\<
      rparr>"

type_synonym ('node, 'class, 'mname, 'field) BIL_graph =
  "('node, 'class edge_type, ('class, 'mname, 'field) c_instr) flowgraph"
type_synonym ('node, 'class, 'mname, 'field) BIL_part =
  "('node, 'class edge_type, ('class, 'mname, 'field) c_instr) graph_part"

(* This is, in a form, our long-awaited algebra of graphs. *)
definition "only n n' i \<equiv> \<lparr>Nodes = {n, n'}, Edges = {(n, n', seq)}, Start = n
    , Exit = n',
  pLabel = [n \<mapsto> i]\<rparr>"
definition "sequence G1 G2 \<equiv> \<lparr>Nodes = Nodes G1 \<union> Nodes G2, Edges = (
    Edges G1 \<union> Edges G2),
  Start = Start G1, Exit = Exit G2, pLabel = pLabel G1 ++ pLabel G2\<rparr>"
definition "follow G n i \<equiv> \<lparr>Nodes = insert n (Nodes G), Edges = insert (Exit
    G, n, seq) (Edges G),
  Start = Start G, Exit = n, pLabel = pLabel G(Exit G \<mapsto> i)\<rparr>"

lemma only_graph [intro!]: "\<lbrakk>n \<noteq> n'; finite (UNIV::'class set)\<rbrakk> \<
    Longrightarrow> is_doubly_pointed_graph
  ((only n n' (i::('class, 'b, 'c) c_instr))::('node, 'class edge_type, ('class, 'b, 'c)
      c_instr) graph_part)"
by (clarsimp simp add: pointed_graph_def in_edges_def out_edges_def only_def)

lemma sequence_graph [intro!]: "\<lbrakk>is_doubly_pointed_graph G; is_doubly_pointed_graph
     G'; Start G \<noteq> Exit G';
  Start G \<notin> Nodes G'; Exit G' \<notin> Nodes G\<rbrakk> \<Longrightarrow>
      is_doubly_pointed_graph (sequence G G')"
by (clarsimp simp add: pointed_graph_def in_edges_def out_edges_def sequence_def, metis)
```

```
lemma follow_graph [intro!]: "\<lbrakk>is_doubly_pointed_graph G; n \<notin> Nodes G\<
    rbrakk> \<Longrightarrow>
  is_doubly_pointed_graph (follow G n i)"
by (clarsimp simp add: pointed_graph_def in_edges_def out_edges_def follow_def, metis)

lemma follow_simp [simp]: "Exit G \<in> Nodes G \<Longrightarrow> follow G n i = sequence G
     (only (Exit G) n i)"
by (simp add: follow_def sequence_def only_def insert_absorb)

lemma follow_sequence [simp]: "sequence G (follow G' n i) = follow (sequence G G') n i"
by (simp add: follow_def sequence_def)

lemma sequence_assoc [simp]: "sequence (sequence G1 G2) G3 = sequence G1 (sequence G2 G3)"
by (auto simp add: sequence_def)

lemma sequence_nodes [simp]: "Nodes (sequence G1 G2) = Nodes G1 \<union> Nodes G2"
by (simp add: sequence_def)

lemma sequence_edges [simp]: "Edges (sequence G1 G2) = Edges G1 \<union> Edges G2"
by (simp add: sequence_def)

lemma sequence_exit [simp]: "Exit (sequence G1 G2) = Exit G2"
by (simp add: sequence_def)

fun body_toCFG::"('class, 'mname, 'field) body \<Rightarrow> 'node set \<Rightarrow> 'node
    \<Rightarrow>
  ('node, 'class edge_type, ('class, 'mname, 'field) c_instr) graph_part" where
"body_toCFG (Load i) N n = (let n' = new (insert n N) 1 ! 0 in only n n' (ldc_i4 i))" |
"body_toCFG (Seq a b) N n = (let G1 = body_toCFG a N n in let G2 = body_toCFG b (N \<union>
     Nodes G1) (Exit G1) in
  sequence G1 G2)" |
"body_toCFG (Cond a b c) N n = (let G1 = body_toCFG a N n in let n0 = new (N \<union> Nodes
     G1) 1 ! 0 in
  let G2 = body_toCFG b (N \<union> Nodes G1) n0 in let n1 = new (N \<union> Nodes G1 \<
      union> Nodes G2) 1 ! 0 in
  let G3 = body_toCFG c (N \<union> Nodes G1 \<union> Nodes G2) n1 in
  \<lparr>Nodes = Nodes G1 \<union> Nodes G2 \<union> Nodes G3,
   Edges = {(Exit G1, n0, seq), (Exit G1, n1, branch), (Exit G2, Exit G3, seq)} \<union>
   Edges G1 \<union> Edges G2 \<union> Edges G3, Start = n, Exit = Exit G3,
```

```
  pLabel = (pLabel G1 ++ pLabel G2 ++ pLabel G3)(Exit G1 \<mapsto> brtrue, Exit G2 \<
      mapsto> br)\<rparr>)" |
"body_toCFG (While a b) N n = (let n0 = new (insert n N) 1 ! 0 in let G1 = body_toCFG a (
    insert n N) n0 in
  let n1 = new (insert n N \<union> Nodes G1) 1 ! 0 in let G2 = body_toCFG b (insert n N \<
      union> Nodes G1) n1 in
  let n2 = new (insert n N \<union> Nodes G1 \<union> Nodes G2) 1 ! 0 in
  \<lparr>Nodes = {n, n2} \<union> Nodes G1 \<union> Nodes G2,
   Edges = {(n, n0, seq), (Exit G1, n1, seq), (Exit G1, n2, branch), (Exit G2, n0, seq)} \<
      union>
   Edges G1 \<union> Edges G2, Start = n, Exit = n2,
   pLabel = (pLabel G1 ++ pLabel G2)(n \<mapsto> br (* skip *), Exit G1 \<mapsto> brfalse,
      Exit G2 \<mapsto> br)\<rparr>)" |
"body_toCFG (Loadind a) N n = (let G1 = body_toCFG a N n in let n' = new (N \<union> Nodes
    G1) 1 ! 0 in
  follow G1 n' ldind)" |
"body_toCFG (Storeind a b) N n = (let G1 = body_toCFG a N n in
  let G2 = body_toCFG b (N \<union> Nodes G1) (Exit G1) in
  let n' = new (N \<union> Nodes G1 \<union> Nodes G2) 1 ! 0 in follow (sequence G1 G2) n'
      stind)" |
"body_toCFG (Loadarga j) N n = (let n' = new (insert n N) 1 ! 0 in only n n' (ldarga (int j
    )))" |
"body_toCFG (Storearg a j) N n = (let G1 = body_toCFG a N n in let n' = new (N \<union>
    Nodes G1) 1 ! 0 in
  follow G1 n' (starg (int j)))" |
"body_toCFG (Newobj args K) N n = (case args of [] \<Rightarrow> let n' = new (insert n N)
    1 ! 0 in only n n' (newobj K)
  | a # rest \<Rightarrow> let G1 = body_toCFG a N n in let G = fold (\<lambda>b G. let G'
      = body_toCFG b (N \<union> Nodes G) (Exit G)
      in sequence G G') rest G1 in let n' = new (N \<union> Nodes G) 1 ! 0 in follow G n' (
          newobj K))" |
"body_toCFG (Callboxed a args M) N n = (let G1 = body_toCFG a N n in let G = fold (\<lambda
    >b G. let G' =
    body_toCFG b (N \<union> Nodes G) (Exit G) in sequence G G') args G1 in
    let n' = new (N \<union> Nodes G) 1 ! 0 in follow G n' (callvirt M))" |
"body_toCFG (Callunboxed a args M) N n = (let G1 = body_toCFG a N n in let G = fold (\<
    lambda>b G. let G' =
    body_toCFG b (N \<union> Nodes G) (Exit G) in sequence G G') args G1 in
    let n' = new (N \<union> Nodes G) 1 ! 0 in follow G n' (call instance M))" |
```

```
"body_toCFG (Loadflda a t c f) N n = (let G1 = body_toCFG a N n in let n' = new (N \<union>
     Nodes G1) 1 ! 0 in
  follow G1 n' (ldflda t c::f))" |
"body_toCFG (Storefld a b t c f) N n = (let G1 = body_toCFG a N n in let G2 = body_toCFG b
    (N \<union> Nodes G1) (Exit G1) in
  let n' = new (N \<union> Nodes G1 \<union> Nodes G2) 1 ! 0 in follow (sequence G1 G2) n'
      (stfld t c::f))" |
"body_toCFG (Box a vc) N n = (let G1 = body_toCFG a N n in let n' = new (N \<union> Nodes
    G1) 1 ! 0 in
  follow G1 n' (box vc))" |
"body_toCFG (Unbox a vc) N n = (let G1 = body_toCFG a N n in let n' = new (N \<union> Nodes
     G1) 1 ! 0 in
  follow G1 n' (unbox vc))"

fun body_toCFG'::"('class, 'mname, 'field) body \<Rightarrow> 'node set \<Rightarrow> 'node
     \<Rightarrow>
  ('node, 'class edge_type, ('class, 'mname, 'field) c_instr) graph_part" where
"body_toCFG' (Load i) N n = (let n' = new (insert n N) 1 ! 0 in only n n' (ldc_i4 i))" |
"body_toCFG' (Seq a b) N n = (let G1 = body_toCFG' a N n in let G2 = body_toCFG' b (N \<
    union> Nodes G1) (Exit G1) in
  sequence G1 G2)" |
"body_toCFG' (Cond a b c) N n = (let G1 = body_toCFG' a N n in let n0 = new (N \<union>
    Nodes G1) 1 ! 0 in
  let G2 = body_toCFG' b (N \<union> Nodes G1) n0 in let n1 = new (N \<union> Nodes G1 \<
      union> Nodes G2) 1 ! 0 in
  let G3 = body_toCFG' c (N \<union> Nodes G1 \<union> Nodes G2) n1 in
  \<lparr>Nodes = Nodes G1 \<union> Nodes G2 \<union> Nodes G3,
   Edges = {(Exit G1, n0, seq), (Exit G1, n1, branch), (Exit G2, Exit G3, seq)} \<union>
   Edges G1 \<union> Edges G2 \<union> Edges G3, Start = n, Exit = Exit G3,
   pLabel = (pLabel G1 ++ pLabel G2 ++ pLabel G3)(Exit G1 \<mapsto> brtrue, Exit G2 \<
       mapsto> br)\<rparr>)" |
"body_toCFG' (While a b) N n = (let n0 = new (insert n N) 1 ! 0 in let G1 = body_toCFG' a (
    insert n N) n0 in
  let n1 = new (insert n N \<union> Nodes G1) 1 ! 0 in let G2 = body_toCFG' b (insert n N
      \<union> Nodes G1) n1 in
  let n2 = new (insert n N \<union> Nodes G1 \<union> Nodes G2) 1 ! 0 in
  \<lparr>Nodes = {n, n2} \<union> Nodes G1 \<union> Nodes G2,
   Edges = {(n, n0, seq), (Exit G1, n1, seq), (Exit G1, n2, branch), (Exit G2, n0, seq)} \<
       union>
   Edges G1 \<union> Edges G2, Start = n, Exit = n2,
```

```
   pLabel = (pLabel G1 ++ pLabel G2)(n \<mapsto> br (* skip *), Exit G1 \<mapsto> brfalse,
       Exit G2 \<mapsto> br)\<rparr>)" |
"body_toCFG' (Loadind a) N n = (let G1 = body_toCFG' a N n in let n' = new (N \<union>
    Nodes G1) 1 ! 0 in
  follow G1 n' ldind)" |
"body_toCFG' (Storeind a b) N n = (let G1 = body_toCFG' a N n in
  let G2 = body_toCFG' b (N \<union> Nodes G1) (Exit G1) in
  let n' = new (N \<union> Nodes G1 \<union> Nodes G2) 1 ! 0 in follow (sequence G1 G2) n'
      stind)" |
"body_toCFG' (Loadarga j) N n = (let n' = new (insert n N) 1 ! 0 in only n n' (ldarga (int
    j)))" |
"body_toCFG' (Storearg a j) N n = (let G1 = body_toCFG' a N n in let n' = new (N \<union>
    Nodes G1) 1 ! 0 in
  follow G1 n' (starg (int j)))" |
"body_toCFG' (Newobj args K) N n = (case args of [] \<Rightarrow> let n' = new (insert n N)
     1 ! 0 in only n n' (newobj K)
  | a # rest \<Rightarrow> let G1 = body_toCFG' a N n in sequence G1 (body_toCFG' (Newobj
      rest K) (N \<union> Nodes G1) (Exit G1)))" |
"body_toCFG' (Callboxed a args M) N n = (let G1 = body_toCFG' a N n in case args of
    [] \<Rightarrow> let n' = new (N \<union> Nodes G1) 1 ! 0 in follow G1 n' (callvirt M)
  | a # rest \<Rightarrow> sequence G1 (body_toCFG' (Callboxed a rest M) (N \<union> Nodes
      G1) (Exit G1)))" |
"body_toCFG' (Callunboxed a args M) N n = (let G1 = body_toCFG' a N n in case args of
    [] \<Rightarrow> let n' = new (N \<union> Nodes G1) 1 ! 0 in follow G1 n' (call
        instance M)
  | a # rest \<Rightarrow> sequence G1 (body_toCFG' (Callunboxed a rest M) (N \<union>
      Nodes G1) (Exit G1)))" |
"body_toCFG' (Loadflda a t c f) N n = (let G1 = body_toCFG' a N n in let n' = new (N \<
    union> Nodes G1) 1 ! 0 in
  follow G1 n' (ldflda t c::f))" |
"body_toCFG' (Storefld a b t c f) N n = (let G1 = body_toCFG' a N n in let G2 = body_toCFG'
     b (N \<union> Nodes G1) (Exit G1) in
  let n' = new (N \<union> Nodes G1 \<union> Nodes G2) 1 ! 0 in follow (sequence G1 G2) n'
      (stfld t c::f))" |
"body_toCFG' (Box a vc) N n = (let G1 = body_toCFG' a N n in let n' = new (N \<union> Nodes
     G1) 1 ! 0 in
  follow G1 n' (box vc))" |
"body_toCFG' (Unbox a vc) N n = (let G1 = body_toCFG' a N n in let n' = new (N \<union>
    Nodes G1) 1 ! 0 in
  follow G1 n' (unbox vc))"
```

```
thm body_toCFG.induct
thm body_toCFG'.induct

lemma sequence_unfold [simp]: "fold (\<lambda>b G. sequence G (body_toCFG b (N \<union>
    Nodes G) (Exit G))) list (sequence G1 G2) =
  sequence G1 (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G1 \<union>
      Nodes G) (Exit G))) list G2)"
by (induct list arbitrary: G1 G2 N, auto simp add: Un_assoc)

declare sequence_unfold [simp del]
lemma toCFG_fold_aux: "(\<And>a b N n. P a N n \<Longrightarrow> P b (N \<union> Nodes a) (
    Exit a) \<Longrightarrow> P (sequence a b) N n) \<Longrightarrow>
  (\<And>a N n i. P a N n \<Longrightarrow> P (follow a (SOME n. n \<notin> N \<and> n \<
      notin> Nodes a) i) N n) \<Longrightarrow>
  P (G::('a, 'b edge_type, ('b, 'c, 'd) c_instr) graph_part) N n \<Longrightarrow>
  (\<And>x xa (xb::('a, 'b edge_type, ('b, 'c, 'd) c_instr) graph_part). x = body_toCFG a N
       n \<Longrightarrow> xa \<in> set args \<Longrightarrow>
    P (body_toCFG xa (N \<union> Nodes xb) (Exit xb)) (N \<union> Nodes xb) (Exit xb)) \<
        Longrightarrow>
  P (follow (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)))
      args G)
    (SOME na. na \<notin> N \<and> na \<notin> Nodes (fold (\<lambda>b G. sequence G (
        body_toCFG b (N \<union> Nodes G) (Exit G))) args G)) i) N n"
apply (subgoal_tac "P (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (
    Exit G))) args G) N n", simp)
apply (induct args arbitrary: N n G, auto)
done

(* An induction principle that ignores the instructions. *)
lemma toCFG_induct: "(\<And>N n i. P (let n' = new (insert n N) 1 ! 0 in only n n' i) N n)
    \<Longrightarrow>
(\<And>a b N n. P a N n \<Longrightarrow> P b (N \<union> Nodes a) (Exit a) \<
    Longrightarrow> P (sequence a b) N n) \<Longrightarrow>
(\<And>a b c N n. P a N n \<Longrightarrow> P b (N \<union> Nodes a) (new (N \<union> Nodes
     a) 1 ! 0) \<Longrightarrow>
  P c (N \<union> Nodes a \<union> Nodes b) (new (N \<union> Nodes a \<union> Nodes b) 1 !
      0) \<Longrightarrow>
  P (\<lparr>Nodes = Nodes a \<union> Nodes b \<union> Nodes c, Edges = {(Exit a, new (N \<
      union> Nodes a) 1 ! 0, seq),
```

```
  (Exit a, new (N \<union> Nodes a \<union> Nodes b) 1 ! 0, branch), (Exit b, Exit c, seq)
     } \<union>
  Edges a \<union> Edges b \<union> Edges c, Start = n, Exit = Exit c,
  pLabel = (pLabel a ++ pLabel b ++ pLabel c)(Exit a \<mapsto> brtrue, Exit b \<mapsto> br
     )\<rparr>) N n) \<Longrightarrow>
(\<And>a b N n. P a (insert n N) (new (insert n N) 1 ! 0) \<Longrightarrow>
 P b (insert n N \<union> Nodes a) (new (insert n N \<union> Nodes a) 1 ! 0) \<
    Longrightarrow>
 P (let n2 = new (insert n N \<union> Nodes a \<union> Nodes b) 1 ! 0 in
 \<lparr>Nodes = {n, n2} \<union> Nodes a \<union> Nodes b, Edges = {(n, new (insert n N)
    1 ! 0, seq),
 (Exit a, new (insert n N \<union> Nodes a) 1 ! 0, seq), (Exit a, n2, branch), (Exit b,
    new (insert n N) 1 ! 0, seq)} \<union>
  Edges a \<union> Edges b, Start = n, Exit = n2,
  pLabel = (pLabel a ++ pLabel b)(n \<mapsto> br (* skip *), Exit a \<mapsto> brfalse,
     Exit b \<mapsto> br)\<rparr>) N n) \<Longrightarrow>
(\<And>a N n i. P a N n \<Longrightarrow> P (let n' = new (N \<union> Nodes a) 1 ! 0 in
   follow a n' i) N n) \<Longrightarrow>
(\<And>a b N n i. P a N n \<Longrightarrow> P b (N \<union> Nodes a) (Exit a) \<
   Longrightarrow>
 P (let n' = new (N \<union> Nodes a \<union> Nodes b) 1 ! 0 in follow (sequence a b) n' i
    ) N n) \<Longrightarrow>
 P (body_toCFG a N n) N n"
apply (cut_tac P="\<lambda>a N n. P (body_toCFG a N n) N n" in body_toCFG.induct, auto simp
    add: Let_def)
apply (case_tac args, auto simp add: Let_def)
apply (rule_tac P=P in toCFG_fold_aux, simp+)
apply (metis (hide_lams, no_types))
apply (rule_tac P=P in toCFG_fold_aux, simp+)+
done

lemma empty_inter: "\<lbrakk>a \<inter> z = {}; b \<inter> z = {}\<rbrakk> \<Longrightarrow
   > (a \<union> b) \<inter> z = {}"
by (metis Int_Un_distrib2 Un_empty_left)

lemma new_nodes_set: "\<lbrakk>\<not>finite (UNIV::'a set); finite (S::'a set)\<rbrakk> \<
   Longrightarrow> set (new S n) \<inter> S = {}"
by (metis disjoint_iff_in_not_in1 new_nodes_are_new)
```

```
lemma new_nodes_subset: "\<lbrakk>\<not>finite (UNIV::'a set); finite (S::'a set); T \<
    subseteq> S\<rbrakk> \<Longrightarrow>
  set (new S n) \<inter> T = {}"
by (metis (hide_lams, no_types) inf_absorb2 inf_assoc inf_bot_left new_nodes_set)

lemma start_fold [simp]: "Start (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union>
    Nodes G) (Exit G))) list
  G) = Start G"
by (induct list arbitrary: G, auto simp add: sequence_def)

lemma fold_mono [intro]: "n \<in> Nodes G \<Longrightarrow>
  n \<in> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)
      )) list G)"
by (induct list arbitrary: G, auto simp add: sequence_def)

lemma start_n [simp]: "Start (body_toCFG b N n) = n"
apply (cut_tac P="\<lambda>b N n. Start b = n" in toCFG_induct,
  simp_all add: Let_def only_def follow_def split: list.splits)
apply (simp add: sequence_def)+
done

lemma body_props: "\<lbrakk>infinite (UNIV::'node set); finite N\<rbrakk> \<Longrightarrow>
     finite (Nodes (body_toCFG b N n)) \<and>
  (n' \<in> N \<and> n \<noteq> n' \<longrightarrow> n' \<notin> Nodes (body_toCFG (b::('
      class, 'mname, 'field) body) (N::'node set) n)) \<and>
  Exit (body_toCFG b N n) \<in> Nodes (body_toCFG b N n) \<and> n \<in> Nodes (body_toCFG b
       N n) \<and>
  Exit (body_toCFG b N n) \<noteq> n \<and>
  dom (pLabel (body_toCFG b N n)) = Nodes (body_toCFG b N n) - {Exit (body_toCFG b N n)} \<
      and>
  (u, u, seq) \<notin> Edges (body_toCFG b N n)"
apply (cut_tac P="\<lambda>(b::('node, 'class, 'mname, 'field) BIL_part) (N::'node set) n.
    \<forall>n'. finite N \<longrightarrow>
  finite (Nodes b) \<and> (n' \<in> N \<and> n \<noteq> n' \<longrightarrow> n' \<notin>
      Nodes b) \<and> Exit b \<in> Nodes b \<and> n \<in> Nodes b \<and>
  Exit b \<noteq> n \<and> dom (pLabel b) = Nodes b - {Exit b} \<and> (u, u, seq) \<notin>
      Edges b" and a=b and N=N and n=n
  in toCFG_induct, simp_all add: Let_def only_def follow_def sequence_def)
apply (clarsimp, drule_tac A="insert n Na" in fresh_new, clarsimp+)
apply (rule conjI, clarsimp, blast)
```

```
apply clarsimp
apply (rule conjI, clarsimp)
apply (case_tac "n' = Exit a", simp_all)
apply force
apply clarsimp
apply (cut_tac A="Na \<union> Nodes a" in fresh_new, clarsimp+)
apply (cut_tac A="Na \<union> Nodes a \<union> Nodes b" in fresh_new, clarsimp+)
apply force
apply clarsimp
apply (frule_tac A="insert n Na" in fresh_new, simp+)
apply (cut_tac A="insert n Na \<union> Nodes a" in fresh_new, clarsimp+)
apply (cut_tac A="insert n Na \<union> Nodes a \<union> Nodes b" in fresh_new, clarsimp+)
apply force
apply clarsimp
apply (frule_tac A="Na \<union> Nodes a" in fresh_new, clarsimp+)
apply force
apply clarsimp
apply (frule_tac A="Na \<union> Nodes a" in fresh_new, clarsimp+)
apply (frule_tac A=Na in fresh_new, simp+)
apply (cut_tac A="Na \<union> Nodes a \<union> Nodes b" in fresh_new, clarsimp+)
apply force
done

corollary toCFG_no_loop: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set)\<
    rbrakk> \<Longrightarrow>
  (u, u, seq) \<notin> Edges (body_toCFG b N n)"
by (simp add: body_props)

lemma n_in [intro!, simp]: "n \<in> Nodes (body_toCFG b N n)"
by (cut_tac P="\<lambda>b N n. n \<in> Nodes b" in toCFG_induct,
  simp_all add: Let_def only_def follow_def split: list.splits)

corollary n_in_fold [intro!, simp]: "n \<in> Nodes (
  fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G))) list (
      body_toCFG b N n))"
by (rule fold_mono, simp)

lemma exit_in [intro!]: "Exit (body_toCFG b N n) \<in> Nodes (body_toCFG b N n)"
by (cut_tac P="\<lambda>b N n. Exit b \<in> Nodes b" in toCFG_induct,
  simp_all add: Let_def only_def follow_def sequence_def split: list.splits)
```

```
corollary fold_exit_in [intro!]: "Exit G \<in> Nodes G \<Longrightarrow>
  Exit (fold (\<lambda>b G. sequence G (body_toCFG b (Na \<union> Nodes G) (Exit G))) list
      G) \<in>
  Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (Na \<union> Nodes G) (Exit G))) list
       G)"
apply (induct list arbitrary: Na G, auto)
by (smt UnCI doubly_pointed_graph.select_convs(1) doubly_pointed_graph.select_convs(4)
    exit_in
  only_def follow_def sequence_def)

(* This might provide an easier induction. *)
lemma body_conv: "body_toCFG' b N n = body_toCFG b N n"
apply (cut_tac P="\<lambda>b N n. body_toCFG' b N n = body_toCFG b N n" in body_toCFG'.
    induct,
  auto simp add: sequence_unfold)
apply (case_tac args, auto simp add: Let_def sequence_unfold)
apply (case_tac list, auto simp add: Let_def sequence_unfold)
apply (subgoal_tac "(SOME na. na \<noteq> Exit (body_toCFG a N n) \<and> na \<notin> N \<
    and> na \<notin> Nodes (body_toCFG a N n)) =
  (SOME na. na \<notin> N \<and> na \<notin> Nodes (body_toCFG a N n))", simp)
apply (rule sym, rule follow_simp, clarsimp)
apply (rule_tac f=Eps in arg_cong, auto)
apply (case_tac args, auto simp add: Let_def sequence_unfold)
apply (case_tac args, auto simp add: Let_def sequence_unfold)
done

lemma body_nodes_finite [simp]: "finite (Nodes (body_toCFG b N n))"
by (cut_tac P="\<lambda>b N n. \<forall>N n. finite (Nodes b)" in toCFG_induct,
  auto simp add: Let_def only_def follow_def split: list.splits)

corollary fold_nodes_finite [simp]: "finite (Nodes G) \<Longrightarrow>
  finite (Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)
      )) list G))"
by (induct list arbitrary: G, auto simp add: sequence_def)

lemma new_CFG_distinct: "\<lbrakk>infinite (UNIV::'node set); finite N; n \<in> N; n \<
    noteq> n'\<rbrakk> \<Longrightarrow>
  n \<notin> Nodes (body_toCFG (b::('class, 'mname, 'field) body) (N::'node set) n')"
by (simp add: body_props)
```

```
lemma exit_distinct: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set)\<rbrakk>
    \<Longrightarrow>
  Exit (body_toCFG (b::('a, 'b, 'c) body) N n) \<noteq> n"
by (simp add: body_props)

corollary exit_out [intro!]: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set);
    finite N'\<rbrakk> \<Longrightarrow>
  Exit (body_toCFG b (N' \<union> Nodes (body_toCFG a N n)) n') \<notin> Nodes (body_toCFG
      a N n)"
apply (clarsimp, cut_tac b=b and n="Exit (body_toCFG b (N' \<union> Nodes (body_toCFG a N n
    )) n')" and
  N="N' \<union> Nodes (body_toCFG a N n)" and n'=n' in new_CFG_distinct, simp+)
apply (rule exit_distinct, simp+, simp add: exit_in)
done

corollary exit_out2 [intro!]: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set)\<
    rbrakk> \<Longrightarrow>
  Exit (body_toCFG b N n) \<notin> N"
apply (clarsimp, cut_tac b=b and n="Exit (body_toCFG b N n)" and N=N and n'=n in
    new_CFG_distinct,
  simp+)
apply (rule exit_distinct, simp+, simp add: exit_in)
done

declare is_doubly_pointed_graph_def [simp del]

lemma fold_graph: "\<lbrakk>is_doubly_pointed_graph (G::('a, 'b edge_type, ('b, 'c, 'd)
    c_instr) graph_part);
  infinite (UNIV::'a set); finite Na; (\<forall>aa lista x xa (xb::('a, 'b edge_type, ('b,
      'c, 'd) c_instr) graph_part).
  a = aa \<and> list = lista \<longrightarrow> x = body_toCFG aa Na n \<longrightarrow> xa
      \<in> set lista \<longrightarrow> finite (Nodes xb) \<longrightarrow>
  is_doubly_pointed_graph (body_toCFG xa (Na \<union> Nodes xb) (Exit xb)))\<rbrakk> \<
      Longrightarrow>
  is_doubly_pointed_graph (fold (\<lambda>b G. sequence G (body_toCFG b (Na \<union> Nodes
      G) (Exit G))) list G)"
apply (induct list arbitrary: G, simp_all)
apply (subgoal_tac "is_doubly_pointed_graph (sequence G (body_toCFG a (Na \<union> Nodes G)
     (Exit G)))", simp)
```

```
apply (rule sequence_graph, simp+)
apply (erule_tac x=a in allE, clarsimp, erule_tac x=G in allE)
apply (simp add: is_doubly_pointed_graph_def, drule pointed_graph.finite_nodes, simp)
apply (simp add: is_doubly_pointed_graph_def, frule pointed_graph.has_start)
apply (cut_tac b=a and N="Na \<union> Nodes G" and n="Exit G" in exit_in)
apply (cut_tac b=a and N="Na \<union> Nodes G" and n'="Exit G" and n="Start G" in
    new_CFG_distinct, simp_all)
apply (erule pointed_graph.finite_nodes)
apply (erule pointed_graph.start_not_exit)
apply metis
apply (simp add: is_doubly_pointed_graph_def, frule pointed_graph.has_start)
apply (cut_tac b=a and N="Na \<union> Nodes G" and n'="Exit G" and n="Start G" in
    new_CFG_distinct, simp_all)
apply (erule pointed_graph.finite_nodes)
apply (erule pointed_graph.start_not_exit)
apply clarsimp
apply (simp add: is_doubly_pointed_graph_def, frule pointed_graph.has_start)
apply (cut_tac b=a and N="Na \<union> Nodes G" and n'="Exit G" and n="Exit (body_toCFG a (
    Na \<union> Nodes G)
  (Exit G))" in new_CFG_distinct, simp_all)
apply (erule pointed_graph.finite_nodes)
apply (metis (hide_lams, no_types) Un_infinite_iff exit_distinct pointed_graph.finite_nodes
    )
by (metis exit_in)

lemma toCFG_graph: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set); finite (
    UNIV::'class set)\<rbrakk> \<Longrightarrow>
  is_doubly_pointed_graph (body_toCFG (b::('class, 'b, 'c) body) N n)"
apply (cut_tac P="\<lambda>(b::('class, 'b, 'c) body) (N::'node set) n. finite N \<
    longrightarrow>
  is_doubly_pointed_graph (body_toCFG b N n)" in body_toCFG.induct, simp_all add: Let_def)
apply clarsimp
apply (cut_tac A="insert n Na" in fresh_new, clarsimp+)
apply (rule sequence_graph, simp_all)
apply (metis n_in exit_out)
apply (rule new_CFG_distinct, simp+)
apply (metis exit_distinct)
apply (metis exit_out)
apply (clarsimp simp add: is_doubly_pointed_graph_def)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n)" in fresh_new, simp+)
```

```
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n) \<union> Nodes (body_toCFG b (Na \<
    union> Nodes (body_toCFG a Na n))
  (SOME na. na \<notin> Na \<and> na \<notin> Nodes (body_toCFG a Na n)))" in fresh_new,
      simp+)
apply (subgoal_tac "is_doubly_pointed_graph (body_toCFG b (Na \<union> Nodes (body_toCFG a
    Na n))
  (SOME na. na \<notin> Na \<and> na \<notin> Nodes (body_toCFG a Na n)))")
apply (subgoal_tac "is_doubly_pointed_graph (body_toCFG c (Na \<union> Nodes (body_toCFG a
    Na n) \<union>
  Nodes (body_toCFG b (Na \<union> Nodes (body_toCFG a Na n)) (SOME na. na \<notin> Na \<
      and> na \<notin> Nodes (body_toCFG a Na n))))
  (SOME na. na \<notin> Na \<and> na \<notin> Nodes (body_toCFG a Na n) \<and> na \<notin>
      Nodes (body_toCFG b
  (Na \<union> Nodes (body_toCFG a Na n)) (SOME na. na \<notin> Na \<and> na \<notin> Nodes
       (body_toCFG a Na n)))))")
apply (unfold_locales, simp_all add: in_edges_def out_edges_def Let_def)
apply (erule disjE, clarsimp)+
apply (metis exit_in)
apply (erule disjE, drule pointed_graph.edges_ok, simp+)
apply (erule disjE, simp only: is_doubly_pointed_graph_def)
apply (drule_tac Nodes="Nodes ((body_toCFG b (Na \<union> Nodes (body_toCFG a Na n))
  (SOME na. na \<notin> Na \<and> na \<notin> Nodes (body_toCFG a Na n))))" in
      pointed_graph.edges_ok, simp+)
apply (simp only: is_doubly_pointed_graph_def)
apply (drule_tac Nodes="Nodes (body_toCFG c (Na \<union> Nodes (body_toCFG a Na n) \<union>
  Nodes (body_toCFG b (Na \<union> Nodes (body_toCFG a Na n)) (SOME na. na \<notin> Na \<
      and> na \<notin> Nodes (body_toCFG a Na n))))
  (SOME na. na \<notin> Na \<and> na \<notin> Nodes (body_toCFG a Na n) \<and> na \<notin>
      Nodes (body_toCFG b
  (Na \<union> Nodes (body_toCFG a Na n)) (SOME na. na \<notin> Na \<and> na \<notin> Nodes
       (body_toCFG a Na n)))))"
  in pointed_graph.edges_ok, simp+)
apply (metis exit_in)
apply (smt Un_assoc Un_commute body_nodes_finite exit_out finite_UnI n_in)
apply clarsimp
apply (rule conjI, metis n_in)
apply (rule conjI, metis n_in)
apply (rule conjI)
apply (metis (lifting) Un_iff body_nodes_finite exit_in finite_Un n_in new_CFG_distinct)
apply (drule pointed_graph.start_first, simp add: in_edges_def)
```

```
apply (smt Un_iff Un_infinite_iff body_nodes_finite n_in new_CFG_distinct pointed_graph.
    edges_ok)
apply (clarsimp, rule conjI)
apply (metis (lifting) Un_iff body_nodes_finite exit_in finite_Un new_CFG_distinct)
apply (rule conjI)
apply (metis (lifting) Un_iff body_nodes_finite exit_in finite_Un new_CFG_distinct)
apply (rule conjI)
apply (metis (lifting) Un_iff body_nodes_finite exit_in finite_Un new_CFG_distinct)
apply (rule conjI)
apply (smt body_nodes_finite exit_out finite_Un inf_sup_aci(5) inf_sup_aci(6) pointed_graph
    .edges_ok)
apply (rule conjI)
apply (smt Un_infinite_iff body_nodes_finite exit_out pointed_graph.edges_ok)
apply (simp only: is_doubly_pointed_graph_def)
apply ((drule pointed_graph.exit_last)+, simp add: out_edges_def)
apply (simp add: is_doubly_pointed_graph_def)
apply (simp add: is_doubly_pointed_graph_def)
(**)
apply (clarsimp simp add: is_doubly_pointed_graph_def)
apply (cut_tac A="insert n Na" in fresh_new, simp+)
apply (cut_tac A="insert n Na \<union> Nodes (body_toCFG a (insert n Na) (SOME na. na \<
    noteq> n \<and> na \<notin> Na))" in fresh_new, simp+)
apply (cut_tac A="insert n Na \<union> Nodes (body_toCFG a (insert n Na) (SOME na. na \<
    noteq> n \<and> na \<notin> Na)) \<union> Nodes (body_toCFG b
  (insert n (Na \<union> Nodes (body_toCFG a (insert n Na) (SOME na. na \<noteq> n \<and>
      na \<notin> Na)))) (SOME na. na \<noteq> n \<and> na \<notin> Na \<and>
  na \<notin> Nodes (body_toCFG a (insert n Na) (SOME na. na \<noteq> n \<and> na \<notin>
      Na))))" in fresh_new, simp+)
apply (subgoal_tac "is_doubly_pointed_graph (body_toCFG a (insert n Na) (SOME na. na \<
    noteq> n \<and> na \<notin> Na))")
apply (subgoal_tac "is_doubly_pointed_graph (body_toCFG b (insert n (Na \<union> Nodes (
    body_toCFG a (insert n Na)
  (SOME na. na \<noteq> n \<and> na \<notin> Na)))) (SOME na. na \<noteq> n \<and> na \<
      notin> Na \<and> na \<notin> Nodes (body_toCFG a (insert n Na)
  (SOME na. na \<noteq> n \<and> na \<notin> Na))))")
apply (unfold_locales, simp_all add: in_edges_def out_edges_def Let_def)
apply (erule disjE, clarsimp)+
apply (smt pointed_graph.edges_ok)
apply (simp only: is_doubly_pointed_graph_def)
```

```
apply (drule_tac Nodes="Nodes (body_toCFG b (insert n (Na \<union> Nodes (body_toCFG a (
    insert n Na)
  (SOME na. na \<noteq> n \<and> na \<notin> Na)))) (SOME na. na \<noteq> n \<and> na \<
      notin> Na \<and> na \<notin> Nodes (body_toCFG a (insert n Na)
  (SOME na. na \<noteq> n \<and> na \<notin> Na))))" in pointed_graph.edges_ok, simp+)
apply metis
apply clarsimp
apply (rule conjI)
apply (metis (lifting) finite_insert insertI1 new_CFG_distinct pointed_graph.edges_ok)
apply (clarsimp simp only: is_doubly_pointed_graph_def)
apply (drule_tac Nodes="Nodes (body_toCFG b (insert n (Na \<union> Nodes (body_toCFG a (
    insert n Na)
  (SOME na. na \<noteq> n \<and> na \<notin> Na)))) (SOME na. na \<noteq> n \<and> na \<
      notin> Na \<and> na \<notin> Nodes (body_toCFG a
  (insert n Na) (SOME na. na \<noteq> n \<and> na \<notin> Na))))" in pointed_graph.
      edges_ok, simp+)
apply (metis (lifting, mono_tags) body_nodes_finite finite.insertI finite_UnI insertI1
    new_CFG_distinct)
apply (clarsimp, rule conjI)
apply (metis exit_in)
apply (rule conjI)
apply (metis (lifting) exit_in)
apply (rule conjI)
apply (metis (lifting) exit_in)
apply (rule conjI)
apply (smt body_nodes_finite exit_out finite_Un inf_sup_aci(5) inf_sup_aci(6) pointed_graph
    .edges_ok)
apply (smt Un_infinite_iff body_nodes_finite exit_out pointed_graph.edges_ok)
apply (simp add: is_doubly_pointed_graph_def)
apply (simp add: is_doubly_pointed_graph_def)
(**)
apply clarsimp
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n)" in fresh_new, simp+)
apply (clarsimp, rule follow_graph, rule sequence_graph, simp_all)
apply (metis exit_out n_in)
apply (metis UnCI body_nodes_finite exit_distinct finite_Un n_in new_CFG_distinct)
apply (metis exit_out)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n) \<union> Nodes (body_toCFG b (Na \<
    union>
  Nodes (body_toCFG a Na n)) (Exit (body_toCFG a Na n)))" in fresh_new, simp+)
```

```
apply clarsimp
apply (cut_tac A="insert n Na" in fresh_new, clarsimp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n)" in fresh_new, simp+)
apply (case_tac args, clarsimp)
apply (cut_tac A="insert n Na" in fresh_new, simp+)
apply (clarsimp simp add: Let_def, rule follow_graph)
apply (rule fold_graph, simp+)
apply metis
apply (cut_tac A="Na \<union> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (Na \<
    union> Nodes G) (Exit G))) list
  (body_toCFG a Na n))" in fresh_new, simp+)
apply (clarsimp simp add: Let_def, rule follow_graph)
apply (rule fold_graph, simp+)
apply (cut_tac A="Na \<union> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (Na \<
    union> Nodes G) (Exit G))) args
  (body_toCFG a Na n))" in fresh_new, simp+)
apply (clarsimp simp add: Let_def, rule follow_graph)
apply (rule fold_graph, simp+)
apply (cut_tac A="Na \<union> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (Na \<
    union> Nodes G) (Exit G))) args
  (body_toCFG a Na n))" in fresh_new, simp+)
apply clarsimp
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n)" in fresh_new, simp+)
apply (clarsimp, rule follow_graph, rule sequence_graph, simp_all)
apply (metis exit_out n_in)
apply (metis UnCI body_nodes_finite exit_distinct finite_Un n_in new_CFG_distinct)
apply (metis exit_out)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n) \<union> Nodes (body_toCFG b (Na \<
    union>
  Nodes (body_toCFG a Na n)) (Exit (body_toCFG a Na n)))" in fresh_new, simp+)
apply clarsimp
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n)" in fresh_new, simp+)
apply clarsimp
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na n)" in fresh_new, simp+)
apply force
done
(* Should be simpler with a better induction principle. *)

corollary toCFG_edges_in: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set);
```

```
  finite (UNIV::'class set); (u, v, t) \<in> Edges (body_toCFG (b::('class, 'b, 'c) body) N
       n)\<rbrakk> \<Longrightarrow>
  u \<in> Nodes (body_toCFG b N n) \<and> v \<in> Nodes (body_toCFG b N n) \<and> u \<noteq
      > Exit (body_toCFG b N n) \<and>
  v \<noteq> n"
apply (drule toCFG_graph, simp+, clarsimp simp add: is_doubly_pointed_graph_def
    pointed_graph_def
  in_edges_def out_edges_def)
apply (erule allE, erule allE, erule impE, force+)
done

corollary fold_edges_in': "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set);
  finite (UNIV::'class set); (u, v, t) \<in> Edges (fold (\<lambda>b G. sequence G (
      body_toCFG b (N \<union> Nodes G)
  (Exit G))) list G); finite (Nodes (G::('node, 'class, 'b, 'c) BIL_part)); n \<in> Nodes G
      ;
  (u, v, t) \<in> Edges G \<longrightarrow> (u \<in> Nodes G \<and> v \<in> Nodes G \<and>
      u \<noteq> Exit G \<and> v \<noteq> n)\<rbrakk> \<Longrightarrow>
  u \<in> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)
      )) list G) \<and>
  v \<in> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)
      )) list G) \<and>
  u \<noteq> Exit (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit
      G))) list G) \<and>
  v \<noteq> n"
apply (induct list arbitrary: G, simp_all)
apply (subgoal_tac "finite (Nodes (sequence G (body_toCFG a (N \<union> Nodes G) (Exit G)))
    ) \<and>
  n \<in> Nodes (sequence G (body_toCFG a (N \<union> Nodes G) (Exit G))) \<and>
  ((u, v, t) \<in> Edges (sequence G (body_toCFG a (N \<union> Nodes G) (Exit G))) \<
      longrightarrow>
  u \<in> Nodes (sequence G (body_toCFG a (N \<union> Nodes G) (Exit G))) \<and>
  v \<in> Nodes (sequence G (body_toCFG a (N \<union> Nodes G) (Exit G))) \<and>
  u \<noteq> Exit (sequence G (body_toCFG a (N \<union> Nodes G) (Exit G))) \<and> v \<
      noteq> n)")
apply smt
apply (clarsimp simp add: sequence_def)
apply (rule conjI, clarsimp)
apply (frule_tac b=a and N="N \<union> Nodes G" and n="Exit G" in exit_out2, simp+)
apply clarsimp
```

```
apply (frule_tac N="N \<union> Nodes G" and b=a in toCFG_edges_in, simp+)
by (smt Un_def finite_Un mem_Collect_eq new_CFG_distinct)

corollary fold_edges_in: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set);
  finite (UNIV::'class set); (u, v, t) \<in> Edges (fold (\<lambda>b G. sequence G (
      body_toCFG b (N \<union> Nodes G)
  (Exit G))) list (body_toCFG (a::('class, 'b, 'c) body) N n))\<rbrakk> \<Longrightarrow>
  u \<in> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)
      )) list (body_toCFG a N n)) \<and>
  v \<in> Nodes (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit G)
      )) list (body_toCFG a N n)) \<and>
  u \<noteq> Exit (fold (\<lambda>b G. sequence G (body_toCFG b (N \<union> Nodes G) (Exit
      G))) list (body_toCFG a N n)) \<and>
  v \<noteq> n"
apply (rule fold_edges_in', simp_all)
apply (clarsimp, rule toCFG_edges_in, simp+)
done

lemma toCFG_label_dom: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set)\<rbrakk>
     \<Longrightarrow>
  dom (pLabel (body_toCFG a N n)) = Nodes (body_toCFG (a::('a, 'b, 'c) body) N n) - {Exit (
      body_toCFG a N n)}"
by (simp add: body_props)

corollary dom_in: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set);
  pLabel (body_toCFG a N n) p = Some i\<rbrakk> \<Longrightarrow> p \<in> Nodes (body_toCFG
       (a::('a, 'b, 'c) body) N n) \<and>
  p \<noteq> Exit (body_toCFG a N n)"
apply (frule_tac a=a and N=N and n=n in toCFG_label_dom, simp+)
apply (drule domI, simp)
done

locale BIL_main = BIL where
  methods="methods::'class \<Rightarrow> ('class, 'mname) sig \<rightharpoonup> ('class, '
      mname, 'field) body" for methods +
  fixes main_class::'class and main_method::"('class, 'mname) sig"
  assumes main_exists: "methods main_class main_method \<noteq> None" begin

(* making an interprocedural CFG *)
```

```
definition "method_CFGs \<equiv> fst (fold (\<lambda>(a, b) (l, N). let n = new N 1 ! 0 in
    let G = body_toCFG b N n in
  ((a, G\<lparr>pLabel := (pLabel G)(Exit G \<mapsto> ret)\<rparr>) # l, N \<union> Nodes G
      ))
  (list_of_set {((c, s), b) | c s b. methods c s = Some b}) ([], {}))"

lemma fold_graphs_disjoint: "\<lbrakk>l = fst (fold (\<lambda>(a, b) (l, N). let n = new N
    1 ! 0 in
  let G = body_toCFG b (N::'node set) n in ((a, G\<lparr>pLabel := (pLabel G)(Exit G \<
      mapsto> ret)\<rparr>) # l,
  (N \<union> Nodes G))) a (b, c)); i < length l; j < length l; n \<in> Nodes (snd (l ! i))
      ;
  n \<in> Nodes (snd (l ! j)); finite c; \<forall>i j. i < length b \<and> j < length b \<
      and> n \<in> Nodes (snd (b ! i)) \<and>
  n \<in> Nodes (snd (b ! j)) \<longrightarrow> i = j; c = \<Union>(Nodes ` snd ` set b);
      infinite (UNIV::'node set)\<rbrakk> \<Longrightarrow>
  i = j"
apply (induct a arbitrary: b c l, auto)
apply (case_tac "let n = SOME n. \<forall>a\<in>set ba. n \<notin> Nodes (snd a); G =
    body_toCFG b (\<Union>a\<in>set ba. Nodes (snd a)) n
  in ((a, G\<lparr>pLabel := pLabel G(Exit G \<mapsto> ret)\<rparr>) # ba, ((\<Union>a\<in>
      set ba. Nodes (snd a)) \<union> Nodes G))")
apply (subgoal_tac "bb = (\<Union>a\<in>set aa. Nodes (snd a)) \<and> (\<forall>x\<in>set
    aa. finite (Nodes (snd x))) \<and>
  (\<forall>i j. i < length aa \<and> j < length aa \<and> n \<in> Nodes (snd (aa ! i)) \<
      and> n \<in> Nodes (snd (aa ! j)) \<longrightarrow> i = j)", simp)
apply (clarsimp simp add: Let_def, rule conjI, blast)
apply clarsimp
apply (case_tac ia, clarsimp, case_tac ja, simp+)
apply (frule_tac N="\<Union>a\<in>set ba. Nodes (snd a)" and n=n and b=b and
  n'="(SOME n. \<forall>a\<in>set ba. n \<notin> Nodes (snd a))" in new_CFG_distinct, simp
      +)
apply (smt in_set_conv_nth)
apply (cut_tac A="\<Union>a\<in>set ba. Nodes (snd a)" in fresh_new, simp+)
apply (smt in_set_conv_nth)
apply simp
apply (case_tac ja, clarsimp)
apply (frule_tac N="\<Union>a\<in>set ba. Nodes (snd a)" and n=n and b=b and
  n'="(SOME n. \<forall>a\<in>set ba. n \<notin> Nodes (snd a))" in new_CFG_distinct, simp
      +)
```

```
apply (smt in_set_conv_nth)
apply (cut_tac A="\<Union>a\<in>set ba. Nodes (snd a)" in fresh_new, simp+)
apply (smt in_set_conv_nth)
apply simp+
done

corollary method_CFGs_disjoint: "\<lbrakk>l = method_CFGs; i < length l; j < length l;
  n \<in> Nodes (snd (l ! i)); (n::'node) \<in> Nodes (snd (l ! j)); infinite (UNIV::'node
      set)\<rbrakk> \<Longrightarrow> i = j"
by (simp add: method_CFGs_def, rule_tac a="list_of_set {((c, s), b) |c s b. methods c s =
    Some b}"
  and b="[]" in fold_graphs_disjoint, simp_all)
(* Variable necessary to constrain polymorphism. *)

lemma fold_graphs_dom: "\<lbrakk>l = fst (fold (\<lambda>(a, b) (l, N). let n = new N 1 ! 0
     in
  let G = body_toCFG b (N::'node set) n in ((a, G\<lparr>pLabel := (pLabel G)(Exit G \<
      mapsto> ret)\<rparr>) # l,
  (N \<union> Nodes G))) a (b, c)); finite c; infinite (UNIV::'node set); (s, G) \<in> set
      l;
  \<forall>(s, G) \<in> set b. dom (pLabel G) = Nodes G\<rbrakk> \<Longrightarrow> dom (
      pLabel G) = Nodes G"
apply (induct a arbitrary: b c s G l, simp_all)
apply (metis (lifting, full_types) splitD)
apply (case_tac a1, simp add: Let_def)
apply (subgoal_tac "dom (pLabel (body_toCFG ba c (SOME n. n \<notin> c)\<lparr>pLabel :=
  pLabel (body_toCFG ba c (SOME n. n \<notin> c))(Exit (body_toCFG ba c (SOME n. n \<notin>
       c)) \<mapsto> ret)\<rparr>)) =
  Nodes (body_toCFG ba c (SOME n. n \<notin> c)\<lparr>pLabel := pLabel (body_toCFG ba c (
      SOME n. n \<notin> c))
  (Exit (body_toCFG ba c (SOME n. n \<notin> c)) \<mapsto> ret)\<rparr>)", simp+)
by (metis (lifting, no_types) toCFG_label_dom exit_in insert_Diff)

corollary method_CFGs_dom: "\<lbrakk>(s, G) \<in> set method_CFGs; infinite (UNIV::'node
    set)\<rbrakk> \<Longrightarrow>
  dom (pLabel G) = ((Nodes G)::'node set)"
by (simp add: method_CFGs_def, rule_tac b="[]" and c="{}" in fold_graphs_dom, simp+)

abbreviation "get_CFG c s \<equiv> map_of method_CFGs (c, s)"
abbreviation "main_CFG \<equiv> the (get_CFG main_class main_method)"
```

```
abbreviation "CFG_nodes \<equiv> \<Union>(Nodes ‘ snd ‘ set method_CFGs)"
abbreviation "CFG_edges \<equiv> \<Union>(Edges ‘ snd ‘ set method_CFGs)"
abbreviation "label_of n \<equiv> THE i. \<exists>G\<in>snd ‘ set method_CFGs. n \<in>
    Nodes G \<and> pLabel G n = Some i"

lemma label_of_correct: "\<lbrakk>(n::'node) \<in> CFG_nodes; infinite (UNIV::'node set)\<
    rbrakk> \<Longrightarrow>
  \<exists>!i. \<exists>G\<in>snd ‘ set method_CFGs. n \<in> Nodes G \<and> pLabel G n =
      Some i"
apply (subgoal_tac "\<exists>!G\<in>snd ‘ set method_CFGs. n \<in> Nodes G", clarsimp)
apply (frule method_CFGs_dom, simp)
apply (case_tac "pLabel ba n")
apply (metis domIff)
apply (rule_tac a=ab in ex1I, force, force)
apply clarsimp
apply (rule_tac a=ba in ex1I, force)
apply (clarsimp simp add: set_conv_nth)
apply (subgoal_tac "i = ia", clarsimp)
apply (metis snd_conv)
apply (rule_tac n=n in method_CFGs_disjoint, simp_all)
apply (metis snd_conv)
done

definition "call_edges \<equiv> \<Union>{{(n, Start G, mcall c), (Exit G, n', seq)} | n n'
    G c. \<exists>B l As.
  label_of n = call instance B c::l(As) \<and> get_CFG c (Sig B l As) = Some G \<and> (n, n
      ', seq) \<in> CFG_edges} \<union>
 \<Union>{{(n, Start G, mcall c'), (Exit G, n', seq)} | n n' G c'. \<exists>B c l As.
  label_of n = callvirt B c::l(As) \<and> get_CFG c' (Sig B l As) = Some G \<and> (n, n',
      seq) \<in> CFG_edges}"
(* In a well-formed program, we expect that c' inherits from c. *)

definition "CFG \<equiv> \<lparr>Nodes = CFG_nodes, Edges = call_edges \<union> CFG_edges,
    Start = Start main_CFG,
  Exit = Exit main_CFG, Label = \<lambda>n. if n \<in> CFG_nodes then label_of n else br\<
      rparr>"

end
```

```
(* This is a bit awkward. Heap reads only return Res, but sometimes we want to write a
    Boxed. *)
datatype ('class, 'href, 'field) val = Res "('href, 'field) result"
  | Boxed "('class, 'href, 'field) boxed_obj"
fun unbox_val where
"unbox_val (Res r) = r" |
"unbox_val (Boxed (BoxedObj c u)) = ObjResult u"

(* Despite the names given in BIL.thy, a stack frame actually contains an evaluation stack,
     an
   argument array, and an instruction pointer. In high-level BIL, everything but the
       argument array
   is implicit. *)
type_synonym ('node, 'href, 'field) stack_frame =
  "('href, 'field) result list \<times> ('href, 'field) frame \<times> 'node"
type_synonym ('node, 'href, 'field) config =
  "('node, 'href, 'field) stack_frame list \<times> ('node, 'href, 'field) stack_frame"
type_synonym ('thread, 'class, 'href, 'field) BIL_access =
  "('thread, ('href, 'field) pointer, ('class, 'href, 'field) val) access"

locale BIL_MM = BIL where methods =
  "methods::'class \<Rightarrow> ('class, 'mname) sig \<rightharpoonup> ('class, 'mname, '
      field) body" +
  memory_model where update_mem =
  "update_mem::'memory \<Rightarrow> ('thread, 'class, 'href, 'field) BIL_access set \<
      Rightarrow> 'memory \<Rightarrow> bool"
  for methods update_mem

context BIL begin

fun stack_lookup where
"stack_lookup \<sigma> args (FieldRef p f) = (case stack_lookup \<sigma> args p of Some (
    ObjResult u) \<Rightarrow> u f | _ \<Rightarrow> None)" |
"stack_lookup \<sigma> (.args(ws)) (FrameArg i j) = (if i - 1 < length \<sigma>
  then case \<sigma> ! (i - 1) of (_, .args(vs), _) \<Rightarrow> Some (vs ! (j - 1))
  else Some (ws ! (j - 1)))"

fun stack_update where
"stack_update \<sigma> args p v' = (case (get_base p, get_fields p) of (FrameArg i j, fs)
    \<Rightarrow>
```

```
  if i - 1 < length \<sigma> then case \<sigma> ! (i - 1) of (s, .args(vs), n) \<Rightarrow
      >
    (let j' = if j - 1 < length vs then j - 1 else length vs - 1 in
      case rupdate (vs ! j') fs v' of Some v'' \<Rightarrow>
        Some (\<sigma>[i - 1 := (s, .args(vs[j' := v'']), n)], args) | _ \<Rightarrow> None
            )
  else case args of .args(ws) \<Rightarrow> (let j' = if j - 1 < length ws then j - 1 else
      length ws - 1 in
      case rupdate (ws ! j') fs v' of Some v'' \<Rightarrow>
        Some (\<sigma>, .args(ws[j' := v''])) | _ \<Rightarrow> None))"

lemma stack_lookup_eq_lookup: "\<lbrakk>s = map (fst o snd) \<sigma> @ [args];
  \<exists>i\<le>Suc (length \<sigma>). \<exists>j. get_base p = FrameArg i j\<rbrakk> \<
      Longrightarrow>
  stack_lookup \<sigma> args p = lookup (h, s) p"
apply (induct p, simp_all)
apply (case_tac args, clarsimp)
apply (rule conjI, clarsimp simp add: nth_append)
apply (case_tac "\<sigma> ! (nat1 - 1)", simp split: frame.splits)
apply (clarsimp simp add: nth_append)
apply (auto split: frame.splits result.splits)
done

lemma stack_update_eq_update: "\<lbrakk>s = map (fst o snd) \<sigma> @ [args];
  \<exists>i\<le>Suc (length \<sigma>). \<exists>j. get_base p = FrameArg i j; stack_update
       \<sigma> args p v' = Some (\<sigma>', args')\<rbrakk> \<Longrightarrow>
  \<exists>s'. update (h, s) p v' = Some (h, s') \<and> s' = map (fst o snd) \<sigma>' @ [
      args']"
apply (clarsimp split: if_splits)
apply (clarsimp simp add: nth_append)
apply (case_tac "\<sigma> ! (i - 1)", simp)
apply (case_tac b, clarsimp simp add: list_update_append map_update split: if_splits)
apply (clarsimp simp add: nth_append)
apply (case_tac args, clarsimp simp add: list_update_append map_update split: if_splits)
done

lemma stack_update_Some: "\<exists>\<sigma>' args'. stack_update \<sigma> args (FrameArg i
    j) v' = Some (\<sigma>', args')"
by (auto split: prod.splits frame.splits)
```

```
end

locale BIL_graph = BIL_MM where methods =
  "methods::'class \<Rightarrow> ('class, 'mname) sig \<rightharpoonup> ('class, 'mname, '
      field) body" and update_mem =
  "update_mem::'memory \<Rightarrow> ('thread, 'class, 'href, 'field) BIL_access set \<
      Rightarrow> 'memory \<Rightarrow> bool"
  for methods update_mem +
  fixes Edges::"('node \<times> 'node \<times> 'class edge_type) set"
    and Label::"'node \<Rightarrow> ('class, 'mname, 'field) c_instr"
    and Exit::'node begin

(* Per-thread semantics with memory model *)
(* See ECMA-335 section I.12.3 for information about what's local and what's shared. *)
(* Memory model should include memory layout! *)
(* The ECMA standard doesn't have any VoidResults, so I'm considering them an artifact of
    the
   big-step semantics. *)
inductive BIL_step::"'thread \<Rightarrow> 'memory \<Rightarrow> ('node, 'href, 'field)
    config \<Rightarrow>
  ('thread, 'class, 'href, 'field) BIL_access set \<Rightarrow> ('node, 'href, 'field)
      config \<Rightarrow> bool" where
(* control flow *)
step_ldc [intro]: "\<lbrakk>Label n = ldc.i4 i; n \<noteq> Exit; n' = next_node Edges seq n
    \<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, IntResult i # vs, args, n')" |
step_br [intro]: "\<lbrakk>Label n = br; n \<noteq> Exit; n' = next_node Edges seq n\<
    rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs, args, n')" |
step_brtrue [intro]: "\<lbrakk>Label n = brtrue; vs = v # rest; e = (if v = IntResult 0
    then seq else branch);
  n \<noteq> Exit; n' = next_node Edges e n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, rest, args, n')" |
step_brfalse [intro]: "\<lbrakk>Label n = brfalse; vs = v # rest; e = (if v = IntResult 0
    then branch else seq);
  n \<noteq> Exit; n' = next_node Edges e n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, rest, args, n')" |
(* pointer types *)
step_ldind_heap [intro]: "\<lbrakk>Label n = ldind; vs = PtrResult ptr # rest; is_heap ptr;
```

```
  v \<in> can_read h t ptr; vs' = unbox_val v # rest; n \<noteq> Exit; n' = next_node Edges
       seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {Read t ptr v} (\<sigma>, vs', args, n')" |
step_ldind_stack [intro]: "\<lbrakk>Label n = ldind; vs = PtrResult ptr # rest; \<not>
    is_heap ptr;
  stack_lookup \<sigma> args ptr = Some v; vs' = v # rest; n \<noteq> Exit; n' = next_node
      Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs', args, n')" |
step_stind_heap [intro]: "\<lbrakk>Label n = stind; vs = v # PtrResult ptr # vs'; is_heap
    ptr;
  n \<noteq> Exit; n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {Write t ptr (Res v)} (\<sigma>, vs', args, n')" |
step_stind_stack [intro]: "\<lbrakk>Label n = stind; vs = v # PtrResult ptr # vs'; \<not>
    is_heap ptr;
  Some (\<sigma>', args') = stack_update \<sigma> args ptr v; n \<noteq> Exit; n' =
      next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>', vs', args', n')" |
(* arguments *)
step_ldarga [intro]: "\<lbrakk>Label n = ldarga j; length \<sigma> = i; j > 0;
  vs' = PtrResult (FrameArg (i + 1) (nat j)) # vs; n \<noteq> Exit; n' = next_node Edges
      seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs', args, n')" |
step_starg [intro]: "\<lbrakk>Label n = starg j; length \<sigma> = i; n \<noteq> Exit; n' =
     next_node Edges seq n; j > 0;
  vs = u # vs'; Some (\<sigma>', args') = stack_update \<sigma> args ((FrameArg (i + 1) (
      nat j))::('href, 'field) pointer) u\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>', vs', args', n')" |
(* for reference types only *)
step_newobj [intro]: "\<lbrakk>Label n = newobj (void c::.ctor(As)); c \<notin> ValueClass;
     vs = as @ rest;
  length as = length As; PBoxed p \<in> free_set h; vs' = PtrResult (PBoxed p) # rest; n \<
      noteq> Exit;
  n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {Write t (PBoxed p) (Boxed (BoxedObj c (map_of (zip
      (map fst (fields c)) (rev as)))))} (\<sigma>, vs', args, n')" |
step_callvirt [intro]: "\<lbrakk>Label n = callvirt (B c::l(As)); vs = as @ PtrResult (
    PBoxed p) # rest;
  length as = length As; Boxed (BoxedObj c' u) \<in> can_read h t (PBoxed p); c' <: c (*
      step is invalid otherwise *);
```

```
  \<sigma>' = \<sigma> @ [(vs, args, next_node Edges seq n)]; args' =  .args(PtrResult (
      PBoxed p) # rev as);
  n \<noteq> Exit; n' = next_node Edges (mcall c') n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {Read t (PBoxed p) (Boxed (BoxedObj c' u))} (\<sigma
      >', [], args', n')" |
(* for reference and value types *)
step_ldflda [intro]: "\<lbrakk>Label n = (ldflda a c::f); vs = PtrResult ptr # rest;
  vs' = PtrResult ((ptr).f) # rest; n \<noteq> Exit; n' = next_node Edges seq n\<rbrakk> \<
      Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs', args, n')" |
step_stfld_heap [intro]: "\<lbrakk>Label n = stfld a c::f; vs = v # PtrResult ptr # vs';
    is_heap ptr;
  n \<noteq> Exit; n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {Write t ((ptr).f) (Res v)} (\<sigma>, vs', args, n
      ')" |
step_stfld_stack [intro]: "\<lbrakk>Label n = stfld a c::f; n \<noteq> Exit; vs = v #
    PtrResult ptr # vs';
  \<not>is_heap ptr; Some (\<sigma>', args') = stack_update \<sigma> args ((ptr).f) v; n' =
       next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>', vs', args', n')" |
step_ret [intro]: "\<lbrakk>Label n = ret; n \<noteq> Exit; (* ignoring typing *) vs = v #
    rest;
  \<sigma> = \<sigma>' @ [(oldvs, args', n')]; vs' = v # oldvs; (n, n', seq) \<in> Edges\<
      rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>', vs', args', n')" |
(* for value types only *)
step_newobjv [intro]: "\<lbrakk>Label n = newobj (void vc::.ctor(As)); vc \<in> ValueClass;
     vs = as @ rest;
  length as = length As; vs' = ObjResult (map_of (zip (map fst (fields c)) (rev as))) #
      rest;
  n \<noteq> Exit; n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs', args, n')" |
step_call [intro]: "\<lbrakk>Label n = call instance (B vc::l(As)); vc \<in> ValueClass; vs
     = as @ PtrResult ptr # rest;
  length as = length As; \<sigma>' = \<sigma> @ [(vs, args, next_node Edges seq n)];
  args' = .args(PtrResult ptr # rev as); n \<noteq> Exit; n' = next_node Edges (mcall vc) n
      \<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>', [], args', n')" |
(* boxing and unboxing *)
```

```
step_box [intro]: "\<lbrakk>Label n = box vc; vc \<in> ValueClass; PBoxed p \<in> free_set
    h; vs = PtrResult ptr # rest;
  v \<in> can_read h t ptr; unbox_val v = ObjResult u; vs' = PtrResult (PBoxed p) # rest; n
      \<noteq> Exit;
  n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {Read t ptr v, Write t (PBoxed p) (Boxed (BoxedObj
      vc u))} (\<sigma>, vs', args, n')" |
step_unbox [intro]: "\<lbrakk>Label n = unbox vc; vc \<in> ValueClass; vs = PtrResult (
    PBoxed p) # rest;
  n \<noteq> Exit; n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs, args, n')" |
(* evaluation stack ops *)
step_dup [intro]: "\<lbrakk>Label n = dup; n \<noteq> Exit; vs = v # rest; vs' = v # v #
    rest; n' = next_node Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs', args, n')" |
step_pop [intro]: "\<lbrakk>Label n = pop; n \<noteq> Exit; vs = v # vs'; n' = next_node
    Edges seq n\<rbrakk> \<Longrightarrow>
  BIL_step t h (\<sigma>, vs, args, n) {} (\<sigma>, vs', args, n')"

lemma step_not_exit: "BIL_step t h (\<sigma>, vs, args, n) ops C' \<Longrightarrow> n \<
    noteq> Exit"
by (erule BIL_step.cases, auto)

end

lemmas body_conv_rev = body_conv [THEN sym]

context BIL begin

fun BIL_edges::"('class, 'name, 'field) c_instr \<Rightarrow> ('class edge_type \<
    Rightarrow> nat) set" where
"BIL_edges ret = {no_edges(seq := n) | n. True}" |
"BIL_edges brtrue = {no_edges(seq := 1, branch := 1)}" |
"BIL_edges brfalse = {no_edges(seq := 1, branch := 1)}" |
"BIL_edges (callvirt (B c::l(As))) = {\<lambda>e. case e of seq \<Rightarrow> 1 |
  mcall c' \<Rightarrow> if c' <: c then 1 else 0 | _ \<Rightarrow> 0}" |
"BIL_edges (call instance (B vc::l(As))) = {no_edges(seq := 1, mcall vc := 1)}" |
"BIL_edges _ = {no_edges(seq := 1)}"

declare [[goals_limit=2]]
```

```
lemma toCFG_out_edges: "\<lbrakk>infinite (UNIV::'node set); finite (N::'node set); finite
    (UNIV::'class set);
  pLabel (body_toCFG b N n) u = Some i; \<forall>K M. i \<noteq> ret \<and> i \<noteq>
     newobj K \<and>
  i \<noteq> callvirt (M::('class, 'mname) method_ref) \<and> i \<noteq> call instance M\<
     rbrakk> \<Longrightarrow>
  edge_types (out_edges (Edges (body_toCFG b N n)) u) \<in> BIL_edges i"
apply (frule dom_in, simp+)
apply (cut_tac P="\<lambda>b N (n::'node). finite N \<and> pLabel (body_toCFG' b N n) u =
    Some i \<longrightarrow>
  edge_types (out_edges (Edges (body_toCFG' b N n)) u) \<in> BIL_edges i" in body_toCFG'.
     induct,
  simp_all add: Let_def only_def follow_def)
apply (clarsimp simp add: out_edges_def edge_types_def intro!: ext)
apply (clarsimp simp add: sequence_def body_conv)
apply (erule disjE, frule_tac N="Na \<union> Nodes (body_toCFG a Na na)" in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na)" and n'="Exit (
    body_toCFG a Na na)"
  and b=ba in new_CFG_distinct, simp_all)
apply (clarsimp, frule_tac N=Na in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)
    )
  (Exit (body_toCFG a Na na)))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na)" and n'="Exit (
    body_toCFG a Na na)"
  and b=ba in new_CFG_distinct, simp_all)
apply (rule conjI, clarsimp simp add: body_conv)
apply (metis (lifting) exit_in exit_out)
apply (clarsimp simp add: body_conv, rule conjI, clarsimp)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG ba (Na \<union
    > Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
     ))) = {}", subgoal_tac
  "out_edges (Edges (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (SOME n. n \<
     notin> Na \<and> n \<notin> Nodes
```

```
  (body_toCFG a Na na)))) (Exit (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (
      SOME n. n \<notin> Na \<and>
  n \<notin> Nodes (body_toCFG a Na na)))) = {}", subgoal_tac "out_edges (Edges (body_toCFG
       c (Na \<union> Nodes
  (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na
      )) (SOME n. n \<notin> Na \<and>
  n \<notin> Nodes (body_toCFG a Na na)))) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (
      body_toCFG a Na na) \<and> n \<notin> Nodes
  (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n
      \<notin> Nodes
  (body_toCFG a Na na)))))) (Exit (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (
      SOME n. n \<notin> Na
  \<and> n \<notin> Nodes (body_toCFG a Na na)))) = {}", clarsimp intro!: ext simp add:
      edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n="Exit (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (SOME n. n
     \<notin> Na \<and> n \<notin>
  Nodes (body_toCFG a Na na)))" and N="Na \<union> Nodes (body_toCFG a Na na) \<union>
      Nodes (body_toCFG ba (Na \<union>
  Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
      Na na)))" and b=c and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin>
      Nodes (body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in new_CFG_distinct, simp+)
apply (metis exit_in)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
    \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in fresh_new, simp+)
apply (metis exit_in)
apply simp
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (smt exit_out)
apply (clarsimp, rule conjI, clarsimp)
```

```
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG a Na na)) =
    {}",
  subgoal_tac "out_edges (Edges (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (
      SOME n. n \<notin> Na \<and> n \<notin> Nodes
  (body_toCFG a Na na)))) (Exit (body_toCFG a Na na)) = {}", subgoal_tac "out_edges (Edges
  (body_toCFG c (Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
      \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))) (SOME n. n \<notin> Na \<and> n \<notin>
  Nodes (body_toCFG a Na na) \<and> n \<notin> Nodes (body_toCFG ba (Na \<union> Nodes (
      body_toCFG a Na na)) (SOME n.
  n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)))))) (Exit (body_toCFG a Na na
      )) = {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equalsOI, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n="Exit (body_toCFG a Na na)" and N="Na \<union> Nodes (body_toCFG a Na na
    ) \<union> Nodes (body_toCFG ba (Na \<union>
  Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
      Na na)))" and b=c and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin>
      Nodes (body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in new_CFG_distinct, simp+)
apply (metis exit_in)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
    \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in fresh_new, simp+)
apply (metis exit_in)
apply simp
apply (rule equalsOI, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (frule_tac n="Exit (body_toCFG a Na na)" and N="Na \<union> Nodes (body_toCFG a Na na
    )" and b=ba and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)" in
      new_CFG_distinct, simp+)
apply (metis exit_in)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na)" in fresh_new, simp+)
apply (metis exit_in, simp)
```

```
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply clarsimp
apply (erule disjE, frule_tac N="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (
    body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) u = {}", subgoal_tac "out_edges
    (Edges
  (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na))
  (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (
    body_toCFG ba (Na \<union>
  Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
      Na na)))" and b=c and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin>
      Nodes (body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in new_CFG_distinct, simp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
    \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in fresh_new, simp+,
  force, simp+)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (
    body_toCFG ba (Na \<union>
  Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
      Na na)))" and b=c and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin>
      Nodes (body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in new_CFG_distinct, simp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
    \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in fresh_new, simp+,
```

```
  force, simp+)
apply clarsimp
apply (erule_tac P="pLabel (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na))
  (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na))) u = Some i" in
      disjE)
apply (frule_tac N="Na \<union> Nodes (body_toCFG a Na na) " in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) u = {}", subgoal_tac "out_edges
    (Edges
  (body_toCFG c (Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
      \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))) (SOME n. n \<notin> Na \<and>
  n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin> Nodes (body_toCFG ba (Na \<union>
       Nodes (body_toCFG a Na na))
  (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)))))) u = {}", simp)
apply (rule equalsOI, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (
    body_toCFG ba (Na \<union>
  Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
      Na na)))" and b=c and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin>
      Nodes (body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in new_CFG_distinct, simp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
    \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in fresh_new, simp+,
  force, simp+)
apply (rule equalsOI, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na)" and b=ba and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)" in
      new_CFG_distinct, simp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na)" in fresh_new, simp+, force, simp
    +)
apply clarsimp
apply (frule_tac N=Na in dom_in, simp+)
```

```
apply (subgoal_tac "out_edges (Edges (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)
    )
  (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)))) u = {}",
      subgoal_tac "out_edges (Edges
  (body_toCFG c (Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
      \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))) (SOME n. n \<notin> Na \<and>
  n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin> Nodes (body_toCFG ba (Na \<union>
       Nodes (body_toCFG a Na na))
  (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)))))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (
    body_toCFG ba (Na \<union>
  Nodes (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
      Na na)))" and b=c and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na) \<and> n \<notin>
      Nodes (body_toCFG ba (Na \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in new_CFG_distinct, simp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na) \<union> Nodes (body_toCFG ba (Na
    \<union> Nodes
  (body_toCFG a Na na)) (SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)
      ))" in fresh_new, simp+,
  force, simp+)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na)" and b=ba and
  n'="SOME n. n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a Na na)" in
      new_CFG_distinct, simp+)
apply (cut_tac A="Na \<union> Nodes (body_toCFG a Na na)" in fresh_new, simp+, force, simp
    +)
(**)
apply (clarsimp simp add: body_conv)
apply (rule conjI, clarsimp)
apply (smt Un_iff body_nodes_finite exit_in exit_out2 finite_Un finite_insert insert_iff)
apply (clarsimp, rule conjI, clarsimp)
apply (smt Un_iff body_nodes_finite exit_in exit_out2 finite_Un finite_insert insert_iff)
apply (clarsimp, rule conjI, clarsimp)
```

```
apply (smt Un_iff body_nodes_finite exit_in exit_out2 finite_Un finite_insert insert_iff)
apply (clarsimp, rule conjI, clarsimp)
apply (subgoal_tac "out_edges (Edges (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<
    and> n \<notin> Na)))
  (Exit (body_toCFG ba (insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME n.
      n \<noteq> na \<and> n \<notin> Na))))
  (SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a (insert
       na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))))) = {}",
  subgoal_tac "out_edges (Edges (body_toCFG ba (insert na (Na \<union> Nodes (body_toCFG a
      (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))))
  (SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a (insert
       na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na)))))
  (Exit (body_toCFG ba (insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME n.
      n \<noteq> na \<and> n \<notin> Na))))
  (SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a (insert
       na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))))) = {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (frule_tac n="Exit (body_toCFG ba (insert na (Na \<union> Nodes (body_toCFG a (insert
     na Na)
  (SOME n. n \<noteq> na \<and> n \<notin> Na)))) (SOME n. n \<noteq> na \<and> n \<notin>
      Na \<and> n \<notin> Nodes (body_toCFG a (insert na Na)
  (SOME n. n \<noteq> na \<and> n \<notin> Na))))" and N="insert na (Na \<union> Nodes (
      body_toCFG a (insert na Na)
  (SOME n. n \<noteq> na \<and> n \<notin> Na)))" and b=ba and n'="SOME n. n \<noteq> na \<
      and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
  (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))" in new_CFG_distinct, simp+)
apply (cut_tac A="insert na (Na \<union> Nodes (body_toCFG a (insert na Na)
  (SOME n. n \<noteq> na \<and> n \<notin> Na)))" in fresh_new, simp+, force)
apply (metis exit_in)
apply (clarsimp, rule conjI, clarsimp)
apply (subgoal_tac "out_edges (Edges (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<
    and> n \<notin> Na)))
  (Exit (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))) = {}",
      subgoal_tac "out_edges
  (Edges (body_toCFG ba (insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME n.
       n \<noteq> na \<and> n \<notin> Na))))
```

```
  (SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a (insert
       na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na)))))
  (Exit (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))) = {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (frule_tac n="Exit (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<and> n \<
    notin> Na))" and
  N="insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<and
      > n \<notin> Na)))" and b=ba and
  n'="SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a (
      insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))"
  in new_CFG_distinct, simp+, force)
apply (cut_tac A="insert na (Na \<union> Nodes (body_toCFG a (insert na Na)
  (SOME n. n \<noteq> na \<and> n \<notin> Na)))" in fresh_new, simp+)
apply (metis exit_in, simp+)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (clarsimp, rule conjI, clarsimp)
apply (subgoal_tac "out_edges (Edges (body_toCFG a (insert u Na) (SOME n. n \<noteq> u \<
    and> n \<notin> Na))) u = {}",
  subgoal_tac "out_edges (Edges (body_toCFG ba (insert u (Na \<union> Nodes (body_toCFG a (
      insert u Na)
  (SOME n. n \<noteq> u \<and> n \<notin> Na)))) (SOME n. n \<noteq> u \<and> n \<notin> Na
       \<and> n \<notin> Nodes (body_toCFG a (insert u Na)
  (SOME n. n \<noteq> u \<and> n \<notin> Na))))) u = {}", clarsimp intro!: ext simp add:
      edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (frule_tac n=u and N="insert u (Na \<union> Nodes (body_toCFG a (insert u Na) (SOME n
    . n \<noteq> u \<and> n \<notin> Na)))"
  and b=ba and n'="SOME n. n \<noteq> u \<and> n \<notin> Na \<and> n \<notin> Nodes (
      body_toCFG a (insert u Na)
  (SOME n. n \<noteq> u \<and> n \<notin> Na))" in new_CFG_distinct, simp+)
apply (cut_tac A="insert u (Na \<union> Nodes (body_toCFG a (insert u Na) (SOME n. n \<
    noteq> u \<and> n \<notin> Na)))"
  in fresh_new, simp+, force, simp+)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
```

```
apply (frule_tac n=u and N="insert u Na" and b=a and n'="SOME n. n \<noteq> u \<and> n \<
    notin> Na" in new_CFG_distinct, simp+)
apply (cut_tac A="insert u Na" in fresh_new, simp+, force, simp+)
apply clarsimp
apply (erule disjE, frule_tac N="insert na (Na \<union> Nodes (body_toCFG a (insert na Na)
  (SOME n. n \<noteq> na \<and> n \<notin> Na)))" in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<
    and> n \<notin> Na))) u = {}",
  simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME
     n. n \<noteq> na \<and> n \<notin> Na)))"
  and b=ba and n'="SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (
      body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))"
  in new_CFG_distinct, simp+)
apply (cut_tac A="insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME n. n \<
    noteq> na \<and> n \<notin> Na)))"
  in fresh_new, simp+, force, simp+)
apply clarsimp
apply (frule_tac N="insert na Na" in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG ba (insert na (Na \<union> Nodes (
    body_toCFG a
  (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na)))) (SOME n. n \<noteq> na \<
      and> n \<notin> Na \<and> n \<notin> Nodes (body_toCFG a
  (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME
     n. n \<noteq> na \<and> n \<notin> Na)))"
  and b=ba and n'="SOME n. n \<noteq> na \<and> n \<notin> Na \<and> n \<notin> Nodes (
      body_toCFG a (insert na Na) (SOME n. n \<noteq> na \<and> n \<notin> Na))"
  in new_CFG_distinct, simp+)
apply (cut_tac A="insert na (Na \<union> Nodes (body_toCFG a (insert na Na) (SOME n. n \<
    noteq> na \<and> n \<notin> Na)))"
  in fresh_new, simp+, force, simp+)
apply (clarsimp simp add: body_conv)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG a Na na)) =
    {}",
  clarsimp intro!: ext simp add: edge_types_def)
```

```
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply clarsimp
apply (clarsimp simp add: sequence_def body_conv, rule conjI, clarsimp)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG ba (Na \<union
    > Nodes
  (body_toCFG a Na na)) (Exit (body_toCFG a Na na)))) = {}", subgoal_tac "out_edges (Edges
  (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (Exit (body_toCFG a Na na))))
  (Exit (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (Exit (body_toCFG a Na na))
      ))= {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na)" and n'="Exit (
    body_toCFG a Na na)"
  and b=ba in new_CFG_distinct, simp_all)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (metis (hide_lams, mono_tags) exit_out)
apply clarsimp
apply (erule disjE, frule_tac N="(Na \<union> Nodes (body_toCFG a Na na))" in dom_in, simp
    +)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="(Na \<union> Nodes (body_toCFG a Na na))" and b=ba and
  n'="Exit (body_toCFG a Na na)" in new_CFG_distinct, simp+)
apply clarsimp
apply (frule_tac N=Na in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)
    )
  (Exit (body_toCFG a Na na)))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="(Na \<union> Nodes (body_toCFG a Na na))" and b=ba and
  n'="Exit (body_toCFG a Na na)" in new_CFG_distinct, simp+)
apply (clarsimp simp add: out_edges_def edge_types_def intro!: ext)
apply (clarsimp simp add: body_conv)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG a Na na)) =
    {}",
```

```
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply clarsimp+
(**)
apply (case_tac args, clarsimp simp add: only_def Let_def edge_types_def out_edges_def
    intro!: ext
  split: if_splits)
apply (clarsimp simp add: Let_def sequence_def)
apply (erule disjE)
apply (subgoal_tac "out_edges (Edges (body_toCFG' a Na na)) u = {}", simp)
apply (metis body_conv_rev body_nodes_finite)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule_tac N=Na in toCFG_edges_in, simp_all, simp add: body_conv)
apply (frule_tac N="Na \<union> Nodes (body_toCFG' a Na na)" and a="Newobj list K" in
    dom_in,
  simp add: body_conv)
apply (simp add: body_conv_rev)
apply (metis (lifting, mono_tags) UnI2 body_conv_rev body_nodes_finite finite_Un
    new_CFG_distinct)
apply clarsimp
apply (subgoal_tac "out_edges (Edges (body_toCFG' (Newobj list K) (Na \<union> Nodes (
    body_toCFG' a Na na))
  (Exit (body_toCFG' a Na na)))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def simp del: body_toCFG'.simps)
apply (smt Un_iff body_conv_rev body_nodes_finite dom_in finite_Un new_CFG_distinct
    toCFG_edges_in)
apply (case_tac args, force simp add: only_def follow_def Let_def edge_types_def
    out_edges_def
  intro!: ext split: if_splits)
apply (clarsimp simp add: Let_def sequence_def)
apply (erule disjE)
apply (subgoal_tac "out_edges (Edges (body_toCFG' a Na na)) u = {}", simp)
apply (metis body_conv_rev body_nodes_finite)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule_tac N=Na in toCFG_edges_in, simp_all, simp add: body_conv)
apply (frule_tac N="Na \<union> Nodes (body_toCFG' a Na na)" and a="Callboxed aa list M" in
     dom_in,
  simp add: body_conv)
apply (simp add: body_conv_rev Let_def)
```

```
apply (metis (lifting, mono_tags) UnI2 body_conv_rev body_nodes_finite finite_Un
    new_CFG_distinct)
apply clarsimp
apply (subgoal_tac "out_edges (Edges (body_toCFG' (Callboxed aa list M) (Na \<union> Nodes
    (body_toCFG' a Na na))
  (Exit (body_toCFG' a Na na)))) u = {}", simp add: Let_def)
apply (rule equals0I, clarsimp simp add: out_edges_def simp del: body_toCFG'.simps)
apply (smt Un_iff body_conv_rev body_nodes_finite dom_in finite_Un new_CFG_distinct
    toCFG_edges_in)
apply (case_tac args, force simp add: only_def follow_def Let_def edge_types_def
    out_edges_def
  intro!: ext split: if_splits)
apply (clarsimp simp add: Let_def sequence_def)
apply (erule disjE)
apply (subgoal_tac "out_edges (Edges (body_toCFG' a Na na)) u = {}", simp)
apply (metis body_conv_rev body_nodes_finite)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule_tac N=Na in toCFG_edges_in, simp_all, simp add: body_conv)
apply (frule_tac N="Na \<union> Nodes (body_toCFG' a Na na)" and a="Callunboxed aa list M"
    in dom_in,
  simp add: body_conv)
apply (simp add: body_conv_rev Let_def)
apply (metis (lifting, mono_tags) UnI2 body_conv_rev body_nodes_finite finite_Un
    new_CFG_distinct)
apply clarsimp
apply (subgoal_tac "out_edges (Edges (body_toCFG' (Callunboxed aa list M) (Na \<union>
    Nodes (body_toCFG' a Na na))
  (Exit (body_toCFG' a Na na)))) u = {}", simp add: Let_def)
apply (rule equals0I, clarsimp simp add: out_edges_def simp del: body_toCFG'.simps)
apply (smt Un_iff body_conv_rev body_nodes_finite dom_in finite_Un new_CFG_distinct
    toCFG_edges_in)
apply (clarsimp simp add: body_conv)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG a Na na)) =
    {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (clarsimp simp add: sequence_def body_conv, rule conjI, clarsimp)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG ba (Na \<union
    > Nodes
```

```
  (body_toCFG a Na na)) (Exit (body_toCFG a Na na)))) = {}", subgoal_tac "out_edges (Edges
  (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (Exit (body_toCFG a Na na))))
  (Exit (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)) (Exit (body_toCFG a Na na))
     ))= {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="Na \<union> Nodes (body_toCFG a Na na)" and n'="Exit (
    body_toCFG a Na na)"
  and b=ba in new_CFG_distinct, simp_all)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (metis (hide_lams, mono_tags) exit_out)
apply clarsimp
apply (erule disjE, frule_tac N="(Na \<union> Nodes (body_toCFG a Na na))" in dom_in, simp
    +)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="(Na \<union> Nodes (body_toCFG a Na na))" and b=ba and
  n'="Exit (body_toCFG a Na na)" in new_CFG_distinct, simp+)
apply clarsimp
apply (frule_tac N=Na in dom_in, simp+)
apply (subgoal_tac "out_edges (Edges (body_toCFG ba (Na \<union> Nodes (body_toCFG a Na na)
    )
  (Exit (body_toCFG a Na na)))) u = {}", simp)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all)
apply (frule_tac n=u and N="(Na \<union> Nodes (body_toCFG a Na na))" and b=ba and
  n'="Exit (body_toCFG a Na na)" in new_CFG_distinct, simp+)
apply (clarsimp simp add: body_conv)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG a Na na)) =
    {}",
  clarsimp intro!: ext simp add: edge_types_def)
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (clarsimp simp add: body_conv)
apply (subgoal_tac "out_edges (Edges (body_toCFG a Na na)) (Exit (body_toCFG a Na na)) =
    {}",
  clarsimp intro!: ext simp add: edge_types_def)
```

```
apply (rule equals0I, clarsimp simp add: out_edges_def)
apply (frule toCFG_edges_in, simp_all, simp)
apply (clarsimp simp add: body_conv)
apply force
done

end

context BIL_MM begin

abbreviation step::"'thread \<Rightarrow> ('node, 'class edge_type, ('class, 'mname, 'field
    ) c_instr) flowgraph \<Rightarrow>
  'memory \<Rightarrow> ('node, 'href, 'field) config \<Rightarrow> ('thread, 'class, 'href
      , 'field) BIL_access set \<Rightarrow>
  ('node, 'href, 'field) config \<Rightarrow> bool" where
"step t G \<equiv> BIL_graph.BIL_step ValueClass fields inherits free_set can_read (Edges G
    ) (Label G) (Exit G) t"

lemma BIL_graph [simp, intro!]: "BIL_graph ValueClass Object fields inherits free_set
    methods update_mem"
by unfold_locales

lemmas step_induct = BIL_graph.BIL_step.induct [OF BIL_graph]
   and step_cases = BIL_graph.BIL_step.cases [OF BIL_graph]

lemma exists_call: "\<lbrakk>edge_types (out_edges (Edges G) p) = edge_type_case (Suc 0) 0
  (\<lambda>c'. if c' <: c then 1 else 0); c' <: c\<rbrakk> \<Longrightarrow> \<exists>p'.
      (p, p', mcall c') \<in> Edges G"
apply (drule_tac x="mcall c'" in cong, simp, simp add: edge_types_def)
apply (clarsimp simp add: card_Suc_eq)
by (metis (mono_tags) Collect_empty_eq out_by_t_def out_edges_by_t singleton_not_empty)

lemma step_along_edge: "\<lbrakk>step t G h (\<sigma>, vs, args, n) ops (\<sigma>', vs',
    args', n'); \<forall>u \<in> Nodes G. u \<noteq> Exit G \<longrightarrow>
  edge_types (out_edges (Edges G) u) \<in> BIL_edges (Label G u); n \<in> Nodes G; finite (
      Edges G)\<rbrakk> \<Longrightarrow>
  \<exists>t. (n, n', t) \<in> Edges G"
apply (erule_tac x=n in ballE, simp_all)
apply (erule step_cases)
apply (clarsimp, force simp add: out_edges_def)
```

```
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, rule conjI)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (clarsimp, rule conjI)
apply (force simp add: out_edges_def)
apply (force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, force simp add: out_edges_def)
apply (clarsimp, metis exists_call next_in)
apply (clarsimp, force simp add: out_edges_def)+
done

corollary step_along_edge': "\<lbrakk>step t G h (\<sigma>, vs, args, n) ops (\<sigma>', vs
    ', args', n');
  is_flowgraph G seq BIL_edges; n \<in> Nodes G\<rbrakk> \<Longrightarrow>
  \<exists>t. (n, n', t) \<in> Edges G"
apply (erule step_along_edge, auto simp add: is_flowgraph_def flowgraph_def
    flowgraph_axioms_def)
apply (erule pointed_graph.finite_edges)
done

corollary step_to_node: "\<lbrakk>step t G h (\<sigma>, vs, args, n) ops (\<sigma>', vs',
    args', n');
  is_flowgraph G seq BIL_edges; n \<in> Nodes G\<rbrakk> \<Longrightarrow>
  n' \<in> Nodes G"
by (drule step_along_edge', simp+, simp add: is_flowgraph_def flowgraph_def
    pointed_graph_def)

lemma step_transfer: "\<lbrakk>step t G h (\<sigma>, vs, args, n) ops b; n \<in> Nodes G; n
     \<in> Nodes G';
  out_edges (Edges G') n = out_edges (Edges G) n; Label G' n = Label G n;
  Exit G' \<notin> (Nodes G - {Exit G})\<rbrakk> \<Longrightarrow> step t G' h (\<sigma>,
      vs, args, n) ops b"
```

```
apply (cut_tac P="\<lambda>t h (\<sigma>, vs, args, n) ops b. \<forall>G'. (n \<in> Nodes G
    ' \<and> out_edges (Edges G') n =
  out_edges (Edges G) n \<and> Label G' n = Label G n \<and> Exit G' \<notin> (Nodes G - {
      Exit G}) \<and>
  n \<in> Nodes G) \<longrightarrow> step t G' h (\<sigma>, vs, args, n) ops b" in
      step_induct, simp_all)
apply clarsimp
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_ldc, rule BIL_graph, simp, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_br, rule BIL_graph, simp+, clarsimp, simp)
apply rule+
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=e in next_depends,
    simp+)
apply (simp split: if_splits)
apply (rule BIL_graph.step_brtrue, rule BIL_graph, simp, simp, simp, clarsimp, simp)+
apply rule+
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=e in next_depends,
    simp+)
apply (simp split: if_splits)
apply (rule BIL_graph.step_brfalse, rule BIL_graph, simp, simp, simp, clarsimp, simp)+
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_ldind_heap, rule BIL_graph, simp, force, simp+, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_ldind_stack, rule BIL_graph, simp, force, simp+, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_stind_heap, rule BIL_graph, simp, force, simp+, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_stind_stack, rule BIL_graph, simp, force, simp+, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_ldarga, rule BIL_graph, simp, force, simp+, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
```

```
apply (rule BIL_graph.step_starg, rule BIL_graph, simp+, force, simp+, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_newobj, rule BIL_graph, simp+, force, simp+)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t="mcall c'"
    in next_depends, simp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_callvirt, rule BIL_graph, simp+, force, simp+, force, force)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_ldflda, rule BIL_graph, simp+, force, simp+, force, simp+,
    clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_stfld_heap, rule BIL_graph, simp+, force, simp+, force, simp+,
    clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_stfld_stack, rule BIL_graph, simp+, force, simp+, force, force,
    simp+)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_ret, rule BIL_graph, simp+, force, force, simp+)
apply (metis (lifting) mem_Collect_eq out_by_t_def out_edges_by_t)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_newobjv, rule BIL_graph, simp+, force, simp+, force, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t="mcall vc" in
    next_depends, simp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_call, rule BIL_graph, simp+, force, simp+, force, force, simp+,
    clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_box, rule BIL_graph, simp+, force, simp+)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_unbox, rule BIL_graph, simp+, force, clarsimp+)
```

```
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_dup, rule BIL_graph, simp+, force, force, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'a" and p=na and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_pop, rule BIL_graph, simp+, force, force, clarsimp+)
done

end

type_synonym ('thread, 'node, 'class, 'mname, 'field) BIL_tCFG =
  "('thread, ('node, 'class edge_type, ('class, 'mname, 'field) c_instr) flowgraph) map"

print_locale BIL_MM
locale BIL_threads = BIL_MM where can_read =
  "can_read::'memory \<Rightarrow> 'thread \<Rightarrow> ('href, 'field) pointer \<
      Rightarrow> ('class, 'href, 'field) val set" and
  methods = "methods::'class \<Rightarrow> ('class, 'mname) sig \<Rightarrow> ('class, '
      mname, 'field) body option" +
  tCFG where instr_edges=BIL_edges and Seq=seq and
  CFGs="CFGs::('thread, 'node, 'class, 'mname, 'field) BIL_tCFG" for methods CFGs can_read
      +
  assumes infinite_nodes [simp]: "infinite (UNIV::'node set)"
begin

end

(* Patterns and substitution *)
datatype ('thread, 'node, 'class, 'mname, 'field) object = ONode 'node | OInt int
 | OInstr "('class, 'mname, 'field) c_instr" | OEdgeType "'class edge_type" | OThread '
     thread
 | OMRef "('class, 'mname) method_ref" | OKRef "'class constr_ref" | OClass 'class
 | OMName 'mname | OField 'field | OType "'class BIL_type" | OTypeList "'class BIL_type
     list"

(* pattern setup *)
datatype 'mvar mref_pat = MRPat "('mvar BIL_type, 'mvar) literal" 'mvar 'mvar 'mvar

fun type_fv where
"type_fv (tclass c) = {c}" |
```

```
"type_fv (valueclass c) = {c}" |
"type_fv (t&) = type_fv t" |
"type_fv t = {}"

lemma type_fv_finite [simp]: "finite (type_fv t)"
by (induct t, auto)

primrec mref_fv where
"mref_fv (MRPat t c m ts) = lit_fv type_fv t \<union> {c, m, ts}"

lemma mref_fv_finite [simp]: "finite (mref_fv m)"
apply (case_tac m, auto)
apply (case_tac literal, auto)
done

datatype 'mvar kref_pat = KRPat 'mvar 'mvar

primrec kref_fv where
"kref_fv (KRPat c ts) = {c, ts}"

lemma kref_fv_finite [simp]: "finite (kref_fv k)"
by (case_tac k, auto)

fun node_subst where
"node_subst CFGs \<sigma> (MVar m) = (case \<sigma> m of ONode t \<Rightarrow> Some t | _
    \<Rightarrow> None)" |
"node_subst CFGs \<sigma> (Inj (NExit t)) = (case \<sigma> t of OThread t' \<Rightarrow> (
    case CFGs t' of
  Some G \<Rightarrow> Some (Exit G) | _ \<Rightarrow> None) | _ \<Rightarrow> None)" |
"node_subst CFGs \<sigma> (Inj (NStart t)) = (case \<sigma> t of OThread t' \<Rightarrow> (
    case CFGs t' of
  Some G \<Rightarrow> Some (Start G) | _ \<Rightarrow> None) | _ \<Rightarrow> None)"

lemma node_same_subst: "\<forall>x\<in>node_fv n. \<sigma> x = \<sigma>' x \<Longrightarrow
    > node_subst CFGs \<sigma> n = node_subst CFGs \<sigma>' n"
apply (case_tac n, simp_all)
apply (case_tac data, simp_all)
done
```

```
fun type_subst_aux::"('mvar \<Rightarrow> ('thread, 'node, 'class, 'mname, 'field) object)
    \<Rightarrow>
  'mvar BIL_type \<rightharpoonup> 'class BIL_type" where
"type_subst_aux \<sigma> (tclass c) = (case \<sigma> c of OClass c' \<Rightarrow> Some (
    tclass c') | _ \<Rightarrow> None)" |
"type_subst_aux \<sigma> (valueclass c) = (case \<sigma> c of OClass c' \<Rightarrow> Some
    (valueclass c') | _ \<Rightarrow> None)" |
"type_subst_aux \<sigma> (tpointer t) = (case type_subst_aux \<sigma> t of Some t' \<
    Rightarrow> Some (tpointer t') | _ \<Rightarrow> None)" |
"type_subst_aux \<sigma> tvoid = Some tvoid" |
"type_subst_aux \<sigma> int32 = Some int32"

lemma type_aux_same_subst: "\<forall>x\<in>type_fv t. \<sigma> x = \<sigma>' x \<
    Longrightarrow> type_subst_aux \<sigma> t = type_subst_aux \<sigma>' t"
by (induct t, auto)

primrec type_subst where
"type_subst \<sigma> (MVar v) = (case \<sigma> v of OType t \<Rightarrow> Some t | _ \<
    Rightarrow> None)" |
"type_subst \<sigma> (Inj t) = type_subst_aux \<sigma> t"

lemma type_same_subst: "\<forall>x\<in>lit_fv type_fv t. \<sigma> x = \<sigma>' x \<
    Longrightarrow> type_subst \<sigma> t = type_subst \<sigma>' t"
by (case_tac t, auto intro: type_aux_same_subst)

primrec subst_list::"(('mvar \<Rightarrow> ('thread, 'node, 'class, 'mname, 'field) object)
     \<Rightarrow> 'a \<rightharpoonup> 'b) \<Rightarrow>
  ('mvar \<Rightarrow> ('thread, 'node, 'class, 'mname, 'field) object) \<Rightarrow> 'a
      list \<rightharpoonup> 'b list" where
"subst_list subst \<sigma> [] = Some []" |
"subst_list subst \<sigma> (x # rest) = (case (subst \<sigma> x, subst_list subst \<sigma>
    rest) of
  (Some x', Some rest') \<Rightarrow> Some (x' # rest') | _ \<Rightarrow> None)"

primrec mref_subst::"('mvar \<Rightarrow> ('thread, 'node, 'class, 'mname, 'field) object)
    \<Rightarrow>
  'mvar mref_pat \<rightharpoonup> ('class, 'mname) method_ref" where
"mref_subst \<sigma> (MRPat t c m ts) = (case (type_subst \<sigma> t, \<sigma> c, \<sigma>
    m, \<sigma> ts) of
```

```
  (Some t', OClass c', OMName m', OTypeList ts') \<Rightarrow> Some (MRef t' c' m' ts') | _
       \<Rightarrow> None)"

lemma mref_same_subst: "\<forall>x\<in>mref_fv M. \<sigma> x = \<sigma>' x \<Longrightarrow
    > mref_subst \<sigma> M = mref_subst \<sigma>' M"
apply (case_tac M, clarsimp)
apply (drule type_same_subst, simp)
done

primrec kref_subst where
"kref_subst \<sigma> (KRPat c ts) = (case (\<sigma> c, \<sigma> ts) of
  (OClass c', OTypeList ts') \<Rightarrow> Some (KRef c' ts') | _ \<Rightarrow> None)"

lemma kref_same_subst: "\<forall>x\<in>kref_fv K. \<sigma> x = \<sigma>' x \<Longrightarrow
    > kref_subst \<sigma> K = kref_subst \<sigma>' K"
by (case_tac K, simp)

type_synonym 'mvar instr_pattern = "('mvar, ('mvar BIL_type, 'mvar) literal, 'mvar,
  (int, 'mvar) literal, ('mvar kref_pat, 'mvar) literal, ('mvar mref_pat, 'mvar) literal)
      instr"

primrec instr_subst::"('mvar \<Rightarrow> ('thread, 'node, 'class, 'mname, 'field) object)
     \<Rightarrow>
  'mvar instr_pattern \<rightharpoonup> ('class, 'mname, 'field) c_instr" where
"instr_subst \<sigma> (ldc.i4 l) = (case l of MVar v \<Rightarrow> (case \<sigma> v of OInt
     i \<Rightarrow> Some (ldc.i4 i) | _ \<Rightarrow> None)
  | Inj i \<Rightarrow> Some (ldc.i4 i))" |
"instr_subst \<sigma> br = Some br" |
"instr_subst \<sigma> brtrue = Some brtrue" |
"instr_subst \<sigma> brfalse = Some brfalse" |
"instr_subst \<sigma> ldind = Some ldind" |
"instr_subst \<sigma> stind = Some stind" |
"instr_subst \<sigma> (ldarga l) = (case l of MVar v \<Rightarrow> (case \<sigma> v of OInt
     i \<Rightarrow>
  if i > 0 then Some (ldarga i) else None| _ \<Rightarrow> None)
  | Inj i \<Rightarrow> if i > 0 then Some (ldarga i) else None)" |
"instr_subst \<sigma> (starg l) = (case l of MVar v \<Rightarrow> (case \<sigma> v of OInt
    i \<Rightarrow>
  if i > 0 then Some (starg i) else None| _ \<Rightarrow> None)
  | Inj i \<Rightarrow> if i > 0 then Some (starg i) else None)" |
```

```
"instr_subst \<sigma> (newobj l) = (case l of MVar v \<Rightarrow> (case \<sigma> v of
    OKRef K \<Rightarrow> Some (newobj K) | _ \<Rightarrow> None)
  | Inj K \<Rightarrow> case kref_subst \<sigma> K of Some K' \<Rightarrow> Some (newobj K
      ') | _ \<Rightarrow> None)" |
"instr_subst \<sigma> (callvirt l) = (case l of MVar v \<Rightarrow> (case \<sigma> v of
    OMRef M \<Rightarrow> Some (callvirt M) | _ \<Rightarrow> None)
  | Inj M \<Rightarrow> case mref_subst \<sigma> M of Some M' \<Rightarrow> Some (callvirt
      M') | _ \<Rightarrow> None)" |
"instr_subst \<sigma> (call instance l) = (case l of MVar v \<Rightarrow> (case \<sigma> v
    of OMRef M \<Rightarrow> Some (call instance M) | _ \<Rightarrow> None)
  | Inj M \<Rightarrow> case mref_subst \<sigma> M of Some M' \<Rightarrow> Some (call
      instance M') | _ \<Rightarrow> None)" |
"instr_subst \<sigma> ret = Some ret" |
"instr_subst \<sigma> (ldflda t c::f) = (case (type_subst \<sigma> t, \<sigma> c, \<sigma>
    f) of
  (Some t', OClass c', OField f') \<Rightarrow> Some (ldflda t' c'::f') | _ \<Rightarrow>
      None)" |
"instr_subst \<sigma> (stfld t c::f) = (case (type_subst \<sigma> t, \<sigma> c, \<sigma> f
    ) of
  (Some t', OClass c', OField f') \<Rightarrow> Some (stfld t' c'::f') | _ \<Rightarrow>
      None)" |
"instr_subst \<sigma> (box c) = (case \<sigma> c of OClass c' \<Rightarrow> Some (box c') |
     _ \<Rightarrow> None)" |
"instr_subst \<sigma> (unbox c) = (case \<sigma> c of OClass c' \<Rightarrow> Some (unbox c
    ') | _ \<Rightarrow> None)" |
"instr_subst \<sigma> dup = Some dup" |
"instr_subst \<sigma> pop = Some pop"

primrec subst where
"subst \<sigma> (Inj i) = instr_subst \<sigma> i" |
"subst \<sigma> (MVar x) = (case \<sigma> x of OInstr i \<Rightarrow> Some i | _ \<
    Rightarrow> None)"

fun edge_type_subst where
"edge_type_subst \<sigma> (MVar v) = (case \<sigma> v of OEdgeType t \<Rightarrow> Some t |
     _ \<Rightarrow> None)" |
"edge_type_subst \<sigma> (Inj (mcall c)) = (case \<sigma> c of OClass c' \<Rightarrow>
    Some (mcall c') | _ \<Rightarrow> None)" |
"edge_type_subst \<sigma> (Inj seq) = Some seq" |
"edge_type_subst \<sigma> (Inj branch) = Some branch"
```

```
fun thread_subst where
"thread_subst \<sigma> v = (case \<sigma> v of OThread t \<Rightarrow> Some t | _ \<
    Rightarrow> None)"

fun int_of where
"int_of \<sigma> v = (case \<sigma> v of OInt i \<Rightarrow> Some i | _ \<Rightarrow> None
    )"

(* free variables *)
fun pattern_fv::"'mvar instr_pattern \<Rightarrow> 'mvar set" where
"pattern_fv (ldc.i4 i) = lit_fv (\<lambda>i. {}) i" |
"pattern_fv (ldarga i) = lit_fv (\<lambda>i. {}) i" |
"pattern_fv (starg i) = lit_fv (\<lambda>i. {}) i" |
"pattern_fv (newobj K) = lit_fv kref_fv K" |
"pattern_fv (callvirt M) = lit_fv mref_fv M" |
"pattern_fv (call instance M) = lit_fv mref_fv M" |
"pattern_fv (ldflda t c::f) = lit_fv type_fv t \<union> {c, f}" |
"pattern_fv (stfld t c::f) = lit_fv type_fv t \<union> {c, f}" |
"pattern_fv (box c) = {c}" |
"pattern_fv (unbox c) = {c}" |
"pattern_fv p = {}"

lemma pattern_fv_finite [simp]: "finite (pattern_fv p)"
apply (case_tac p, auto)
apply (case_tac constr, auto)
apply (case_tac method, auto)
apply (case_tac method, auto)
apply (case_tac type, auto)+
done

fun edge_type_fv where
"edge_type_fv (mcall c) = {c}" |
"edge_type_fv t = {}"

lemma edge_type_fv_finite [simp]: "finite (edge_type_fv t)"
by (case_tac t, auto)

lemma edge_type_same_subst: "\<forall>x\<in>lit_fv edge_type_fv t. \<sigma> x = \<sigma>' x
     \<Longrightarrow>
```

```
  edge_type_subst \<sigma> t = edge_type_subst \<sigma>' t"
apply (case_tac t, simp_all)
apply (case_tac data, simp_all)
done

lemma same_subst: "\<forall>x\<in>pattern_fv p. \<sigma> x = \<sigma>' x \<Longrightarrow>
    instr_subst \<sigma> p = instr_subst \<sigma>' p"
apply (case_tac p, simp_all)
apply (case_tac int_val, simp_all)
apply (case_tac int_val, simp_all)
apply (case_tac int_val, simp_all)
apply (case_tac constr, simp_all)
apply (drule kref_same_subst, simp)
apply (case_tac method, simp_all)
apply (drule mref_same_subst, simp)
apply (case_tac method, simp_all)
apply (drule mref_same_subst, simp)
apply (clarsimp, drule type_same_subst, simp)+
done

(* setup for PTRANS *)
sublocale BIL_threads \<subseteq> TRANS_preds where subst="\<lambda>G. subst::(string \<
    Rightarrow>
  ('thread, 'node, 'class, 'mname, 'field) object) \<Rightarrow> (string instr_pattern,
      string) literal \<rightharpoonup>
  ('class, 'mname, 'field) c_instr" and node_subst=node_subst
  and type_subst=edge_type_subst and thread_subst=thread_subst and var_subst="\<lambda>\<
      sigma> v. None::unit option"
  and eval_other="\<lambda>G q \<sigma> a::unit. True" and type_fv="lit_fv edge_type_fv"
      and pat_fv=pattern_fv
  and other_fv="\<lambda>a. {}" and instr_fv="\<lambda>i. {}" and int_of=int_of and
      instr_edges=BIL_edges and Seq=seq
apply (unfold_locales, auto)
apply (case_tac p, simp_all, force intro: same_subst)
apply (force intro: node_same_subst)
apply (force intro: edge_type_same_subst)
apply (case_tac ty, simp+)
by (metis infinite_UNIV_listI)
```

```
definition "start_state CFGs states mem \<equiv> \<forall>t. (\<forall>G. CFGs t = Some G
    \<longrightarrow>
  (\<exists>args. states t = Some ([], [], args, Start G))) \<and> (CFGs t = None \<
      longrightarrow> states t = None)"

lemma start_points [simp]: "start_state CFGs C0 m0 \<Longrightarrow>
  (\<lambda>C. \<lfloor>case C of (\<sigma>, vs, args, n) \<Rightarrow> n\<rfloor>) \<circ
      >\<^sub>m C0 = start_points CFGs"
by (force simp add: start_state_def start_points_def)

locale BIL_CFGs = BIL_threads where CFGs="CFGs::('thread, 'node, 'class, 'mname, 'field)
    BIL_tCFG" +
  tCFG'!: tCFG where CFGs="CFGs'::('thread, 'node, 'class, 'mname, 'field) BIL_tCFG" and
      Seq=seq
  and instr_edges=BIL_edges for CFGs CFGs'

context BIL_threads begin

lemma step_can_read: "\<lbrakk>step t G mem C ops C'; CFGs t = \<lfloor>G\<rfloor>;
  \<forall>l\<in>get_ptrs ops. can_read mem t l = can_read mem' t l; free_set mem =
      free_set mem'\<rbrakk> \<Longrightarrow>
  step t G mem' C ops C'"
apply (erule step_cases, simp_all)
apply (rule BIL_graph.step_ldc, rule BIL_graph, simp+)
apply (rule BIL_graph.step_br, rule BIL_graph, simp+)
apply (rule conjI, clarsimp)
apply (rule BIL_graph.step_brtrue, rule BIL_graph, simp+)
apply (clarsimp, rule BIL_graph.step_brtrue, rule BIL_graph, simp+)
apply (rule conjI, clarsimp)
apply (rule BIL_graph.step_brfalse, rule BIL_graph, simp+)
apply (clarsimp, rule BIL_graph.step_brfalse, rule BIL_graph, simp+)
apply (rule BIL_graph.step_ldind_heap, rule BIL_graph, simp+)
apply (rule BIL_graph.step_ldind_stack, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_stind_heap, rule BIL_graph, simp+)
apply (rule BIL_graph.step_stind_stack, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_ldarga, rule BIL_graph, simp+)
apply (rule BIL_graph.step_starg, rule BIL_graph, simp+)
apply (rule BIL_graph.step_newobj, rule BIL_graph, simp+)
apply (rule BIL_graph.step_callvirt, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_ldflda, rule BIL_graph, simp+, force, force, simp+)
```

```
apply (rule BIL_graph.step_stfld_heap, rule BIL_graph, simp+, force, force, simp+)
apply (rule BIL_graph.step_stfld_stack, rule BIL_graph, simp+, force, force, force, simp+)
apply (rule BIL_graph.step_ret, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_newobjv, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_call, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_box, rule BIL_graph, simp+)
apply (rule BIL_graph.step_unbox, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_dup, rule BIL_graph, simp+, force, simp+)
apply (rule BIL_graph.step_pop, rule BIL_graph, simp+)
done

lemma ops_thread: "\<lbrakk>step t G mem state ops state'; t \<in> dom CFGs; a \<in> ops\<
    rbrakk> \<Longrightarrow> get_thread a = t"
by (rule step_cases, auto)

end

print_locale sim_base
sublocale BIL_CFGs \<subseteq> sim?: sim_base where free_set=free_set and update_mem=
    update_mem
  and start_mem=start_mem and can_read=can_read and step_rel=step and get_point="\<lambda
      >(\<sigma>, vs, args, n). n"
  and instr_edges=BIL_edges and Seq=seq and start_state=start_state and CFGs=CFGs and CFGs
      '=CFGs'
apply unfold_locales
apply (clarsimp simp add: start_state_def safe_points_def start_points_def)
apply (case_tac C, case_tac C', clarsimp)
apply (erule step_along_edge', simp+)
apply (rule step_can_read, simp+)
apply (rule ops_thread, auto)
done

context BIL_CFGs begin

lemma run_safe: "run_prog CFGs (C, mem) \<Longrightarrow> safe_points CFGs (get_points C)"
by (rule run_prog_safe, simp, unfold_locales)

lemma run_safe': "run_prog CFGs' (C, mem) \<Longrightarrow> safe_points CFGs' (get_points C
    )"
by (rule run_prog_safe, simp, unfold_locales)
```

```
end

(* memory models *)
fun read_ref where
"read_ref v [] = Some v" |
"read_ref (Boxed (BoxedObj c u)) fs = (case rlookup (ObjResult u) fs of Some v \<Rightarrow
    > Some (Res v) | _ \<Rightarrow> None)" |
"read_ref (Res v) fs = (case rlookup v fs of Some v' \<Rightarrow> Some (Res v') | _ \<
    Rightarrow> None)"

fun write_ref where
"write_ref (Boxed (BoxedObj c u)) fs (Res v') = (case rupdate (ObjResult u) fs v' of Some (
    ObjResult u') \<Rightarrow>
  Some (Boxed (BoxedObj c u')) | _ \<Rightarrow> None)" |
"write_ref (Res v) fs (Res v') = (case rupdate v fs v' of Some v'' \<Rightarrow> Some (Res
    v'') | _ \<Rightarrow> None)" |
"write_ref _ _ _ = None"

locale BIL_SC = BIL where methods="methods::'class \<Rightarrow> ('class, 'mname) sig \<
    rightharpoonup> ('class, 'mname, 'field) body" +
  tCFG where instr_edges=BIL_edges and Seq=seq and CFGs="CFGs::('thread, 'node, 'class, '
      mname, 'field) BIL_tCFG" +
  SC where undef=undefined for methods CFGs +
  assumes infinite_nodes [simp]: "infinite (UNIV::'node set)"

sublocale BIL_SC \<subseteq> BIL_threads where update_mem=update_mem and free_set=free_set
  and can_read=can_read and start_mem=start_mem
by (unfold_locales, metis SC.alloc_not_free, metis SC.stays_not_free, simp)

locale BIL_CFGs_SC = BIL_SC where CFGs="CFGs::('thread, 'node, 'class, 'mname, 'field)
    BIL_tCFG" +
  tCFG': tCFG where CFGs="CFGs'::('thread, 'node, 'class, 'mname, 'field) BIL_tCFG" and Seq
      =seq
  and instr_edges=BIL_edges for CFGs CFGs'

sublocale BIL_CFGs_SC \<subseteq> CFGs?: BIL_CFGs where update_mem=update_mem and free_set=
    free_set
  and can_read=can_read and start_mem=start_mem
by (unfold_locales)
```

```
end

(* BIL_opt.thy *)
(* William Mansky *)
(* Optimizations on GraphBIL. *)

theory BIL_opt
imports BIL_graph trans_preds
begin

(* Looking to optimize instructions that pop things off the evaluation stack: stind, starg,
     stfld.
   Let's start with the one that can't affect shared memory: starg. *)
definition "safe t j \<equiv> \<not>sc (stmt t ldind \<or>sc stmt t stind \<or>sc
  (stmt t (starg (MVar j))) \<or>sc stmt t ret \<or>sc
  SCEx ''M'' (stmt t (callvirt (MVar ''M'')) \<or>sc stmt t (call instance (MVar ''M'')))
      \<or>sc
  SCExs [''A'', ''c'', ''f''] (stmt t (stfld (MVar ''A'') ''c''::''f''))))"

definition "safe_instr i j \<equiv> \<not>(i = ldind \<or> i = stind \<or> i = starg j \<or
    > i = ret \<or>
  (\<exists>M. i = callvirt M \<or> i = call instance M) \<or> (\<exists>t c f. i = stfld t
      c::f))"

definition "RSE_cond \<equiv> A safe ''t'' ''j'' \<U>
  (\<not>sc (node ''t'' ''n'') \<and>sc stmt ''t'' (starg (MVar ''j'')))"

definition "RSE \<equiv> TIf [AReplace (MVar ''n'') [Inj pop]]
  (EF (node ''t'' ''n'' \<and>sc stmt ''t'' (starg (MVar ''j'')) \<and>sc
  A node ''t'' ''n'' \<U> (\<not>sc (node ''t'' ''n'') \<and>sc RSE_cond)))"

context BIL_threads begin

lemma RSE_one_graph: "CFGs' \<in> trans_sf RSE \<tau> CFGs \<Longrightarrow>
  \<exists>t G G'. CFGs t = Some G \<and> CFGs' t = Some G' \<and> (\<forall>t'. t' \<noteq
      > t \<longrightarrow> CFGs' t' = CFGs t')"
by (force simp add: RSE_def action_list_sf_def split: if_splits)

lemma RSE_tCFG: "CFGs' \<in> trans_sf RSE \<tau> CFGs \<Longrightarrow> tCFG CFGs'
    BIL_edges seq"
```

```
apply (clarsimp simp add: RSE_def action_list_sf_def split: if_splits)
apply (erule_tac x=ya in allE, clarsimp)
apply (unfold_locales, simp_all split: if_splits)
apply (case_tac "\<sigma> ''j''", simp_all split: if_splits)
apply (frule_tac l=l in thread_of_path, simp+, clarsimp)

apply (frule_tac t="thread_of a CFGs" in CFGs, clarsimp simp add: is_flowgraph_def
    flowgraph_def
  flowgraph_axioms_def)
apply (erule_tac x=a in allE, erule impE, assumption, simp)
apply (frule_tac s="starg int" in sym, simp)
apply (erule CFGs)
apply (clarsimp, erule disjoint, simp+)+
apply (erule disjoint, simp+)
done

lemma RSE_cond_by_thread: "by_thread RSE_cond ''t''"
by (auto simp add: RSE_cond_def safe_def)

lemma models_safe: "\<lbrakk>models CFGs \<sigma> q (safe t j); \<sigma> t = OThread t'; \<
    sigma> j = OInt j'; j' > 0;
  q t' = Some n; CFGs t' = Some G; n \<noteq> Exit G; t \<noteq> ''f''; t \<noteq> ''c''; t
      \<noteq> ''A''; t \<noteq> ''M''\<rbrakk> \<Longrightarrow>
  safe_instr (Label G n) j'"
by (force simp add: safe_def safe_instr_def split: object.splits if_splits)

lemma safe_models: "\<lbrakk>safe_instr (Label G n) j'; \<sigma> t = OThread t'; \<sigma> j
     = OInt j'; j' > 0;
  q t' = Some n; CFGs t' = Some G; n \<noteq> Exit G; t \<noteq> ''f''; t \<noteq> ''c''; t
      \<noteq> ''A''; t \<noteq> ''M''\<rbrakk> \<Longrightarrow>
  models CFGs \<sigma> q (safe t j)"
by (clarsimp simp add: safe_def safe_instr_def split: object.splits)

lemma safe_step_transfer: "\<lbrakk>step t G h (\<sigma>, vs, args, n) ops (\<sigma>', vs',
     args', n'); n \<in> Nodes G; n \<in> Nodes G';
  out_edges (Edges G') n = out_edges (Edges G) n; Label G' n = Label G n; safe_instr (Label
      G n) j;
  \<forall>j. Label G n \<noteq> starg j; Exit G' \<notin> (Nodes G - {Exit G})\<rbrakk> \<
      Longrightarrow>
```

```
  args' = args \<and> step t G' h (\<sigma>, vs, args'', n) ops (\<sigma>', vs', args'', n
      ')"
apply (rule step_cases, simp, simp_all)
apply clarsimp
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_ldc, rule BIL_graph, simp, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_br, rule BIL_graph, simp+, clarsimp, simp)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=e in next_depends, simp
    +)
apply (simp split: if_splits)
apply (rule BIL_graph.step_brtrue, rule BIL_graph, simp, simp, simp, clarsimp, simp)+
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=e in next_depends, simp
    +)
apply (simp split: if_splits)
apply (rule BIL_graph.step_brfalse, rule BIL_graph, simp, simp, simp, clarsimp, simp)+
apply (clarsimp simp add: safe_instr_def)+
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_ldarga, rule BIL_graph, simp, force, simp+, clarsimp+)
apply force
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_newobj, rule BIL_graph, simp+, force, simp+)
apply (clarsimp simp add: safe_instr_def)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_ldflda, rule BIL_graph, simp+, force, simp+, force, simp+,
    clarsimp+)
apply (clarsimp simp add: safe_instr_def)+
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_newobjv, rule BIL_graph, simp+, force, simp+, force, clarsimp+)
apply (clarsimp simp add: safe_instr_def)
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_box, rule BIL_graph, simp+, force, simp+)
```

```
apply (clarsimp, cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in
    next_depends, simp+)
apply (rule BIL_graph.step_unbox, rule BIL_graph, simp+, force, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_dup, rule BIL_graph, simp+, force, force, clarsimp+)
apply (cut_tac Edges="Edges G" and Edges'="Edges G'" and p=n and t=seq in next_depends,
    simp+)
apply (rule BIL_graph.step_pop, rule BIL_graph, simp+, force, force, clarsimp+)
done

end

context BIL_CFGs begin

lemma RSE_cond_step: "\<lbrakk>step t G m (s, vs, args, n) ops (s', vs', args', n'); models
     CFGs \<sigma> q RSE_cond;
  \<sigma> ''t'' = OThread t; q t = Some n; CFGs t = Some G; n \<in> Nodes G; \<sigma> ''j
      '' = OInt j;
  Label G n \<noteq> starg j\<rbrakk> \<Longrightarrow> models CFGs \<sigma> (q(t \<mapsto>
       n')) RSE_cond"
apply (clarsimp simp add: RSE_cond_def split: if_splits)
apply (frule_tac CFGs=CFGs in step_increment_path, unfold_locales, simp+, simp add:
    map_upd_triv)
apply (erule_tac x="[q] \<frown> l" in ballE, simp_all, clarsimp)
apply (case_tac i, clarsimp+)
apply (rule_tac x=nat in exI, simp)
apply (erule disjE, clarsimp)
apply (case_tac "\<sigma> ''n''", simp_all, force)
apply (metis Suc_less_eq i_append_Nil i_append_nth_Cons_Suc)
apply (cut_tac q=q in exists_path, simp)
done

corollary RSE_cond_one_step: "\<lbrakk>one_step t G ((s, vs, args, n), m) ((s', vs', args',
     n'), m');
  models CFGs \<sigma> q RSE_cond; \<sigma> ''t'' = OThread t; q t = Some n; CFGs t = Some
      G; n \<in> Nodes G;
  \<sigma> ''j'' = OInt j; Label G n \<noteq> starg j\<rbrakk> \<Longrightarrow> models
      CFGs \<sigma> (q(t \<mapsto> n')) RSE_cond"
by (erule one_step.cases, clarsimp, rule RSE_cond_step, simp+)
```

```
lemma safe_fv [simp]: "cond_fv pred_fv (safe t j) = {t, j}"
by (auto simp add: safe_def)

lemma RSE_cond_fv [simp]: "cond_fv pred_fv RSE_cond = {''n'', ''t'', ''j''}"
by (auto simp add: RSE_cond_def)

(* Memory-model-independent. *)
definition "RSE_rel t G' G C C' \<equiv> case (C, C') of ((s, mem), (s', mem')) \<
    Rightarrow> mem = mem' \<and>
  (s = s' \<or> (\<exists>\<sigma> vs args args' n j v n'. s = (\<sigma>, vs, .args(args),
      n) \<and> s' = (\<sigma>, vs, .args(args'), n) \<and>
   args' = args[if j - Suc 0 < length args then j - 1 else length args - 1 := v] \<and>
   Label G n' \<noteq> Label G' n' \<and> Label G n' = starg (int j) \<and>
   models CFGs ((\<lambda>x. undefined)(''t'' := OThread t, ''j'' := OInt (int j), ''n'' :=
       ONode n')) [t \<mapsto> n] RSE_cond))"

declare split_if [split del]

lemma if_length_upd [simp]: "(if ja - Suc 0 < length args then ja - 1 else
  length (args[if j - Suc 0 < length args then j - 1 else length args - 1 := v]) - 1) =
  (if ja - Suc 0 < length args then ja - 1 else length args - 1)"
by (simp split: if_splits)

theorem "\<lbrakk>CFGs' \<in> trans_sf RSE \<tau> CFGs; CFGs t = Some G; CFGs' t = Some G';
  \<forall>t'. t' \<noteq> t \<longrightarrow> CFGs' t' = CFGs t'\<rbrakk> \<Longrightarrow
      >
  tCFG_sim (lift_reach_sim_rel (RSE_rel t) CFGs' CFGs t) (op =) CFGs' CFGs conc_step UNIV (
      get_mem o snd)"
apply (rule sim_no_mem, simp_all)
apply (clarsimp simp add: RSE_def)
apply (case_tac "\<sigma> ''t''", simp_all)
apply (erule_tac x=y and P="\<lambda>y. ?P y \<and> ?Q y" in allE, clarsimp)
apply (erule disjE)
apply (case_tac "\<sigma> ''n''", simp_all, clarsimp)
apply (case_tac "\<sigma> ''j''", simp_all split: if_splits)
apply (unfold_locales, clarsimp simp add: trsys_of_tCFG_def add_reach_def RSE_rel_def)
apply (frule run_safe, frule run_safe')
apply (rule one_step.cases, simp, clarsimp)
apply (clarsimp simp add: action_list_sf_def)
```

```
apply (case_tac "y \<noteq> t")
(* the transformation cannot leave the graph unchanged *)
apply (cut_tac l=l and CFGs=CFGs in tCFG.thread_of_path)
apply (unfold_locales, simp+, clarsimp)
apply (case_tac "node \<notin> nodes CFGs", simp+)
apply (clarsimp simp add: RSE_cond_def)
apply (erule allE, erule impE, assumption, simp)
apply (frule_tac f=Label in arg_cong, simp)
apply (drule_tac x=node in fun_cong, simp)
(* main case *)
apply (cut_tac l=l and CFGs=CFGs in tCFG.thread_of_path)
apply (unfold_locales, simp+)
apply (case_tac "node \<notin> nodes CFGs", simp+, clarsimp)
apply (erule disjE, clarsimp)
(* we have not yet hit the changed instruction *)
apply (case_tac "bb \<noteq> node")
(* another ordinary step *)
apply ((rule exI)+, rule context_conjI)
apply (rule step_single, rule_tac G="G\<lparr>Label := (Label G)(node := pop)\<rparr>" in
    step_transfer, simp+)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp+)
apply (rule_tac x="S(thread_of node CFGs \<mapsto> (al, am, an, bf))" in exI, simp)
apply (rule conjI, rule run_prog_step, simp+)
apply (rule_tac x="S'(thread_of node CFGs \<mapsto> (al, am, an, bf))" in exI, simp)
apply (rule run_prog_step, simp+)
apply (rule_tac G="G\<lparr>Label := (Label G)(node := pop)\<rparr>" in step_transfer, simp
    +)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp+)
(* the modified node *)
apply (rule step_cases, assumption, simp_all)
apply (cut_tac \<sigma>=\<sigma>' and args=args and i="Suc (length \<sigma>')" and j="nat
    int" and v'=v
  in stack_update_Some, clarify)
apply ((rule exI)+, rule context_conjI)
apply (rule step_single, rule BIL_graph.step_starg, rule BIL_graph, simp+, rule sym, simp+,
     force,
  force, simp+)
```

```
apply (rule_tac x="S(thread_of n CFGs \<mapsto> (\<sigma>', vs', args, next_node (Edges G)
    seq n))" in exI, simp)
apply (rule conjI, rule run_prog_step, simp+)
apply (rule_tac x="S'(thread_of n CFGs \<mapsto> (\<sigma>'', vs', args', next_node (Edges
    G) seq n))" in exI, simp)
apply (rule conjI, rule run_prog_one_step, simp+)
apply (rule disjI2, case_tac args, clarsimp)
apply (rule_tac x="nat int" in exI, rule conjI, force)
apply (cut_tac q="l i(thread_of n CFGs \<mapsto> next_node (Edges G) seq n)" and CFGs=CFGs
  in tCFG.exists_path, unfold_locales, clarsimp)
apply (frule_tac G=G and CFGs=CFGs and l=la in one_step_increment_path, unfold_locales,
    simp+)
apply (rule_tac CFGs=CFGs in tCFG.node_of_graph, unfold_locales, simp+, simp add:
    map_upd_triv)
apply (rule_tac x=n in exI, clarsimp)
apply (rule conjI, clarsimp)
apply (erule_tac x="[l i] \<frown> la" in ballE, simp_all, clarsimp)
apply (case_tac ia, simp+)
apply (case_tac nata, simp_all)
apply (cut_tac q="start_points CFGs" and CFGs=CFGs in tCFG.combine_paths, unfold_locales,
    simp+)
apply (rule_tac l="(l \<Down> i @ [l i]) \<frown> la" and i="Suc i" in by_thread_pathD1 [OF
     RSE_cond_by_thread],
  simp+)
apply unfold_locales
apply (rule cond_same_subst [THEN subst], simp_all)
apply unfold_locales
apply (erule_tac x=1 in allE, clarsimp)
apply (cut_tac CFGs=CFGs in tCFG.CFGs, unfold_locales, simp+)
apply (simp add: is_flowgraph_def)
apply (frule flowgraph.instr_edges_ok)
apply (rule_tac CFGs=CFGs in tCFG.node_of_graph, unfold_locales, simp+)
apply (subgoal_tac "finite (Edges G)", clarsimp simp add: out_edges_def)
apply (drule_tac s="starg int" in sym, clarsimp)
apply (frule next_in, simp)
apply (drule_tac u=n in flowgraph.no_loop, simp)
apply (clarsimp simp add: flowgraph_def, erule pointed_graph.finite_edges)
(* between the changed instruction and the following store *)
apply clarsimp
apply (case_tac "n' \<noteq> node", simp+)
```

```
apply (case_tac "ba = node", clarsimp simp add: RSE_cond_def)
(* cannot be at the changed instruction *)
apply (cut_tac q="[thread_of node CFGs \<mapsto> node]" and CFGs=CFGs in tCFG.exists_path,
    unfold_locales,
  clarsimp)
apply (erule_tac x=la in ballE, simp_all, clarsimp)
apply (case_tac ia, simp+)
apply (case_tac "j \<le> 0", simp+)
apply (erule_tac x=0 in allE, simp)
apply (drule models_safe, simp+)
apply (simp add: safe_instr_def)
(* so, both threads take the same step *)
apply (case_tac "\<exists>j'. Label G ba = starg j' \<and> (if j - Suc 0 < length args
  then j - 1 else length args - 1) = (if nat j' - Suc 0 < length args
  then nat j' - 1 else length args - 1)", clarsimp)
(* at the following store *)
apply (rule step_cases, simp, simp_all, clarsimp)
apply ((rule exI)+, rule context_conjI)
apply (rule step_single, rule BIL_graph.step_starg, rule BIL_graph, simp+, force, force,
    simp+)
apply (rule_tac x="S(thread_of node CFGs \<mapsto> (\<sigma>', vs', .args(args[if nat ja -
    Suc 0 < length args
  then nat ja - 1 else length args - 1 := u]), next_node (Edges G) seq n))" in exI, simp)
apply (rule conjI, rule run_prog_step, simp+)
apply (rule_tac x="S'(thread_of node CFGs \<mapsto> (\<sigma>', vs', .args(args[if nat ja -
     Suc 0 < length args
  then nat ja - 1 else length args - 1 := u]), next_node (Edges G) seq n))" in exI, simp)
apply (rule run_prog_one_step, simp+)
(* Speed this up by defining a bounds_checked function. *)
(* in between, at a safe instruction *)
apply (subgoal_tac "safe_instr (Label G ba) int")
apply (case_tac "\<exists>j'. Label G ba = starg j'", clarsimp)
apply (rule step_cases, simp, simp_all, clarsimp)
apply ((rule exI)+, rule context_conjI)
apply (rule step_single, rule BIL_graph.step_starg, rule BIL_graph, simp+, force, force,
    simp+)
apply (rule_tac x="S(thread_of node CFGs \<mapsto> (\<sigma>', vs', .args(args[if nat ja -
    Suc 0 < length args
  then nat ja - 1 else length args - 1 := u]), next_node (Edges G) seq n))" in exI, simp)
apply (rule conjI, rule run_prog_step, simp+)
```

```
apply (rule_tac x="S'(thread_of node CFGs \<mapsto> (\<sigma>', vs', .args(args[if j - Suc
    0 < length args
  then j - 1 else length args - 1 := v, if nat ja - Suc 0 < length args then nat ja - 1
  else length args - 1 := u]), next_node (Edges G) seq n))" in exI, simp)
apply (rule conjI, rule run_prog_one_step, simp+)
apply (rule disjI2, rule exI, rule conjI, rule_tac x=v in exI, rule list_update_swap, force
    )
apply (drule_tac G=G in RSE_cond_one_step, simp+)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp+)
apply (clarsimp split: if_splits)
apply (rule_tac x=node in exI, simp)
apply (cut_tac G="G\<lparr>Label := (Label G)(node := pop)\<rparr>" and args''=".args(args[
    if j - Suc 0 <
  length args then j - 1 else length args - 1 := v])" in safe_step_transfer, simp+)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp+)
apply ((rule exI)+, rule context_conjI)
apply (rule step_single, simp+, force, simp+)
apply (rule_tac x="S(thread_of node CFGs \<mapsto> (al, am, .args(args), bf))" in exI, simp
    )
apply (rule conjI, rule run_prog_step, simp+)
apply (simp add: fun_upd_def, simp+)
apply (rule_tac x="S'(thread_of node CFGs \<mapsto> (al, am, .args(args[if j - Suc 0 <
    length args
  then j - 1 else length args - 1 := v]), bf))" in exI, simp)
apply (rule conjI, rule run_prog_one_step, simp+)
apply (rule disjI2, rule exI, rule conjI, force)
apply (clarsimp, drule RSE_cond_step, simp+)
apply (simp add: safe_points_def, erule_tac x="thread_of node CFGs" in allE, simp+)
apply (rule_tac x=node in exI, simp)
apply (clarsimp simp add: RSE_cond_def)
apply (cut_tac q="[thread_of node CFGs \<mapsto> ba]" and CFGs=CFGs in tCFG.exists_path,
  unfold_locales, clarsimp)
apply (erule_tac x=la in ballE, simp_all, clarsimp)
apply (case_tac ia, simp split: if_splits)
apply (erule_tac x=0 and P="\<lambda>x. x < ia \<longrightarrow> ?P x" in allE, simp)
apply (erule models_safe, simp+)
apply (rule BIL_graph.step_not_exit, rule BIL_graph, simp+)
apply (case_tac "\<sigma> ''j''", simp_all split: if_splits)
apply (simp add: RSE_rel_def)+
```

```
done

end

end
```

A.2 Executable Semantics in F#

The following code was developed with Microsoft Visual Studio 2012 on Windows 8, using Microsoft Visual F# 2012 and Z3 4.3.0. Note that F# is whitespace-sensitive, and line wrapping may have introduced indentation errors in this listing.

```
module Ptrans

open System.Collections.Generic
open Microsoft.Z3
open System.Reflection
open Microsoft.FSharp.Reflection

(* utility functions *)
let upd f x y z = if z = x then y else f z

let contains x l = List.exists (fun y -> y = x) l

let conmap f l = List.concat (Seq.map f l)

let rec remdups l =
    match l with
    | [] -> []
    | x::xs -> if contains x xs then remdups xs else x :: remdups xs

let writer = System.IO.File.CreateText("output.txt")

(* module trans_syntax begin *)

(* General utility type for defining syntax with variables. *)
type literal<'data, 'mvar> = Inj of 'data | MVar of 'mvar

(* Basic syntax bits: node and edge literals. *)
type node_const<'thread> = NStart of 'thread | NExit of 'thread
```

```
type node_lit<'thread, 'mvar> = literal<'thread node_const, 'mvar>

(* Expression pattern EPSubst (e, e') matches "an expression e with e' somewhere in it",
   allowing basic pattern-matching within a transformation. *)
type expr_pattern<'mvar, 'expr> = EPInj of 'expr | EPSubst of 'mvar * 'expr | EPVar of '
    mvar

(* Collecting free variables. *)
let lit_fv fv x = match x with Inj data -> fv data | MVar mv -> [mv]
let type_fv x = match x with Inj _ -> [] | MVar mv -> [mv]

let expr_pattern_fv expr_fv x = match x with EPInj e -> expr_fv e | EPSubst (x, e) -> x ::
    expr_fv e | EPVar v -> [v]

(* Actions are the atomic rewrites. *)
type action<'thread, 'mvar, 'edge_type, 'pattern> =
    | AReplace of node_lit<'thread, 'mvar> * 'pattern list
    | ARemoveEdge of node_lit<'thread, 'mvar> * node_lit<'thread, 'mvar> * literal<'
        edge_type, 'mvar>
    | AAddEdge of node_lit<'thread, 'mvar> * node_lit<'thread, 'mvar> * literal<'edge_type,
        'mvar>
    | ASplitEdge of node_lit<'thread, 'mvar> * node_lit<'thread, 'mvar> * literal<'
        edge_type, 'mvar> * 'pattern

(* Side conditions are CTL formulae on program graphs. *)
type side_cond<'mvar, 'pred> =
 | SCTrue | SCPred of 'pred | SCAnd of side_cond<'mvar, 'pred> * side_cond<'mvar, 'pred> |
     SCNot of side_cond<'mvar, 'pred>
 | SCAU of side_cond<'mvar, 'pred> * side_cond<'mvar, 'pred> | SCEU of side_cond<'mvar, '
     pred> * side_cond<'mvar, 'pred>
 | SCAB of side_cond<'mvar, 'pred> * side_cond<'mvar, 'pred> | SCEB of side_cond<'mvar, '
     pred> * side_cond<'mvar, 'pred>
 | SCEx of 'mvar * System.Type * side_cond<'mvar, 'pred>

let rec cond_fv pred_fv x =
    match x with
    | SCTrue -> []
    | SCPred p -> pred_fv p
    | SCAnd ( 1 ,  2 ) -> cond_fv pred_fv  1 @ cond_fv pred_fv  2
```

```
    | SCNot    -> cond_fv pred_fv
    | SCAU (  1 ,   2 ) -> cond_fv pred_fv   1 @ cond_fv pred_fv   2
    | SCEU (  1 ,   2 ) -> cond_fv pred_fv   1 @ cond_fv pred_fv   2
    | SCAB (  1 ,   2 ) -> cond_fv pred_fv   1 @ cond_fv pred_fv   2
    | SCEB (  1 ,   2 ) -> cond_fv pred_fv   1 @ cond_fv pred_fv   2
    | SCEx (x, _,    ) -> List.filter (fun y -> y <> x) (cond_fv pred_fv   )

(* CTL abbreviations *)
let SCFalse = SCNot SCTrue
let SCOr (  1 ,   2 ) = SCNot (SCAnd (SCNot   1 , SCNot   2 ))
let SCImp (  1 ,   2 ) = SCNot (SCAnd (  1 , SCNot   2 ))
let SCEF     = SCEU (SCTrue,   )
let SCAF     = SCAU (SCTrue,   )
let SCEG     = SCNot (SCAF (SCNot   ))
let SCAG     = SCNot (SCEF (SCNot   ))
let SCAll (x, ty,    ) = SCNot (SCEx (x, ty, SCNot   ))
let SCAW (  ,   ) = SCNot (SCEU (SCNot   , (SCAnd (SCNot   , SCNot   ))))
let SCEW (  ,   ) = SCOr (SCEU (  ,   ), SCEG   )

let rec SCAnds ps =
    match ps with
    [] -> SCTrue
    | p::ps -> SCAnd(p, SCAnds ps)

let rec SCOrs ps =
    match ps with
    [] -> SCFalse
    | p::ps -> SCOr(p, SCOrs ps)

let rec SCExs vs p =
    match vs with
    [] -> p
    | (v, ty) :: rest -> SCEx (v, ty, SCExs rest p)

(* Transformations are the top-level specs. *)
(* State conditions are paired with metavars indicating the node at which they're evaluated
    . *)
type transform<'thread, 'mvar, 'edge_type, 'pattern, 'pred> =
    | TIf of action<'thread, 'mvar, 'edge_type, 'pattern> list * side_cond<'mvar, 'pred> *
        'mvar list
```

```
    | TMatch of side_cond<'mvar, 'pred> * transform<'thread, 'mvar, 'edge_type, 'pattern, '
        pred> * 'mvar list
    | TApplyAll of transform<'thread, 'mvar, 'edge_type, 'pattern, 'pred>
    | TChoice of transform<'thread, 'mvar, 'edge_type, 'pattern, 'pred> * transform<'thread
        , 'mvar, 'edge_type, 'pattern, 'pred>
    | TThen of transform<'thread, 'mvar, 'edge_type, 'pattern, 'pred> * transform<'thread,
        'mvar, 'edge_type, 'pattern, 'pred>

(* end *)

(* module trans_flowgraph begin *)

type edge<'node, 'edge_type> = 'node * 'node * 'edge_type

type flowgraph<'node, 'edge_type, 'instr when 'node : comparison and 'edge_type :
    comparison> =
  { Nodes : Set<'node>;
    Edges : Set<edge<'node, 'edge_type>>;
    Start : 'node; Exit : 'node; Label : Map<'node, 'instr> }

type tcfg<'thread, 'node, 'edge_type, 'instr when 'node : comparison and 'thread :
    comparison and 'edge_type : comparison> =
    Map<'thread, flowgraph<'node, 'edge_type, 'instr>>

let nodes cfgs = Map.fold (fun r t G -> Set.union r (G.Nodes)) Set.empty cfgs

let edges cfgs = Map.fold (fun r t G -> Set.union r (G.Edges)) Set.empty cfgs

let graph_of cfgs n = Map.tryPick (fun t G ->
    if Set.exists (fun x -> x = n) G.Nodes then Some (t, G) else None) cfgs

let size cfgs = Map.fold (fun n t G -> n * G.Nodes.Count) 1 cfgs

let out_edges cfgs n =
    match graph_of cfgs n with
    | Some (t, G) -> Set.map (fun (x, y, z) -> (y, z)) (Set.filter (fun (x, y, z) -> x = n)
         G.Edges)
    | None -> Set.empty

let next_node G t n =
```

```
    match Seq.tryFind (fun (a, b, c) -> a = n && c = t) G.Edges with Some (x, y, z) -> Some
         y | _ -> None

let in_edges cfgs n =
    match graph_of cfgs n with
    | Some (t, G) -> Set.map (fun (x, y, z) -> (x, z)) (Set.filter (fun (x, y, z) -> y = n)
         G.Edges)
    | None -> Set.empty

let start_points cfgs = Map.map (fun t G -> G.Start) cfgs

let used_vars fv cfgs = Map.fold (fun r t G -> Map.fold (fun r n i -> fv i @ r) [] G.Label
    @ r) [] cfgs

let increment (cfg:flowgraph<int, 't, 'instr>) n = {
    Nodes = Set.map (fun x -> x + n) cfg.Nodes;
    Edges = Set.map (fun (a, b, t) -> (a + n, b + n, t)) cfg.Edges;
    Start = cfg.Start + n;
    Exit = cfg.Exit + n;
    Label = new Map<int, 'instr>(seq {for KeyValue(k, v) in cfg.Label -> (k + n, v)}) }

(* end *)

(* module ctl_check begin *)

let _opts = new Dictionary<string, string>()
_opts.Add("MODEL","true")
let context = new Context(_opts)
let both = context.MkAnd()
let solver = context.MkSolver()
ignore(solver.Check())
let empty_model = solver.Model
solver.Reset()

let mkvar sort (name:string) = context.MkConst(name, sort)

let rec cross (m:Map<string, Set<int>>) = Map.fold (fun r t l -> conmap (fun s -> List.map
    (fun m -> Map.add t s m) r) l) [(new Map<string, int>(Seq.empty))] m
```

```
(* Generics by reflection. By inspecting the instruction type, we can build Z3 datatypes
    and
   convert instructions and patterns into them. *)
let mutable type_table:Dictionary<System.Type, Sort> = null

(* Make a Z3 sort corresponding to a datatype. *)
(* Works only for non-recursive or directly recursive union types (no mutual recursion). *)
let rec mk_z3_sort (t:System.Type) =
    if t.Name = "literal`2" then mk_z3_sort (t.GenericTypeArguments.[0])
    elif t.Name = "expr_pattern`2" then mk_z3_sort (t.GenericTypeArguments.[1])
    elif type_table.ContainsKey(t) then type_table.[t]
    else let ty = context.MkDatatypeSort(t.Name,[|for c in FSharpType.GetUnionCases(t) ->
        context.MkConstructor(c.Name, "is_" + c.Name,
        [|for f in c.GetFields() -> f.Name|],
        [|for f in c.GetFields() -> let t' = f.PropertyType in if t = t' then null else
            mk_z3_sort t'|],
        [|for f in c.GetFields() -> 0u|])|])
         type_table.[t] <- ty
         ty :> Sort

(* Convert a program pattern to a Z3 datatype, assuming strings represent metavariables. *)
let rec convert_p p =
    let t = p.GetType()
    if t = typeof<string> then mkvar (mk_z3_sort typeof<string>) (p.ToString())
    elif t = typeof<int> then context.MkInt(p.ToString()) :> Expr
    else
        let c, args = FSharpValue.GetUnionFields(p, t)
        if c.DeclaringType.Name = "literal`2" then
            if c.Name = "MVar" then mkvar (mk_z3_sort (t.GenericTypeArguments.[0])) (args
                .[0].ToString()) else convert_p args.[0]
        elif c.DeclaringType.Name = "expr_pattern`2" then
            if c.Name = "EPInj" then convert_p (args.[0])
            elif c.Name = "EPVar" then mkvar (mk_z3_sort (t.GenericTypeArguments.[1])) (
                args.[0].ToString())
            else null (* need code for EPSubst - match a function? *)
        else
            let ty = mk_z3_sort (c.DeclaringType) :?> DatatypeSort
            let i = Array.findIndex (fun c' -> c' = c) (FSharpType.GetUnionCases t)
            context.MkApp(ty.Constructors.[i], [|for x in args -> convert_p x|])
```

```
let get_const (enum_sort:EnumSort) name =
    context.MkApp(Array.find (fun (e:FuncDecl) -> e.Name.ToString() = name) enum_sort.
        ConstDecls)

(* Convert a program object to a Z3 datatype, assuming strings represent concrete variables
    . *)
let rec convert_i p =
    let t = p.GetType()
    if t = typeof<string> then get_const (mk_z3_sort typeof<string> :?> EnumSort) (p.
        ToString())
    elif t = typeof<int> then context.MkInt(p.ToString()) :> Expr
    else
        let c, args = FSharpValue.GetUnionFields(p, t)
        let ty = mk_z3_sort (c.DeclaringType) :?> DatatypeSort
        let i = Array.findIndex (fun c' -> c' = c) (FSharpType.GetUnionCases t)
        context.MkApp(ty.Constructors.[i], [|for x in args -> convert_i x|])

(* Recover an F# datatype from a Z3 datatype. *)
let rec strip (t:System.Type) =
    if t.Name = "literal`2" then strip (t.GenericTypeArguments.[0])
    elif t.Name = "expr_pattern`2" then strip (t.GenericTypeArguments.[1])
    elif t.IsGenericType then t.GetGenericTypeDefinition().MakeGenericType([|for p in t.
        GenericTypeArguments -> strip p|])
    else t

let rec recover_i (e:Expr) =
    let ty = (Seq.head (Seq.filter (fun (KeyValue(_, s)) -> s = e.Sort) type_table)).Key
    let t = strip ty
    if t = typeof<string> then e.FuncDecl.Name.ToString() :> obj
    elif t = typeof<int> then (e :?> IntNum).Int :> obj
    else
        let c = Array.find (fun (x:UnionCaseInfo) -> x.Name = e.FuncDecl.Name.ToString()) (
            FSharpType.GetUnionCases(t))
        FSharpValue.MakeUnion(c, [|for a in e.Args -> recover_i a|])

(* Offload substitution to Z3. *)
let subst (s:Model) p = recover_i(s.Eval(convert_p p))

(* Find strings in recursive datatypes. *)
let rec freevars (p:obj) =
```

```
    let t = p.GetType()
    if t = typeof<string> then [p :?> string]
    elif FSharpType.IsUnion(t) then Array.fold (fun r o -> r @ freevars o) [] (snd (
        FSharpValue.GetUnionFields(p, t)))
    else []

(* Memoization helps a lot. *)
let rec paths_forward caches mk_pred thread_sort cfgs p n m =
    let (_, cache:Dictionary<_,_>, _, _, _, _) = caches
    if cache.ContainsKey((p, n, m)) then cache.[(p, n, m)]
    else
    let res =
        if n <= 0 then context.MkBool(true)
        else let nexts = List.filter (fun m' -> m' <> m) (cross (Map.map (fun t s -> Set.
            add s (Set.map fst (out_edges cfgs s))) m))
             context.MkOr(List.toArray (List.map (fun m' -> context.MkAnd(satis caches
                 mk_pred thread_sort cfgs m' p,
                    paths_forward caches mk_pred thread_sort cfgs p (n - 1) m')) nexts))
    cache.[(p, n, m)] <- res
    res
and paths_backward caches mk_pred thread_sort cfgs p0 p n m =
    let (_, _, cache:Dictionary<_,_>, _, _, _) = caches
    if cache.ContainsKey((p0, p, n, m)) then cache.[(p0, p, n, m)]
    else
    let res =
        if n <= 0 then satis caches mk_pred thread_sort cfgs m p0 else context.MkOr(
            paths_backward caches mk_pred thread_sort cfgs p0 p (n - 1) m, context.MkAnd(
            satis caches mk_pred thread_sort cfgs m p,
            context.MkOr(List.toArray (List.map (fun m' -> paths_backward caches mk_pred
                thread_sort cfgs p0 p (n - 1) m') (List.filter (fun m' -> m' <> m) (cross (
                Map.map (fun t s -> Set.add s (Set.map fst (out_edges cfgs s))) m)))))))
    cache.[(p0, p, n, m)] <- res
    res
and rpaths_and caches mk_pred thread_sort cfgs p0 p n m =
    let (_, _, _, _, _, cache:Dictionary<_,_>) = caches
    if cache.ContainsKey((p0, p, n, m)) then cache.[(p0, p, n, m)]
    else
    let res =
        if n <= 0 then satis caches mk_pred thread_sort cfgs m p0
```

```
        else let prevs = List.filter (fun m' -> m' <> m) (cross (Map.map (fun t s -> Set.
            add s (Set.map fst (in_edges cfgs s))) m))
             context.MkOr(rpaths_and caches mk_pred thread_sort cfgs p0 p (n - 1) m,
                 context.MkAnd(satis caches mk_pred thread_sort cfgs m p,
                if prevs = [] then satis caches mk_pred thread_sort cfgs m p0 else context.
                    MkAnd(List.toArray (List.map (fun m' -> rpaths_and caches mk_pred
                    thread_sort cfgs p0 p (n - 1) m') prevs))))
    cache.[(p0, p, n, m)] <- res
    res
and rpaths_forward2 caches mk_pred thread_sort cfgs p0 p n m =
    let (_, _, _, cache:Dictionary<_,_>, _, _) = caches
    if cache.ContainsKey((p0, p, n, m)) then cache.[(p0, p, n, m)]
    else
    let res =
        if n <= 0 then satis caches mk_pred thread_sort cfgs m p0 else context.MkOr(
            rpaths_forward2 caches mk_pred thread_sort cfgs p0 p (n - 1) m, context.MkAnd(
            satis caches mk_pred thread_sort cfgs m p,
            context.MkOr(List.toArray (List.map (fun m' -> rpaths_forward2 caches mk_pred
                thread_sort cfgs p0 p (n - 1) m') (List.filter (fun m' -> m' <> m) (cross (
                Map.map (fun t s -> Set.add s (Set.map fst (in_edges cfgs s))) m)))))))
    cache.[(p0, p, n, m)] <- res
    res
and paths_and caches mk_pred thread_sort cfgs p0 p n m =
    let (_, _, _, _, cache:Dictionary<_,_>, _) = caches
    if cache.ContainsKey((p0, p, n, m)) then cache.[(p0, p, n, m)]
    else
    let res =
        if n <= 0 then satis caches mk_pred thread_sort cfgs m p0
        else let nexts = List.filter (fun m' -> m' <> m) (cross (Map.map (fun t s -> Set.
            add s (Set.map fst (out_edges cfgs s))) m))
             context.MkOr(paths_and caches mk_pred thread_sort cfgs p0 p (n - 1) m, context
                 .MkAnd(satis caches mk_pred thread_sort cfgs m p,
                if nexts = [] then satis caches mk_pred thread_sort cfgs m p0 else context.
                    MkAnd(List.toArray (List.map (fun m' -> paths_and caches mk_pred
                    thread_sort cfgs p0 p (n - 1) m') nexts))))
    cache.[(p0, p, n, m)] <- res
    res
and satis caches mk_pred thread_sort cfgs m p =
    let (cache:Dictionary<_,_>, _, _, _, _, _) = caches
    if cache.ContainsKey((m, p)) then cache.[(m, p)]
```

```
    else
    let res =
        match p with
        | SCPred p -> let (e:BoolExpr) = mk_pred cfgs thread_sort m p in e.Simplify() :?>
            BoolExpr (* unnecessary and might slow it down, but allows for cleaner debug
            output *)
        | SCTrue -> context.MkBool(true)
        | SCNot p -> context.MkNot(satis caches mk_pred thread_sort cfgs m p)
        | SCAnd (p1, p2) -> context.MkAnd(satis caches mk_pred thread_sort cfgs m p1, satis
             caches mk_pred thread_sort cfgs m p2)
        | SCEx (x:string, ty, p) -> context.MkExists([|context.MkConst(x, mk_z3_sort ty)|],
             satis caches mk_pred thread_sort cfgs m p) :> BoolExpr (* need a sort for x *)
        | SCAU (p1, p2) -> context.MkAnd(context.MkNot(paths_forward caches mk_pred
            thread_sort cfgs (SCAnd (p1, SCNot p2)) (size cfgs + 1) m),
                                     paths_and caches mk_pred thread_sort cfgs p2 p1 (
                                         size cfgs) m)
        | SCEU (p1, p2) -> paths_backward caches mk_pred thread_sort cfgs p2 p1 (size cfgs)
             m
        | SCAB (p1, p2) -> rpaths_and caches mk_pred thread_sort cfgs p2 p1 (size cfgs) m
        | SCEB (p1, p2) -> rpaths_forward2 caches mk_pred thread_sort cfgs p2 p1 (size cfgs
            ) m
    cache.[(m, p)] <- res
    res

let mk_exprs (m:Model) = [|for decl in m.ConstDecls -> context.MkEq(context.MkConst(decl),
    m.ConstInterp(decl))|]

let mk_conds (m:Model) vars =
    [|for decl in Array.filter (fun (d:FuncDecl) -> contains (d.Name.ToString()) vars) m.
        ConstDecls ->
      context.MkEq(context.MkConst(decl), m.ConstInterp(decl))|]

(* memoization courtesy Don Syme http://blogs.msdn.com/b/dsyme/archive/2007/05/31/a-sample-
    of-the-memoization-pattern-in-f.aspx *)
let memo_satis mk_pred thread_sort cfgs m p =
    let scache = Dictionary<_,_>()
    let pfcache = Dictionary<_,_>()
    let pbcache = Dictionary<_,_>()
    let rfcache = Dictionary<_,_>()
    let pacache = Dictionary<_,_>()
```

```
    let racache = Dictionary<_,_>()
    satis (scache, pfcache, pbcache, rfcache, pacache, racache) mk_pred thread_sort cfgs m
        p

let rec get_all_models vars n acc = if n <= 0 then acc else (
    printfn "Invoking Z3..."
    stdout.Flush()
    let response = solver.Check()
    printfn "Z3 returned %s" (response.ToString())
    stdout.Flush()
    if response = Status.SATISFIABLE
    then
        fprintfn writer "\n%A\n" solver.Model
        solver.Assert(context.MkNot(context.MkAnd(mk_conds solver.Model vars)))
        get_all_models vars (n - 1) (solver.Model :: acc)
    else acc)

let keys m = Map.fold (fun r k v -> k :: r) [] m

let check_enum (sort:EnumSort) (var:string) name =
    let tester = Array.find (fun (t:FuncDecl) -> t.Name.ToString() = "is_" + name) sort.
        TesterDecls
    context.MkApp(tester, mkvar sort var) :?> BoolExpr

(* make fresh variables *)
let rec next_fresh vs n = if List.exists (fun v -> "_fresh" + n.ToString() = v) vs then
    next_fresh vs (n + 1) else n
let rec make_fresh_aux names (map, used) n =
    match names with
    | [] -> (map, used)
    | f::rest -> let n' = next_fresh used n in let v' = "_fresh" + n'.ToString() in
        make_fresh_aux rest (Map.add f v' map, v'::used) (n' + 1)
let make_fresh used names = make_fresh_aux names (new Map<string, string>([]), used) 0

(* The main interface to the SMT solver. *)
let get_models mk_pred pred_fv instr_fv fresh tau cfgs p =
    let thread_sort = context.MkEnumSort("thread", List.toArray (keys cfgs))
    let (fresh_map, vars) = make_fresh (used_vars instr_fv cfgs) fresh
    let var_sort = context.MkEnumSort("var", if vars <> [] then List.toArray vars else [|"
        dud"|])
```

```
    type_table <- Dictionary<System.Type, Sort>()
    type_table.[typeof<string>] <- var_sort
    type_table.[typeof<int>] <- context.MkIntSort()
    solver.Reset()
    solver.Assert([|for KeyValue(k, v) in fresh_map -> check_enum var_sort k v|])
    solver.Assert(mk_exprs tau)
    solver.Assert(memo_satis mk_pred thread_sort cfgs (start_points cfgs) p)
    fprintfn writer "%A" (solver.Assertions)
    fprintfn writer "%A" (solver.Assertions.[0].Simplify())
    writer.Flush()
    get_all_models (cond_fv pred_fv p) 200 [] (* set the maximum number of models returned
        *)

(* end *)

(* module trans_semantics begin *)

let node_subst cfgs (s:Model) (n:node_lit<string, string>) =
    match n with
    | Inj (NStart t) -> (match Map.tryFind t cfgs with Some G -> Some G.Start | None ->
        None)
    | Inj (NExit t) -> (match Map.tryFind t cfgs with Some G -> Some G.Exit | None -> None)
    | MVar v -> Some ((s.ConstInterp(context.MkIntConst(v)) :?> IntNum).Int)

let rec update_map m l =
    match l with
    | [] -> m
    | ((x, y) :: rest) -> update_map (Map.add x y m) rest

let remap_succ x y (n, s, t) = if x = n then (y, s, t) else (n, s, t)

let rep_edges seq es ll =
    if ll = [] then es
    else Set.union (Set.map (fun e -> remap_succ (ll.Head) ((List.rev ll).Head) e) es) (Set
        .ofList [for a in 0 .. ll.Length - 2 -> (ll.Item(a), ll.Item(a + 1), seq)])

(* This forces nodes to be ints. We could parameterize for generality. *)
let fresh_nodes l n = [Set.maxElement l + 1 .. Set.maxElement l + n]
```

```
let subst_list subst s pl = List.fold (fun r p -> match (r, subst s p) with (Some l, Some i
    ) -> Some (l @ [i]) | _ -> None) (Some []) pl

let action_sf type_subst subst seq A s cfgs =
    match A with
    | AAddEdge (n, m, e) ->
        match (node_subst cfgs s n, node_subst cfgs s m, type_subst s e) with
        | (Some u, Some v, Some ty) ->
            if not (Set.exists (fun x -> x = u) (nodes cfgs)) then None
            else match graph_of cfgs u with
                 | Some (t, G) -> (if Set.exists (fun x -> x = v) G.Nodes
                                   then Some (Map.add t {G with Edges = Set.add (u, v, ty)
                                       G.Edges} cfgs)
                                   else None)
                 | _ -> None
        | _ -> None
    | ARemoveEdge (n, m, e) ->
        match (node_subst cfgs s n, node_subst cfgs s m, type_subst s e) with
        | (Some u, Some v, Some ty) ->
            if not (Set.exists (fun x -> x = u) (nodes cfgs)) then None
            else match graph_of cfgs u with
                 | Some (t, G) -> (if Set.exists (fun x -> x = v) G.Nodes
                                   then Some (Map.add t {G with Edges = Set.filter (fun e
                                       -> e <> (u, v, ty)) G.Edges} cfgs) else None)
                 | _ -> None
        | _ -> None
    | AReplace (m, pl) ->
       (match node_subst cfgs s m with
        | None -> None
        | Some l ->
           (match graph_of cfgs l with
            | None -> None
            | Some (t, G) -> if pl = [] then (* remove node *)
               (if (l = G.Start || l = G.Exit) then None (* Should we be able to remove
                   Start and/or Exit? *)
                (* What should we do with the in- and out-edges of a removed node? *)
                else Some (Map.add t {G with Nodes = Set.filter (fun x -> x <> l) G.Nodes;
                                             Edges = Set.filter (fun (u, v, t) -> u <> l &&
                                                  v <> l) G.Edges;
```

```
                                          Label = Map.filter (fun k v -> k <> l) (G.
                                             Label)} cfgs))
                              else
                (match subst_list subst s pl with
                 | None -> None
                 | Some il -> let ll = fresh_nodes (nodes cfgs) (List.length il - 1)
                              in Some (Map.add t {G with Nodes = Set.union (G.Nodes) (set ll
                                  ); Edges = rep_edges seq G.Edges (l :: ll);
                                                        Label = update_map G.Label (List.
                                                            zip (l :: ll) il)} cfgs))))
    | ASplitEdge (n, m, e, i) ->
        match (node_subst cfgs s n, node_subst cfgs s m, type_subst s e, subst s i) with
        | (Some u, Some v, Some ty, Some j) ->
            if Set.exists (fun x -> x = (u, v, ty)) (edges cfgs)
            then match graph_of cfgs u with
                 | Some (t, G) ->
                    let q = (fresh_nodes (nodes cfgs) 1).Head
                    in Some (Map.add t {G with Nodes = Set.add q G.Nodes; Edges = Set.union
                         (Set.filter (fun x -> x <> (u, v, ty)) G.Edges) (set [(u, q, ty); (
                        q, v, seq)]);
                                                  Label = Map.add q j G.Label} cfgs)
                 | _ -> None
            else None
        | _ -> None

let rec action_list_sf type_subst subst seq al s cfgs =
    match al with
    | [] -> Some cfgs
    | A :: rest ->
        match action_sf type_subst subst seq A s cfgs with None -> None | Some cfgs' ->
            action_list_sf type_subst subst seq rest s cfgs'

(* Take in one tCFG, produce a list. *)
let rec trans_sf mk_pred pred_fv instr_fv type_subst subst seq T tau cfgs =
    match T with
    | TIf (al, p, fresh) -> let sl = get_models mk_pred pred_fv instr_fv fresh tau cfgs p
        in
                             remdups (List.fold (fun r s -> match action_list_sf type_subst
                                 subst seq al s cfgs with Some cfgs' -> cfgs' :: r | None ->
                                 r) [] sl)
```

```
    | TMatch (p, T, fresh) -> remdups (conmap (fun s -> trans_sf mk_pred pred_fv instr_fv
        type_subst subst seq T s cfgs) (get_models mk_pred pred_fv instr_fv fresh tau cfgs p
        ))
    | TApplyAll T -> let results = remdups (trans_sf mk_pred pred_fv instr_fv type_subst
        subst seq T tau cfgs)
                     in if results = [] || List.forall (fun cfgs' -> cfgs' = cfgs) results
                         then [cfgs]
                        else remdups (conmap (trans_sf mk_pred pred_fv instr_fv type_subst
                            subst seq (TApplyAll T) tau) results)
    | TChoice (T1, T2) -> remdups (trans_sf mk_pred pred_fv instr_fv type_subst subst seq
        T1 tau cfgs @ trans_sf mk_pred pred_fv instr_fv type_subst subst seq T2 tau cfgs)
    | TThen (T1, T2) -> let r = (trans_sf mk_pred pred_fv instr_fv type_subst subst seq T1
        tau cfgs)
                        remdups (conmap (fun cfgs' -> trans_sf mk_pred pred_fv instr_fv
                            type_subst subst seq T2 tau cfgs') r)

(* very simple sample language *)
(* Instruction patterns and substitution. *)
type edge_type = Seq | Branch

type instr = Skip | Assign of string * string

let pattern_fv i =
    match i with
    | Skip -> []
    | Assign (x, y) -> [x; y]

let simple_subst (s:Model) p =
    match p with
    | Skip -> Some Skip
    | Assign (x, y) ->
        let var_sort = type_table.[typeof<string>]
        Some (Assign (s.ConstInterp(context.MkFuncDecl(x, [||], var_sort)).ToString(), s.
            ConstInterp(context.MkFuncDecl(y, [||], var_sort)).ToString()))

(* atomic predicates *)
type int_expr = Intv of string | Intc of int | Plus of int_expr * int_expr | Times of
    int_expr * int_expr
let rec int_expr_to_z3 i =
    match i with
```

```
    | Intv v -> context.MkIntConst(v) :> ArithExpr
    | Intc c -> context.MkInt(c) :> ArithExpr
    | Plus (i, j) -> context.MkAdd(int_expr_to_z3 i, int_expr_to_z3 j)
    | Times (i, j) -> context.MkMul(int_expr_to_z3 i, int_expr_to_z3 j)

type simple_pred<'edge_type, 'instr> =
    | Node of string * string | Stmt of string * 'instr
    | Out of string * string * literal<'edge_type, string> | Start | Exit
    | Is of int_expr * int_expr | IsNot of int_expr * int_expr



let node t n = SCPred (Node (t, n))
let stmt t i = SCPred (Stmt (t, i))
let out t n ty = SCPred (Out (t, n, ty))

(* The generic approach. *)
let add_stmt_case cfgs thread_sort t p e t' s =
    match Map.tryFind s (Map.find t' cfgs).Label with
    | Some i -> context.MkOr(e, context.MkAnd(check_enum thread_sort t t', context.MkEq(
        convert_p p, convert_i i)))
    | None -> e

let mk_simple_pred2 cfgs thread_sort (m:Map<string, int>) p =
    match p with
    | Node (t, n) -> Map.fold (fun e (t':string) (s:int) -> context.MkOr(e, context.MkAnd(
        check_enum thread_sort t t', context.MkEq(context.MkIntConst(n), context.MkInt(s))))
        ) (context.MkBool(false)) m
    | Stmt (t, p) -> Map.fold (add_stmt_case cfgs thread_sort t p) (context.MkBool(false))
        m
    | Out (t, n, ty) -> Map.fold (fun e (t':string) (s:int) ->
        Set.fold (fun e ((m:int), t) -> context.MkOr(e, context.MkAnd(context.MkEq(context.
            MkIntConst(n), context.MkInt(m)), context.MkEq(convert_p ty, convert_i t)))) e (
            out_edges cfgs s)) (context.MkBool(false)) m
    | Start -> context.MkBool(Map.forall (fun t s -> s = (Map.find t cfgs).Start) m)
    | Exit -> context.MkBool(Map.forall (fun t s -> s = (Map.find t cfgs).Exit) m)
    | Is (i, j) -> context.MkEq(int_expr_to_z3 i, int_expr_to_z3 j)
    | IsNot (i, j) -> context.MkNot(context.MkEq(int_expr_to_z3 i, int_expr_to_z3 j))

let type_subst s (x:'a) =
```

```
    let r = subst s x
    if r.GetType() = typeof<'a> then Some (r :?> 'a) else None

let set l = Set.ofList l

let simple_cfg1 = { Nodes = set [1; 2; 3]; Edges = set [(1, 2, Seq); (2, 3, Seq)]; Start =
    1; Exit = 3;
                    Label = new Map<int, instr>(List.toSeq [(1, Skip); (2, Skip); (3, Skip)
                        ]) }
let simple_cfg2 = { Nodes = set [4; 5; 6; 7]; Edges = set [(4, 5, Seq); (5, 6, Seq); (5, 7,
     Branch); (6, 7, Seq)]; Start = 4; Exit = 7;
                    Label = new Map<int, instr>(List.toSeq [(4, Skip); (5, Assign ("x", "y
                        ")); (6, Assign ("a", "b")); (7, Skip)]) }
let simple_tcfg = new Map<string, flowgraph<int, edge_type, instr>>(List.toSeq [("t1",
    simple_cfg1); ("t2", simple_cfg2)])

let simple_trans : transform<string, string, edge_type, instr, simple_pred<edge_type, instr
    >> =
    TIf ([AReplace (MVar "n", [Skip; Skip])], SCEF (SCAnd (SCPred (Node ("t", "n")), SCPred
         (Stmt ("t", Assign ("c", "d"))))), [])

(* end *)

(* Some defined predicates *)
let SCEX t n p = SCAnd (node t n, SCEU (node t n, SCAnd (SCNot (node t n), p)))
let SCAX t n p = SCAnd (node t n, SCAW (node t n, SCAnd (SCNot (node t n), p)))
let SCEP t n p = SCAnd (node t n, SCEB (node t n, SCAnd (SCNot (node t n), p)))
let SCAP t n p = SCAnd (node t n, SCAB (node t n, SCAnd (SCNot (node t n), p)))
let SCEX_t t n ty p = SCEx ("_n", typeof<int>, SCAnd (out t "_n" ty, SCEX t n (SCAnd (node
    t "_n", p))))

(* Finding loops *)
let dom t n m = SCNot (SCEU (SCNot (node t n), node t m))
let pdom t n m = SCEF (SCAnd (SCPred Exit, SCNot (SCEB (SCNot (node t n), node t m))))
let loop t phead head brk tail = SCAnds [dom t phead head; pdom t brk tail; dom t head tail
    ; SCEF (SCAX t phead (node t head)); SCEF (SCEX t tail (node t head))]
let out_loop t phead brk = SCAU (SCNot (node t brk), SCOr (node t phead, SCPred Exit))
let out_jump t phead brk = SCEx ("_n", typeof<int>, SCAnd (SCNot(node t brk), SCEX t "_n" (
    out_loop t phead brk)))
```

```
let in_jump t phead head brk = SCEx ("_n", typeof<int>, SCAnd (SCNot(node t head), SCEP t "
    _n" (out_loop t phead brk)))
let wsloop t phead head brk tail = SCAnd (loop t phead head brk tail,
    SCEF (SCAnds [node t head; SCAU (SCNot (out_jump t phead brk), out_loop t phead brk);
                  SCAU (SCNot (in_jump t phead head brk), out_loop t phead brk)]))

module MiniLLVM

open Microsoft.Z3
open Ptrans

type LLVM_type = Int_ty | Pointer_ty of LLVM_type
type LLVM_expr<'var, 'con> = Local of 'var | CInt of 'con | CPointer of 'con | CUndef |
    Global of string
type LLVM_op = Add | Sub | Mul
type LLVM_cmp = Eq | Ne | Sgt | Sge | Slt | Sle

type LLVM_instr<'var, 'ty, 'expr, 'opr, 'cmp> =
 | Assign of 'var * 'opr * 'ty * 'expr * 'expr | ICmp of 'var * 'cmp * 'ty * 'expr * 'expr
 | Br_i1 of 'expr (* conditional branch *) | Br_label (* unconditional *)
 | Alloca of 'var * 'ty | Load of 'var * 'ty * 'expr | Store of 'ty * 'expr * 'ty * 'expr
 | Cmpxchg of 'var * 'ty * 'expr * 'ty * 'expr * 'ty * 'expr
(* | Call of 'var * 'ty * list<'expr> | Ret of 'ty * 'expr*) | IsPointer of 'expr

type LLVM_edge_type = Seq | True | False

type pattern<'mvar> = literal<LLVM_instr<'mvar, literal<LLVM_type, 'mvar>, expr_pattern<'
    mvar, LLVM_expr<'mvar, literal<int, 'mvar>>>,
     literal<LLVM_op, 'mvar>, literal<LLVM_cmp, 'mvar>>, 'mvar>

(* Z3 does pattern-matching and substitution, and free vars are calculated by reflection,
   so we're ready to test with no prep! *)
type LLVM_const = LLVM_expr<string, int>
type c_instr = LLVM_instr<string, LLVM_type, LLVM_const, LLVM_op, LLVM_cmp>

let llvm_cfg1 = { Nodes = set [1; 2; 3]; Edges = set [(1, 2, Seq); (2, 3, Seq)]; Start = 1;
     Exit = 3;
                    Label = new Map<int, c_instr>(List.toSeq [(1, Br_label);
                        (2, Br_label); (3, Br_label)]) }
let llvm_cfg2 = { Nodes = set [4; 5; 6; 7];
Edges = set [(4, 5, Seq); (5, 6, Seq); (5, 7, True); (6, 7, Seq)];
```

```
Start = 4; Exit = 7;
Label = new Map<int, c_instr>(List.toSeq [(4, Br_label);
(5, Alloca ("x", Int_ty)); (6, Br_label); (7, Br_label)]) }

let llvm_tcfg = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("t1",
     llvm_cfg1); ("t2", llvm_cfg2)])

type trans = transform<string, string, LLVM_edge_type, pattern<string>, simple_pred<
    LLVM_edge_type, pattern<string>>>
let llvm_trans : trans =
    TIf ([AReplace (MVar "n", [Inj Br_label; Inj Br_label])], SCAF (SCAnd (
    SCPred (Node ("t", "n")), SCPred (Stmt ("t", Inj (Alloca ("c", MVar "d")))))), [])

let test_trans T cfgs =
    let time = System.Diagnostics.Stopwatch.StartNew()
    let result = trans_sf mk_simple_pred2 freevars freevars (fun s p -> Some (subst s p :?>
         LLVM_edge_type)) (fun s p -> Some (subst s p :?> c_instr)) Seq T empty_model cfgs
    time.Stop()
    printfn "%i" time.ElapsedMilliseconds
    printfn "%A" result
    fprintfn writer "%A" result
    writer.Flush()
    result

(* semantics *)

(* memory models  *)
type access<'thread> = Read of 'thread * int * LLVM_expr<string, int> | Write of 'thread *
    int * LLVM_expr<string, int>
                     | ARW of 'thread * int * LLVM_expr<string, int> * LLVM_expr<string,
                        int> | Alloc of 'thread * int | Free of 'thread * int
let get_thread a =
    match a with
    | Read (t, _, _) -> t
    | Write (t, _, _) -> t
    | ARW (t, _, _, _) -> t
    | Alloc (t, _) -> t
    | Free (t, _) -> t

let get_loc a =
```

```
    match a with
    | Read (_, l, _) -> l
    | Write (_, l, _) -> l
    | ARW (_, l, _, _) -> l
    | Alloc (_, l) -> l
    | Free (_, l) -> l

(* SC *)
let lookup l m = match Map.tryFind l m with Some r -> r | None -> CUndef

let rec first_free_aux l m =
    match l with
    | [] -> m
    | n :: rest -> if n > m then m else if n = m then first_free_aux rest (m + 1) else
        first_free_aux rest m
let first_free l = first_free_aux l 0
let free_loc_SC m = List.sort((List.map fst (Map.toList m))) |> first_free

let read_SC m t l = lookup l m

let do_ops_SC m a =
    match a with
    | Read (_, _, _) -> m
    | Write (_, l, v) -> Map.add l v m
    | Alloc (_, l) -> Map.add l CUndef m
    | Free (_, l) -> Map.remove l m

let update_mem_SC m ops =
    match List.tryFind (fun a -> match a with ARW (_,_,_,_) -> true | _ -> false) ops with
    | Some (ARW (t, l, v, v')) -> [Map.add l v' m] (* discard all other operations *)
    | None -> (* order is important *) [List.fold do_ops_SC m ops]

(* TSO *)
let free_loc_TSO (m, b) = List.sort((List.map fst (Map.toList m))) |> first_free

let read_TSO (m, b) t l =
    match List.tryFind (fun (l', v) -> l' = l) (Map.find t b) with
    | Some (_, v) -> v
    | None -> lookup l m
```

```
let rec do_bufs (m, b) =
    (m, b) :: conmap do_bufs (conmap (fun (t, q) -> match q with [] -> [] | (l, v) :: r ->
        [(Map.add l v m, Map.add t r b)]) (Map.toList b))

let clear_bufs m = List.filter (fun (m, b) -> Map.forall (fun _ l -> l = []) b) (do_bufs m)

let do_ops_TSO (m, b) a =
    match a with
    | Read (_, _, _) -> (m, b)
    | Write (t, l, v) -> (m, Map.add t (Map.find t b @ [(l, v)]) b)
    | Alloc (_, l) -> (Map.add l CUndef m, b)
    | Free (_, l) -> (Map.remove l m, b)

let update_mem_TSO m ops =
    match List.tryFind (fun a -> match a with ARW (_,_,_,_) -> true | _ -> false) ops with
    | Some (ARW (t, l, v, v')) -> List.map (fun (m, b) -> (Map.add l v' m, b)) (clear_bufs
        m) (* discard all other operations *)
    | None -> (* order is important *) conmap do_bufs (List.map (fun m -> List.fold
        do_ops_TSO m ops) (do_bufs m))

(* expressions *)
let eval_expr env gt e =
    match e with
    | Local i -> lookup i env
    | Global i -> lookup i gt
    | _ -> e

let cmp_helper opr v1 v2 =
    match opr with
    | Eq -> v1 = v2
    | Ne -> v1 <> v2
    | Sgt -> v1 > v2
    | Sge -> v1 >= v2
    | Slt -> v1 < v2
    | Sle -> v1 <= v2

let eval_cmp env gt cmp e1 e2 =
    match (eval_expr env gt e1, eval_expr env gt e2) with
    | (CInt v1, CInt v2) -> if cmp_helper cmp v1 v2 then CInt 1 else CInt 0
    | (CPointer v1, CPointer v2) -> if cmp_helper cmp v1 v2 then CInt 1 else CInt 0
```

```
    | _ -> CUndef

let eval env gt opr e1 e2 =
    match (eval_expr env gt e1, eval_expr gt env e2) with
    | (CInt v1, CInt v2) -> (match opr with Add -> CInt (v1 + v2) | Sub -> CInt (v1 - v2) |
         Mul -> CInt (v1 * v2))
    | (CPointer v1, CPointer v2) -> (match opr with Add -> CPointer (v1 + v2) | Sub ->
        CPointer (v1 - v2) | Mul -> CPointer (v1 * v2))
    | _ -> CUndef

let init_env env gt args =
    fst (List.fold (fun (e, n) a -> (Map.add ("arg" + n.ToString()) (eval_expr env gt a) e,
         n + 1)) (new Map<string, LLVM_expr<_,int>>([]), 0) args)

type state<'node> = 'node * Map<string, LLVM_expr<string,int>> * ('node * string * Map<
    string, LLVM_expr<string,int>> * int list) list * int list

let LLVM_step free_loc read G t mem gt (p, env, stack, allocad) =
    match G.Label.TryFind(p) with
    | None -> None
    | Some (Assign (x, opr, ty, e1, e2)) ->
        match next_node G Seq p with Some p' -> Some ((p', Map.add x (eval env gt opr e1 e2
            ) env, stack, allocad), []) | None -> None
    | Some (ICmp (x, cmp, ty, e1, e2)) ->
        match next_node G Seq p with Some p' -> Some ((p', Map.add x (eval_cmp env gt cmp
            e1 e2) env, stack, allocad), []) | None -> None
    | Some (Br_i1 e) ->
        match next_node G (if eval_expr env gt e = CInt 0 then False else True) p with Some
             p' -> Some ((p', env, stack, allocad), []) | None -> None
    | Some Br_label ->
        match next_node G Seq p with Some p' -> Some ((p', env, stack, allocad), []) | None
             -> None
    | Some (Alloca (x, ty)) ->
        let new_loc = free_loc mem in
        match next_node G Seq p with Some p' -> Some ((p', Map.add x (CPointer new_loc) env
            , stack, new_loc :: allocad), [Alloc (t, new_loc)]) | None -> None
    | Some (Load (x, ty, e)) ->
        match (eval_expr env gt e, next_node G Seq p) with
        | (CPointer l, Some p') -> let v = read mem t l in Some ((p', Map.add x v env,
            stack, allocad), [Read (t, l, v)])
```

```
        | _ -> None
    | Some (Store (ty1, e1, ty2, e2)) ->
        match (eval_expr env gt e2, next_node G Seq p) with
        | (CPointer l, Some p') -> Some ((p', env, stack, allocad), [Write (t, l, (
            eval_expr env gt e1))])
        | _ -> None
    | Some (Cmpxchg (x, ty1, e1, ty2, e2, ty3, e3)) ->
        match (eval_expr env gt e1, next_node G Seq p) with
        | (CPointer l, Some p') -> let v = read mem t l in Some ((p', Map.add x v env,
            stack, allocad),
            [ARW (t, l, v, if eval_expr env gt e2 = v then eval_expr env gt e3 else v)])
        | _ -> None
    | Some (IsPointer e) ->
        match (eval_expr env gt e, next_node G Seq p) with
        | (CPointer l, Some p') -> Some ((p', env, stack, allocad), [])
        | _ -> None

let conc_step free_loc read update_mem cfgs gt (states, mem) =
    conmap (fun (t, G) ->
        let s = Map.find t states in
        match LLVM_step free_loc read G t mem gt s with
        | Some (s', ops) -> List.map (fun mem' -> (Map.add t s' states, mem')) (update_mem
            mem ops)
        | None -> []) (Map.toList cfgs)

let conc_step_star free_loc read update_mem cfgs gt (states, mem) =
    let rec conc_step_star_aux l =
        let res = remdups (conmap (fun (s, m) -> conc_step free_loc read update_mem cfgs gt
             (s, m)) l)
        if res = [] then l else conc_step_star_aux res
    conc_step_star_aux [(states, mem)]

type llvm_tcfg = Map<string, flowgraph<int, LLVM_edge_type, c_instr>>
let start_states (cfgs:llvm_tcfg) = new Map<string, state<int>>(seq {for KeyValue(t, G) in
    cfgs -> (t, (G.Start, new Map<string, LLVM_const>([]), [], []))})
let start_bufs (cfgs:llvm_tcfg) = new Map<string, (int * LLVM_const) list>(seq {for
    KeyValue(t, G) in cfgs -> (t, [])})

let run_SC cfgs = conc_step_star free_loc_SC read_SC update_mem_SC cfgs (new Map<string,
    LLVM_const>([])) (start_states cfgs, new Map<int, LLVM_const>([]))
```

```
let run_TSO_bufs cfgs = conc_step_star free_loc_TSO read_TSO update_mem_TSO cfgs (new Map<
    string, LLVM_const>([])) (start_states cfgs, (new Map<int, LLVM_const>([]), start_bufs
    cfgs))
(* clear out leftover buffers *)
let run_TSO cfgs = List.filter (fun (states, (m, b)) -> Map.forall (fun _ l -> l = []) b) (
    run_TSO_bufs cfgs)

(* examples *)
let llvm_cfg3 = { Nodes = set [5; 6; 7]; Edges = set [(5, 6, Seq); (6, 7, Seq)]; Start = 5;
     Exit = 7;
                    Label = new Map<int, c_instr>(List.toSeq [(5, Br_label); (6, Assign ("x
                        ", Add, Int_ty, CInt 1, CInt 2))]) }

let llvm_tcfg2 = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("t1
    ", llvm_cfg1); ("t3", llvm_cfg3)])

(* real transformations *)

(* simple RSE *)
let RSE0 : trans =
    TApplyAll (TIf ([AReplace ((MVar "n"), [Inj (IsPointer (EPVar "l"))])],
         SCEF (SCAnd (node "t" "n", SCAnd (stmt "t" (Inj (Store (MVar "ty1", EPVar "e1",
             MVar "ty2", EPVar "l"))),
            SCEU (node "t" "n", SCAnd (SCNot (node "t" "n"),
            stmt "t" (Inj (Store (MVar "ty3", EPVar "e2", MVar "ty4", EPVar "l")))))))),
                []))

let RSE_cfg1 = { Nodes = set [1; 2; 3; 4];
Edges = set [(1, 2, Seq); (2, 3, Seq); (3, 4, Seq)]; Start = 1; Exit = 4;
                    Label = new Map<int, c_instr>(List.toSeq [
                    (1, Store (Int_ty, CInt 1, Pointer_ty Int_ty, CPointer 1));
                    (2, Store (Int_ty, CInt 2, Pointer_ty Int_ty, CPointer 1));
                    (3, Store (Int_ty, CInt 3, Pointer_ty Int_ty, CPointer 1))]) }

(*let RSE_test1 = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("t1
    ", RSE_cfg1)])
printfn "%A" (run_TSO RSE_test1)
let RSE_result1 = test_trans RSE0 RSE_test1
List.iter (fun cfgs -> printfn "%A" (run_TSO cfgs)) RSE_result1
writer.Flush()*)
```

```
(* single-thread, multi-thread with error, corrected, example *)
let def t v = SCExs [("_opr", typeof<LLVM_op>); ("_ty1", typeof<LLVM_type>); ("_ty2",
    typeof<LLVM_type>); ("_ty3", typeof<LLVM_type>);
                      ("_e1", typeof<LLVM_const>); ("_e2", typeof<LLVM_const>); ("_e3",
                          typeof<LLVM_const>); ("_cmp", typeof<LLVM_cmp>)]
                     (SCOrs [stmt t (Inj (Assign (v, MVar "_opr", MVar "_ty1", EPVar "_e1",
                         EPVar "_e2")));
                             stmt t (Inj (ICmp (v, MVar "_cmp", MVar "_ty1", EPVar "_e1",
                                 EPVar "_e2")));
                             stmt t (Inj (Alloca (v, MVar "_ty1")));
                             stmt t (Inj (Load (v, MVar "_ty1", EPVar "_e1")));
                             stmt t (Inj (Cmpxchg (v, MVar "_t1", EPVar "_e1", MVar "_t2",
                                 EPVar "_e2", MVar "_t3", EPVar "_e3")))])
let used t v = SCExs [("_x", typeof<string>); ("_opr", typeof<LLVM_op>); ("_ty1", typeof<
    LLVM_type>); ("_ty2", typeof<LLVM_type>); ("_ty3", typeof<LLVM_type>);
                      ("_e1", typeof<LLVM_const>); ("_e2", typeof<LLVM_const>); ("_e3",
                          typeof<LLVM_const>); ("_cmp", typeof<LLVM_cmp>)]
                     (SCOrs [stmt t (Inj (Assign ("_x", MVar "_opr", MVar "_ty1", EPInj (
                         Local v), EPVar "_e2")));
                             stmt t (Inj (Assign ("_x", MVar "_opr", MVar "_ty1", EPVar "_e1
                                 ", EPInj (Local v))));
                             stmt t (Inj (ICmp ("_x", MVar "_cmp", MVar "_ty1", EPInj (Local
                                  v), EPVar "_e2")));
                             stmt t (Inj (ICmp ("_x", MVar "_cmp", MVar "_ty1", EPVar "_e1",
                                  EPInj (Local v))));
                             stmt t (Inj (Br_i1 (EPInj (Local v))));
                             stmt t (Inj (Load ("_x", MVar "_ty1", EPInj (Local v))));
                             stmt t (Inj (Store (MVar "_ty1", EPInj (Local v), MVar "_ty2",
                                 EPVar "_e2")));
                             stmt t (Inj (Store (MVar "_ty1", EPVar "_e1", MVar "_ty2",
                                 EPInj (Local v))));
                             stmt t (Inj (Cmpxchg ("_x", MVar "_t1", EPInj (Local v), MVar "
                                 _t2", EPVar "_e2", MVar "_t3", EPVar "_e3")));
                             stmt t (Inj (Cmpxchg ("_x", MVar "_t1", EPVar "_e1", MVar "_t2
                                 ", EPInj (Local v), MVar "_t3", EPVar "_e3")));
                             stmt t (Inj (Cmpxchg ("_x", MVar "_t1", EPVar "_e1", MVar "_t2
                                 ", EPVar "_e2", MVar "_t3", EPInj (Local v))))])

(* Let's try RSE again. *)
```

```
let not_loads t e = SCNot (SCEx ("_x", typeof<string>, SCEx ("_ty", typeof<LLVM_type>, stmt
     "t" (Inj (Load ("_x", MVar "_ty", e))))))
let not_stores t e = SCNot (SCExs [("_e", typeof<LLVM_const>); ("_ty1", typeof<LLVM_type>);
     ("_ty2", typeof<LLVM_type>)]
                                 (stmt "t" (Inj (Store (MVar "_ty1", EPVar "_e", MVar "_ty2
                                     ", e)))))

let RSE1:trans = TIf ([AReplace (MVar "n", [Inj (IsPointer (EPInj (Local "l")))])],
    SCEF (SCAnds [node "t" "n"; stmt "t" (Inj (Store (MVar "ty1", EPVar "e1", MVar "ty2",
        EPInj (Local "l"))));
                  SCAU (SCAnd (SCAll ("e", typeof<LLVM_const>, not_loads "t" (EPVar "e")),
                      SCNot (def "t" "l")),
                  SCAnd (SCNot (node "t" "n"), stmt "t" (Inj (Store (MVar "ty1'", EPVar "e1
                      '", MVar "ty2'", EPInj (Local "l")))))))]), [])

let RSE_fail1 = { Nodes = set [1; 2; 3; 4]; Edges = set [(1, 2, Seq); (2, 3, Seq); (3, 4,
    Seq)]; Start = 1; Exit = 4;
                    Label = new Map<int, c_instr>(List.toSeq [(1, Store (Int_ty, CInt 1,
                        Pointer_ty Int_ty, Local "x"));
                                                                      (2, Load ("x", Int_ty, Local
                                                                          "x"));
                                                                      (3, Store (Int_ty, CInt 3,
                                                                          Pointer_ty Int_ty, Local "
                                                                          x"))]) }

let RSE_fail2 = { Nodes = set [1; 2; 3; 4]; Edges = set [(1, 2, Seq); (2, 3, Seq); (3, 4,
    Seq)]; Start = 1; Exit = 4;
                    Label = new Map<int, c_instr>(List.toSeq [(1, Store (Int_ty, CInt 1,
                        Pointer_ty Int_ty, Local "x"));
                                                                      (2, Assign ("x", Add,
                                                                          Pointer_ty Int_ty, Local "
                                                                          x", CPointer 1));
                                                                      (3, Store (Int_ty, CInt 3,
                                                                          Pointer_ty Int_ty, Local "
                                                                          x"))]) }

let RSE_fail3 = { Nodes = set [1; 2; 3; 4; 5]; Edges = set [(1, 2, Seq); (2, 3, Seq); (3,
    4, Seq); (4, 5, Seq)]; Start = 1; Exit = 5;
                    Label = new Map<int, c_instr>(List.toSeq [(1, Store (Int_ty, CInt 1,
                        Pointer_ty Int_ty, Local "x"));
```

```
                                                                (2, Assign ("y", Add,
                                                                    Pointer_ty Int_ty, Local "
                                                                    x", CPointer 0));
                                                                (3, Load ("z", Pointer_ty
                                                                    Int_ty, Local "y"));
                                                                (4, Store (Int_ty, CInt 3,
                                                                    Pointer_ty Int_ty, Local "
                                                                    x"))]) }

let RSE_cfg2 = { Nodes = set [1; 2; 3; 4; 5; 6]; Edges = set [(1, 2, Seq); (2, 3, Seq); (3,
     4, Seq); (4, 5, Seq); (5, 6, Seq)]; Start = 1; Exit = 6;
                    Label = new Map<int, c_instr>(List.toSeq [(1, Alloca ("x", Int_ty));
                                                                (2, Assign ("y", Add,
                                                                    Pointer_ty Int_ty, Local "
                                                                    x", CPointer 0));
                                                                (3, Store (Int_ty, CInt 1,
                                                                    Pointer_ty Int_ty, Local "
                                                                    x"));
                                                                (4, Store (Int_ty, CInt 2,
                                                                    Pointer_ty Int_ty, Local "
                                                                    y"));
                                                                (5, Store (Int_ty, CInt 3,
                                                                    Pointer_ty Int_ty, Local "
                                                                    x"))]) }

(*let RSE_test2 = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("t1
    ", RSE_cfg2)])

printfn "%A" (run_SC RSE_test2)
let RSE_result2 = test_trans RSE1 RSE_test2
List.iter (fun cfgs -> printfn "%A" (run_SC cfgs)) RSE_result2*)


(* Okay, sure, but: can we introduce new behavior? I bet we can. *)
let RSE_cfg3 = { Nodes = set [1; 2; 3; 4; 5; 6; 7; 8]; Start = 1; Exit = 8;
                 Edges = set [(1, 2, Seq); (2, 3, Seq); (3, 4, Seq); (4, 5, Seq); (5, 6,
                     Seq); (6, 7, Seq); (7, 8, Seq)];
                 Label = new Map<int, c_instr>(List.toSeq [(1, Alloca ("x", Int_ty));
                                                             (2, Assign ("y", Add, Pointer_ty
                                                                  Int_ty, Local "x", CPointer
```

```
                                                            0));
                                                        (3, Store (Pointer_ty Int_ty,
                                                            Local "y", Pointer_ty (
                                                            Pointer_ty Int_ty), CPointer
                                                            5));
                                                        (4, Store (Int_ty, CInt 0,
                                                            Pointer_ty Int_ty, Local "x")
                                                            );
                                                        (5, Store (Int_ty, CInt 1,
                                                            Pointer_ty Int_ty, Local "x")
                                                            );
                                                        (6, Store (Int_ty, CInt 1,
                                                            Pointer_ty Int_ty, CPointer
                                                            13));
                                                        (7, Store (Int_ty, CInt 3,
                                                            Pointer_ty Int_ty, Local "x")
                                                            )]) }

let RSE_test2 = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("t1",
     RSE_cfg3)])

printfn "%A" (run_SC RSE_test2)
let RSE_result2 = test_trans (TApplyAll RSE1) RSE_test2
List.iter (fun cfgs -> printfn "%A" (run_SC cfgs)) RSE_result2

let RSE_cfg4 = increment { Nodes = set [8; 9; 10; 11; 12; 13; 14; 15]; Start = 8; Exit =
    15;
                 Edges = set [(8, 9, Seq); (9, 10, Seq); (10, 11, Seq); (11, 12, Seq); (12,
                      13, True); (12, 14, False); (13, 15, Seq); (14, 15, Seq)];
                 Label = new Map<int, c_instr>(List.toSeq [(8, Load ("l", Pointer_ty (
                     Pointer_ty Int_ty), CPointer 5)); (* use a global? *)
                                                        (9, Load ("w", Pointer_ty Int_ty
                                                            , CPointer 13));
                                                        (10, Load ("v", Pointer_ty
                                                            Int_ty, Local "l"));
                                                        (11, ICmp ("z", Sge, Int_ty,
                                                            Local "v", Local "w"));
                                                        (12, Br_i1 (Local "z"));
                                                        (13, Store (Int_ty, CInt 7,
                                                            Pointer_ty Int_ty, CPointer
```

```
                                                                            9));
                                                                        (14, Store (Int_ty, CInt 8,
                                                                            Pointer_ty Int_ty, CPointer
                                                                            9))]) } 1
(* Not sure whether I can simplify this, but it'll do the trick. *)

let RSE_test3 = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("t1",
     RSE_cfg3); ("t2", RSE_cfg4)])

(*printfn "%A" (run_SC RSE_test3)
let RSE_result3 = test_trans RSE1 RSE_test3
List.iter (fun cfgs -> printfn "%A" (run_SC cfgs)) RSE_result3

printfn "%s" "\n(* Now, how do we fix it? *)\n"*)

let RSE2:trans = TIf ([AReplace (MVar "n", [Inj (IsPointer (EPInj (Local "l")))])],
    SCEF (SCAnds [node "t" "n"; stmt "t" (Inj (Store (MVar "ty1", EPVar "e1", MVar "ty2",
        EPInj (Local "l"))));
                  SCAU (SCAnd (SCOr (SCAll ("e", typeof<LLVM_const>, SCAnd (not_loads "t" (
                      EPVar "e"), not_stores "t" (EPVar "e"))), node "t" "n"), SCNot (def "t
                      " "l")),
                  SCAnd (SCNot (node "t" "n"), stmt "t" (Inj (Store (MVar "ty1'", EPVar "e1
                      '", MVar "ty2'", EPInj (Local "l"))))))]), [])

(*printfn "%A" (run_SC RSE_test3)
let RSE_result4 = test_trans RSE2 RSE_test3
List.iter (fun cfgs -> printfn "%A" (run_SC cfgs)) RSE_result4*)

let skip_elim:trans = TMatch (SCEF (SCAnds [node "t" "n"; stmt "t" (Inj (Br_label)); SCEU (
    node "t" "n", SCAnd (SCNot (node "t" "n"), node "t" "n'"))]),
    TThen (TApplyAll (TIf ([AAddEdge (MVar "m", MVar "n'", MVar "l"); ARemoveEdge (MVar "m
        ", MVar "n", MVar "l")], SCEF (SCAnd (node "t" "m", out "t" "n" (MVar "l"))), [])),
           TIf ([AReplace (MVar "n", [])], SCTrue, [])), [])

let skip_test1 = { Nodes = set [1; 2; 3; 4]; Edges = set [(1, 2, Seq); (2, 3, Seq); (3, 4,
    Seq)]; Start = 1; Exit = 4;
                    Label = new Map<int, c_instr>(List.toSeq [(1, Store (Int_ty, CInt 1,
                        Pointer_ty Int_ty, CPointer 1));
                                                                           (2, Br_label);
```

```
                                                              (3, Store (Int_ty, CInt 3,
                                                                 Pointer_ty Int_ty,
                                                                 CPointer 1))]) }

(* Note: can't remove Start because we wouldn't know what to make the new Start. *)
let skip_test2 = { Nodes = set [1; 2; 3; 4]; Edges = set [(1, 2, Seq); (2, 3, Seq); (3, 4,
    Seq)]; Start = 1; Exit = 4;
                   Label = new Map<int, c_instr>(List.toSeq [(1, Br_label);
                                                              (2, Assign ("x", Add,
                                                                 Pointer_ty Int_ty,
                                                                 CPointer 0, CPointer 1));
                                                              (3, Store (Int_ty, CInt 3,
                                                                 Pointer_ty Int_ty, Local "
                                                                 x"))]) }

let skip_test3 = { Nodes = set [1; 2; 3; 4; 5]; Edges = set [(1, 2, True); (1, 4, False);
    (2, 3, True); (2, 4, False); (3, 4, Seq); (4, 5, Seq)]; Start = 1; Exit = 5;
                   Label = new Map<int, c_instr>(List.toSeq [(1, Br_i1 (CInt 0));
                                                              (2, Br_i1 (CInt 1));
                                                              (3, Store (Int_ty, CInt 3,
                                                                 Pointer_ty Int_ty,
                                                                 CPointer 1));
                                                              (4, Br_label)]) }

let lift_cfg cfg = new Map<string, flowgraph<int, LLVM_edge_type, c_instr>>(List.toSeq [("
    t1", cfg)])


printfn "%s" "(* Test 1: *)"
let skip_result1 = test_trans skip_elim (lift_cfg skip_test1)
printfn "%s" "\n(* Test 2: *)"
let skip_result2 = test_trans skip_elim (lift_cfg skip_test2)
printfn "%s" "\n(* Test 3: *)"
let skip_result3 = test_trans skip_elim (lift_cfg skip_test3)

ignore (System.Console.Read())
```

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[4] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Robert Atkey. CoqJVM: an executable specification of the Java virtual machine using dependent types. In *Proceedings of the 2007 international conference on Types for proofs and programs*, TYPES'07, pages 18–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa. Tool-assisted specification and verification of the JavaCard platform. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, pages 41–59, London, UK, UK, 2002. Springer-Verlag.

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011.

[8] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* SpringerVerlag, 2004.

[9] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* SpringerVerlag, 2004.

[10] Jan Olaf Blech and Sabine Glesner. A formal correctness proof for code generation from SSA form in Isabelle/HOL. In *Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik.* Lecture Notes in Informatics, September 2004.

[11] Hans-Juergen Boehm and Cormac Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* ACM, 2013.

[12] Jürgen Bohn, Werner Damm, Orna Grumberg, Hardi Hungar, and Karen Laster. First-order-ctl model checking. In Vikraman Arvind and Sundar Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 283–294. Springer Berlin Heidelberg, 1998.

[13] Manfred Broy, Ursula Hinkel, Tobias Nipkow, Christian Prehofer, and Birgit Schieder. Interpreter verification for a functional language. In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 77–88, London, UK, 1994. Springer-Verlag.

[14] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42:54–65, June 2007.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[16] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, LPAR'07, pages 211–225, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] ECMA. *ECMA-335: Common Language Infrastructure (CLI).* ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, December 2001.

[18] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, 2012. To appear.

[19] N.E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, Sep 1992.

[20] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[21] Lars Gesellensetter, Sabine Glesner, and Elke Salecker. Formal verification with Isabelle/HOL in practice: finding a bug in the GCC scheduler. In *Proceedings of the 12th international conference on Formal methods for industrial critical systems*, FMICS'07, pages 85–100, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *SIGPLAN Not.*, 36(3):248–260, January 2001.

[23] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[24] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Berlin / Heidelberg, 1980. 10.1007/3-540-10003-2_79.

[25] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[26] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 353–367, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Myra Van Inwegen and Elsa L. Gunter. HOL-ML. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 61–74, London, UK, 1994. Springer-Verlag.

[28] Sara Kalvala, Richard Warburton, and David Lacey. Program transformations using temporal logic side conditions. *ACM Trans. Program. Lang. Syst.*, 31(4):1–48, 2009.

[29] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298:583–626, April 2003.

[30] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[31] Jens Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, September 2003.

[32] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. *SIGPLAN Not.*, 37(1):283–294, January 2002.

[33] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.

[34] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 2–12, Washington, DC, USA, 2005. IEEE Computer Society.

[36] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification – from verified programs to verified systems. *Electron. Notes Theor. Comput. Sci.*, 217:23–40, July 2008.

[37] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. *SIGPLAN Not.*, 38:220–231, May 2003.

[38] Sorin Lerner, Todd Millstein, and Craig Chambers. Cobalt: A language for writing provably-sound compiler optimizations. *Electron. Notes Theor. Comput. Sci.*, 132(1):5–17, May 2005.

[39] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 42–54, New York, NY, USA, 2006. ACM.

[40] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[41] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[42] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41:1–31, July 2008.

[43] Guodong Li. Validated compilation through logic. In *Proceedings of the 17th international conference on Formal methods*, FM'11, pages 169–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[44] Guodong Li and Konrad Slind. Compilation as rewriting in higher order logic. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 19–34, Berlin, Heidelberg, 2007. Springer-Verlag.

[45] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 455–468, New York, NY, USA, 2012. ACM.

[46] LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, April 2014.

[47] Andreas Lochbihler. Verifying a compiler for java threads. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 427–447, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] Savi Maharaj and Elsa L. Gunter. Studying the ML module system in HOL. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 346–361, London, UK, 1994. Springer-Verlag.

[49] William Mansky, Dennis Griffith, and Elsa L. Gunter. Specifying and executing optimizations for parallel programs. In *GRAPHITE 2014*. Forthcoming, 2014.

[50] William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 371–386, Berlin, Heidelberg, 2010. Springer-Verlag.

[51] William Mansky and Elsa L. Gunter. Using locales to define a rely-guarantee temporal logic. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 299–314. Springer Berlin Heidelberg, 2012.

[52] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 273–284, New York, NY, USA, 2010. ACM.

[53] Paul E. McKenney. Memory ordering in modern microprocessors, part i. *Linux J.*, 2005(136):2–, August 2005.

[54] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[55] J. Strother Moore. A mechanically verified language implementation. *J. Autom. Reason.*, 5(4):461–492, 1989.

[56] J. Strother Moore. *Piton: a mechanically verified assembly-level language*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[57] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 144–152, New York, NY, USA, 1973. ACM.

[58] Leonardo Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[59] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[60] Tobias Nipkow. Verified lexical analysis. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 1–15, London, UK, 1998. Springer-Verlag.

[61] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[63] G. D. Plotkin. A structural approach to operational semantics, 1981.

[64] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[65] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK, 1998. Springer-Verlag.

[66] Y. S. Ramakrishna, L.E. Moser, L. K. Dillon, P.M. Melliar-Smith, and G. Kutty. An automata-theoretic decision procedure for propositional temporal logic with since and until, 1992.

[67] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[68] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *J. Autom. Reason.*, 40:307–326, May 2008.

[69] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[70] Ando Saabas and Tarmo Uustalu. Type systems for optimizing stack-based code. *Electron. Notes Theor. Comput. Sci.*, 190(1):103–119, July 2007.

[71] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. *SIGPLAN Not.*, 46(1):43–54, January 2011.

[72] Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 343–348, New York, NY, USA, 2010. ACM.

[73] Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proceedings of the 18th International Conference on Automated Deduction*, CADE-18, pages 63–77, London, UK, 2002. Springer-Verlag.

[74] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 111–121, New York, NY, USA, 2010. ACM.

[75] David Trachtenherz. Infinite lists. *Archive of Formal Proofs*, February 2011. `http://afp.sf.net/entries/List-Infinite.shtml`, Formal proof development.

[76] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 146–162, Berlin, Heidelberg, 2011. Springer-Verlag.

[77] R.J. van Glabbeek. The Linear Time-Branching Time Spectrum I - The Semantics of Concrete, Sequential Processes. In *Handbook of Process Algebra, chapter 1*, pages 3–99. Elsevier, 1990.

[78] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[79] W. Young. Verified compilation in micro-Gypsy. *SIGSOFT Softw. Eng. Notes*, 14:20–26, November 1989.

[80] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012.

[81] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. *SIGPLAN Not.*, 48(6):175–186, June 2013.