Specifying and Executing Optimizations for Parallel Programs

William Mansky Dennis Griffith
Elsa L. Gunter
Department of Computer Science, University of Illinois at Urbana-Champaign,
Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302
{mansky1,dgriffi3,egunter}@illinois.edu

Compiler optimizations, usually expressed as rewrites on program graphs, are a core part of all modern compilers. However, even production compilers have bugs, and these bugs are difficult to detect and resolve. The problem only becomes more complex when compiling parallel programs; from the choice of graph representation to the possibility of race conditions, optimization designers have a range of factors to consider that do not appear when dealing with single-threaded programs. In this paper we present PTRANS, a domain-specific language for formal specification of compiler transformations, and describe its executable semantics. The fundamental approach of PTRANS is to describe program transformations as rewrites on control flow graphs with temporal logic side conditions. The syntax of PTRANS allows cleaner, more comprehensible specification of program optimizations; its executable semantics allows these specifications to act as prototypes for the optimizations themselves, so that candidate optimizations can be tested and refined before going on to include them in a compiler. We demonstrate the use of PTRANS to state, test, and refine the specification of a redundant store elimination optimization on parallel programs.

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 – most 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 [16], and some of these bugs have been shown to result from mishandling concurrency [10]. 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 goal of VeriF-OPT, a Verification Framework for Optimizations and Program Transformations, is to make correct compilation more widely accessible by providing a standard approach and toolset for specifying, testing, and verifying compilers for a wide range of languages, with a particular focus on optimization and compilation of parallel programs. The core approach of the framework is a new way

Dragan Bošnački, Stefan Edelkamp, Alberto Lluch Lafuente
& Anton Wijs (Eds.): 3rd Workshop on
GRAPH Inspection and Traversal Engineering (GRAPHITE 2014)
EPTCS 159, 2014, pp. 58–70, doi:10.4204/EPTCS.159.6

of looking at optimizations: as rewrites on control flow graphs with temporal logic side conditions, as first proposed by Lacey et al. [6]. This approach is put into practice using a domain-specific language for specifying compiler optimizations and transformations, which we call PTRANS. Temporal logic formulae over program graphs allow us to simply and clearly state the conditions under which an optimization should be applied. PTRANS has both abstract mathematical semantics, derived from its predecessor language TRANS [4], and *executable* semantics that can be used by compiler designers to test and refine their transformations on actual program graphs. In this paper, we present the syntax and the abstract and executable semantics of PTRANS, and illustrate how it can be used to rapidly prototype and test compiler optimizations. Ultimately, we hope that the approach outlined in this paper will assist compiler writers in creating complex, reliable optimizations for parallel code.

2 The PTRANS Specification Language

2.1 PTRANS: A Language for Parallel Program Transformation

The basic approach of the PTRANS specification language is modeled after the TRANS language of Kalvala et al. [4]: optimizations are specified as rewrites on program code in the form of control flow graphs, with side conditions given in Computation Tree Logic (CTL). Intuitively, the rewrite portion of an optimization expresses the transformation to be made, and the side condition characterizes the situations in which the optimization should be applied. The syntax of PTRANS is given by the following grammar:

$$
\begin{array}{rcl}
A & ::= & \mathsf{add\_edge}(n,m,\ell) \mid \mathsf{remove\_edge}(n,m,\ell) \mid \mathsf{split\_edge}(n,m,\ell,i) \\
 & \mid & \mathsf{replace}\ n\ \mathsf{with}\ i_1,\ldots,i_k \\
\varphi & ::= & \mathsf{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathrm{A}\ \varphi\ \mathrm{U}\ \varphi \mid \mathrm{E}\ \varphi\ \mathrm{U}\ \varphi \mid \mathrm{A}\ \varphi\ \mathrm{B}\ \varphi \mid \mathrm{E}\ \varphi\ \mathrm{B}\ \varphi \mid \exists x.\ \varphi \\
T & ::= & A_1,\ldots,A_k \text{ if } \varphi \mid \mathrm{MATCH}\ \varphi\ \mathrm{IN}\ T \mid T\ \mathrm{THEN}\ T \mid T \square T \mid \mathrm{APPLY\_ALL}\ T
\end{array}
$$

It consists of three main syntactic categories: actions, side conditions, and transformations. The atomic *actions* A include add_edge and remove_edge, which add and remove (ℓ-labeled) edges between the specified nodes; split_edge, which splits an edge between two nodes, inserting a new node between them; and replace, which replaces the instruction at a given node with a sequence of instructions, adding new nodes to contain the new instructions as necessary. 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.

The side conditions φ of PTRANS are based on First-Order CTL (FOCTL), and are built starting from a set of atomic predicates p. The B ("back-to") operators are the past-time counterparts to the U ("until") operators; for instance, E φ_1 B φ_2 holds when there exists some path backwards through a graph such that φ_1 holds until the previous point at which φ_2 holds. The derived "finally" and "globally" operators EF, AF, EG, AG are defined from the U operators in the usual way. The 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.

At the top level, a transformation T is built out of conditional rewrites combined with *strategies*. $A_1,\ldots,A_k$ if φ is the basic pairing of one or more rewrites with a temporal logic side condition. 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 transformations. The THEN and $\square$ 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.

2.2 Parallel Control Flow Graphs

The TRANS approach depends fundamentally on a notion of control flow graph (CFG). Atomic rewrites are rewrites on CFGs, and 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 [5], 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.

Definition 1. *A* CFG *is a tuple* (N,E,s,x,L) *describing a labeled directed graph, where* N *is a finite set of nodes,* $E : 2^{N\times N\times T}$ *is a set of labeled edges,* $s,x \in N$ *are the start and exit node of the graph respectively, and* $L : N \to I$ *is a labeling of nodes with program instructions. The set of edge labels* T *is provided by the target language, but must include the sequencing edge* seq. *A* tCFG *is a collection of disjoint CFGs, one for each thread in the program being represented. If* $\mathscr{G}$ *is a tCFG and* t *is a thread, we write* $\mathscr{G}_t$ *for the CFG of* t *in* $\mathscr{G}$.

The set of atomic predicates used in side conditions may depend on the target language under consideration, but some simple predicates are applicable to almost every language, and many optimizations can be specified with only language-independent predicates. These predicates include the following:

- $\mathsf{node}_t(n)$, which is true of a state q when $q(t) = n$.
- $\mathsf{stmt}_t(i)$, which is true of a state q when the instruction at q is i in $\mathscr{G}_t$.
- $\mathsf{out}_t(n',\ell)$, which is true of a state q when $q(t)$ has an outgoing edge to n' with label ℓ in $\mathscr{G}_t$.
- start, which is true when q is at the start node of each of its component graphs, and exit, which is true when q is at the exit node for each graph.

Note that all of these predicates are static properties of tCFGs that do not depend on the semantics of the language under consideration. 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.

3 The Semantics of PTRANS

3.1 Abstract Semantics

In this section we present the mathematical semantics of PTRANS, based on the semantics of TRANS by Kalvala et al. [4]. The semantics of actions is given by a function $[\![A]\!](\sigma,\mathscr{G})$ that takes an action, a substitution (a partial map from metavariables to program objects), and a tCFG and returns the tCFG that results when the action is performed. 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 $\mathscr{G}_t$ on which to perform the action (if two nodes mentioned are in two different graphs, the action simply fails). Suppose we have $\mathscr{G}_t = (N_t,E_t,s_t,x_t,L_t)$; then the semantics of actions are defined as follows:

- $[\![\mathsf{add\_edge}(n,m,\ell)]\!](\sigma,\mathscr{G}) = \mathscr{G}(t \mapsto (N_t, E_t \cup \{(\sigma(n),\sigma(m),\sigma(\ell))\}, s_t, x_t, L_t))$
- $[\![\mathsf{remove\_edge}(n,m,\ell)]\!](\sigma,\mathscr{G}) = \mathscr{G}(t \mapsto (N_t, E_t - \{(\sigma(n),\sigma(m),\sigma(\ell))\}, s_t, x_t, L_t))$
- $[\![\mathsf{replace}\ n\ \mathsf{with}\ i_1,...,i_k]\!](\sigma,\mathscr{G}) = \mathscr{G}(t \mapsto (N_t \cup \{n_2,...,n_k\}, \{\mathsf{remap\_succ}(\sigma(n),n_k,e) \mid e \in E\} \cup \{(n_j,n_{j+1},\mathsf{seq}) \mid 1 < i < k\}, s_t, x_t, L_t + (n_1 \mapsto \sigma(i_1),...,n_k \mapsto \sigma(i_k))))$, where $n_1 = \sigma(n)$ and $n_2,...,n_k$ are new nodes not in $\mathscr{G}$, and remap_succ is defined below

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

In the replace action, we must not only introduce new seq edges between the added nodes, but also move the outgoing edges of the initial node n_1 to instead be outgoing edges of the last added node n_k. To do this we use the auxiliary remap_succ function, defined as

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

The semantics of a list of actions $A_1,...,A_k$ is the composition of the semantic functions of the individual actions, i.e., the graph resulting from applying all of the actions in sequence.

The side conditions of PTRANS are given in the branching-time temporal logic FOCTL. A CTL formula expresses a property over a (possibly infinite) tree of *states*, and at each branching point quantifies over the possible *paths* forward or backward from that state (written as *Paths* and *RPaths* respectively; note that backward paths must always reach the start state of the graph). The formulae are made first-order by allowing variables to appear in the atomic state predicates p, and we can quantify over these variables with the $\exists$ operator. The semantics of an FOCTL formula is given by a satisfaction relation of the form $\mathscr{G},\sigma,q \models \varphi$, where $\mathscr{G}$ is a tCFG, σ a substitution of values for metavariables, q a state (a vector of points in a tCFG), and φ a FOCTL formula, defined as follows (where λ_i denotes the ith element of the path λ):

- $\mathscr{G},\sigma,q \models \mathsf{true}$
- $\mathscr{G},\sigma,q \models p$ if $\sigma(p)$ is true at q in the semantics for $\sigma(p)$ provided by the target language
- $\mathscr{G},\sigma,q \models \varphi_1 \wedge \varphi_2$ if $\mathscr{G},\sigma,q \models \varphi_1$ and $\mathscr{G},\sigma,q \models \varphi_2$
- $\mathscr{G},\sigma,q \models \neg\varphi$ if $\mathscr{G},\sigma,q \not\models \varphi$
- $\mathscr{G},\sigma,q \models$ A φ_1 U φ_2 if $\forall\lambda \in \mathit{Paths}(\mathscr{G},q).\ \exists i.\ \mathscr{G},\sigma,\lambda_i \models \varphi_2 \wedge \forall j < i.\ \mathscr{G},\sigma,\lambda_j \models \varphi_1$
- $\mathscr{G},\sigma,q \models$ E φ_1 U φ_2 if $\exists\lambda \in \mathit{Paths}(\mathscr{G},q).\ \exists i.\ \mathscr{G},\sigma,\lambda_i \models \varphi_2 \wedge \forall j < i.\ \mathscr{G},\sigma,\lambda_j \models \varphi_1$
- $\mathscr{G},\sigma,q \models$ A φ_1 B φ_2 if $\forall\lambda \in \mathit{RPaths}(\mathscr{G},q).\ \exists i.\ \mathscr{G},\sigma,\lambda_i \models \varphi_2 \wedge \forall j < i.\ \mathscr{G},\sigma,\lambda_j \models \varphi_1$
- $\mathscr{G},\sigma,q \models$ E φ_1 B φ_2 if $\exists\lambda \in \mathit{RPaths}(\mathscr{G},q).\ \exists i.\ \mathscr{G},\sigma,\lambda_i \models \varphi_2 \wedge \forall j < i.\ \mathscr{G},\sigma,\lambda_j \models \varphi_1$
- $\mathscr{G},\sigma,q \models \exists x.\ \varphi$ if $\exists o.\ \mathscr{G},\sigma(x \mapsto o),q \models \varphi$

We write $\mathscr{G},\sigma \models \varphi$ to abbreviate $\mathscr{G},\sigma,q_0 \models \varphi$, where q_0 is the vector that for each CFG in $\mathscr{G}$ gives that CFG's starting node.

The semantics of strategies is given by a function $[\![T]\!](\tau,\mathscr{G})$ that takes a transformation, a 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 to a graph some finite (but unbounded) number of times:

$$\frac{}{G \in \mathsf{apply\_some}(T,\tau,G)} \qquad \frac{G' \in [\![T]\!](\tau,G) \quad G'' \in \mathsf{apply\_some}(T,\tau,G')}{G'' \in \mathsf{apply\_some}(T,\tau,G)}$$

Then the semantics of strategies is defined as follows:

- $[\![A_1,...,A_k \text{ if } \varphi]\!](\tau,\mathscr{G}) = \{\mathscr{G}' \mid \exists\sigma.\ \sigma|_{\mathrm{dom}(\tau)} = \tau \wedge \mathscr{G},\sigma \models \varphi \wedge \mathscr{G}' = [\![A_1,...,A_k]\!](\sigma,\mathscr{G})\}$
- $[\![\text{MATCH } \varphi \text{ IN } T]\!](\tau,\mathscr{G}) = \{\mathscr{G}' \mid \exists\sigma.\ \sigma|_{\mathrm{dom}(\tau)} = \tau \wedge \mathscr{G},\sigma \models \varphi \wedge \mathscr{G}' \in [\![T]\!](\sigma,\mathscr{G})\}$
- $[\![T_1 \text{ THEN } T_2]\!](\tau,\mathscr{G}) = \bigcup_{\mathscr{G}' \in [\![T_1]\!](\tau,\mathscr{G})} [\![T_2]\!](\tau,\mathscr{G}')$

- $[\![T_1 \square T_2]\!](\tau,\mathscr{G}) = [\![T_1]\!](\tau,\mathscr{G}) \cup [\![T_2]\!](\tau,\mathscr{G})$
- $[\![\text{APPLY\_ALL } T]\!](\tau,\mathscr{G}) = \mathsf{apply\_some}([\![T]\!],\tau,\mathscr{G}) - \{\mathscr{G}' \mid \exists \mathscr{G}'' \neq \mathscr{G}'.\ \mathscr{G}'' \in [\![T]\!](\tau,\mathscr{G}')\}$

Note in particular the semantics for APPLY_ALL, which produces the set of graphs that result from applying the transformation T repeatedly in any way such that, ultimately, T can no longer be applied to modify the final result.

While the semantic function for actions is straightforwardly executable (modulo suitable data structures for representing sets), the semantic function for transformations is not; it explicitly uses existential witnesses to create the (potentially infinite) set of result graphs. In particular, we frequently quantify over all substitutions that satisfy the side conditions of a transformation. In the remainder of this section, we will give a more directly executable semantics for transformations, but we must first present a method for computing the satisfying substitutions of an FOCTL side condition.

3.2 FOCTL Model Finding

The model checking problem for CTL and its variants is a well-studied problem with a well-known efficient algorithm [2], but considerably less attention has been given to the related problem of *model finding*. The model finding problem in its general form is this: suppose we have an FOCTL formula φ built from a set of atomic predicates, where the predicates may contain free variables. Given a transition system $\mathscr{S}$ and an interpretation of the atomic predicates on $\mathscr{S}$, what are the possible assignments of values to the free variables of φ such that φ holds on $\mathscr{S}$? When a formula contains no free variables, model finding is simply model checking; in the general case, it is considerably more complex.

Following Bohn et al. [1], we present an algorithm for FOCTL model finding. The algorithm is given in a functional style and can be straightforwardly implemented in a functional programming language. We will present the algorithm on a single CFG; it can be extended to tCFGs via a cross product construction. We find satisfying models *symbolically*, by defining a function SATIS that, given a formula φ and a node v, constructs a non-temporal first-order formula characterizing the set of substitutions that make φ true at v. The following theorem states the correctness of the algorithm:

Theorem 1. *Let $\mathscr{G} = (\mathscr{N},\mathscr{E},s,x,L)$ be a CFG, $v \in \mathscr{N}$ and φ a FOCTL formula. Then* $\{\sigma \mid \mathscr{G};\sigma;v \vDash \varphi\} = \{\sigma \mid \sigma \vDash_{\text{FOL}} \textsc{Satis}(\varphi)(v)\}$.

The theorem is proved by induction on the lexicographic order of the number of A φ_1 U φ_2-headed subformulae of φ and the number of subformulae of φ; we will define SATIS and give the proof of its correctness case by case. When the head connective is a non-temporal connective, SATIS recursively translates its subformulae, leaving the connective untouched:

$$\textsc{Satis}(p(\vec{x}))(v) = p(\vec{x}) \qquad \textsc{Satis}(\varphi_1 \wedge \varphi_2)(v) = \textsc{Satis}(\varphi_1)(v) \wedge \textsc{Satis}(\varphi_2)(v)$$
$$\textsc{Satis}(\neg\varphi)(v) = \neg\textsc{Satis}(\varphi)(v) \qquad \textsc{Satis}(\exists x.\ \varphi)(v) = \exists x.\ \textsc{Satis}(\varphi)(v)$$

Correctness for these cases follows directly from the inductive hypothesis.

When $\varphi = \text{E } \varphi_1 \text{ U } \varphi_2$, we need to ensure that we find a suitable witness path for the until-formula for each substitution. To do so, we define an auxilary function $\textsc{Paths}_{\leftarrow}(I,F,n,v)$ that takes an invariant $I\colon \mathscr{N} \to \textbf{FOL}$, a final requirement $F\colon \mathscr{N} \to \textbf{FOL}$, a path length n, and a node $v \in \mathscr{N}$, as follows:

$$\textsc{Paths}_{\leftarrow}(I,F,0,v) = F(v)$$
$$\textsc{Paths}_{\leftarrow}(I,F,n,v) = \textsc{Paths}_{\leftarrow}(I,F,n-1,v) \vee \left(I(v) \wedge \bigvee_{v' \in \mathsf{succ}(\mathscr{E},v)} \textsc{Paths}_{\leftarrow}(I,F,n-1,v') \right)$$

where $\mathsf{succ}(\mathscr{E}, v)$ is the set of successors of v in $\mathscr{E}$, i.e., $\{v' \mid (v, v', \ell) \in \mathscr{E}\}$.

Lemma 1. $\textsc{Paths}_{\leftarrow}(I, F, n, v)$ *characterizes the set of substitutions* σ *such that there is a path* λ *from* v *of length* $k \leq n$ *along which* $\mathscr{G}; \sigma; \lambda_k \vDash F(\lambda_k)$ *and* $\mathscr{G}; \sigma; \lambda_i \vDash I(\lambda_i)$ *for all* $i < k$.

We can then define

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

and finish the proof of this case by noting that, since if there is any witness there must be a cycle-free witness, Lemma 1 ensures the presence of a suitable witness.

When $\varphi = \mathrm{A}\ \phi_1\ \mathrm{U}\ \phi_2$ we again need to look for witnesses to the until-formula, this time in a conjunctive fashion. To do this we define the auxiliary function $\textsc{Paths}_{\wedge}(I, F, n, v)$ that takes an invariant $I \colon \mathscr{N} \to \mathbf{FOL}$, a final requirement $F \colon \mathscr{N} \to \mathbf{FOL}$, a length n, and a node $v \in \mathscr{N}$:

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

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

Lemma 2. $\textsc{Paths}_{\wedge}(I, F, n, v)$ *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* $\mathscr{G}; \sigma; \lambda_i \vDash F(\lambda_i)$ *and* $\mathscr{G}; \sigma; \lambda_i \vDash I(\lambda_j)$ *for all* $j < i$.

This correctness lemma is more difficult to state, since the conjunctive search and the presence of a sink (the exit node) means we must carefully handle paths with length less than $|\mathscr{N}|$. Unfortunately, while this gives us an ability to say that we can "always" find a witness for our until-formula, this function by itself still allows for infinite paths that never reach their satisfying witness. We need one more auxilary function to help us avoid these infinite counterexamples, defined below.

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

$$\textsc{Paths}_{\rightarrow}(I, n, v) = \bigvee_{v' \in \mathsf{succ}(\mathscr{E}, v)} \left(I(v') \wedge \textsc{Paths}_{\rightarrow}(I, n-1, v') \right)$$

Lemma 3. $\textsc{Paths}_{\rightarrow}(F, n, v)$ *characterizes the set of substitutions* σ *such that there is a path* λ *from* v *of length* n *along which* $\mathscr{G}; \sigma; \lambda_i \vDash I(\lambda_i)$ *for every* i.

We can then define

$$\begin{aligned} \textsc{Satis}(\mathrm{A}\ \varphi_1\ \mathrm{U}\ \varphi_2)(v) = \ & \neg \textsc{Paths}_{\rightarrow}(\textsc{Satis}(\varphi_1 \wedge \neg\varphi_2), |\mathscr{N}| + 1, v) \\ & \wedge \textsc{Paths}_{\wedge}(\textsc{Satis}(\varphi_2), \textsc{Satis}(\varphi_1), |\mathscr{N}|, v) \end{aligned}$$

We use our two auxilary functions to ensure that every path from v has a suitable witness for φ_2 and that, by the pigeonhole principle, it has no paths along which φ_1 holds and φ_2 is never reached. This case demonstrates why simple induction on the size of φ is not sufficient to prove correctness; $\textsc{Satis}(\mathrm{A}\ \varphi_1\ \mathrm{U}\ \varphi_2)(v)$ makes recursive calls on strictly larger formulae than $\mathrm{A}\ \varphi_1\ \mathrm{U}\ \varphi_2$, but those subformulae have fewer AU-connectives.

In handling the past-time connectives we have a slight advantage: since all paths backward must eventually reach the start node of the graph, we can ignore the possiblity of infinite paths. We make an additional simplifying assumption that all nodes in the graph are reachable from the start node (unreachable nodes can safely be discarded). This allows us to handle the backwards cases with the duals

of PATHS$_{\leftarrow}$ and PATHS$_{\wedge}$ formed by following predecessors instead of successors, denoted PATHS$_{\overleftarrow{\leftarrow}}$ and PATHS$_{\overleftarrow{\wedge}}$ respectively.

$$\text{SATIS}(\text{E } \varphi_1 \text{ B } \varphi_2)(v) = \text{PATHS}_{\overleftarrow{\leftarrow}}(\text{SATIS}(\varphi_2), \text{SATIS}(\varphi_1), \mathscr{N}, v)$$
$$\text{SATIS}(\text{A } \varphi_1 \text{ B } \varphi_2)(v) = \text{PATHS}_{\overleftarrow{\wedge}}(\text{SATIS}(\varphi_2), \text{SATIS}(\varphi_1), \mathscr{N}, v)$$

This completes the definition of the SATIS function. The running time of the algorithm as written is $O\left(|\mathscr{N}|^{|\mathscr{N}||\phi|}\right)$, but we can do better using dynamic programming. We begin with a table of $O(|\phi||\mathscr{N}|)$ entries for SATIS and tables of size $O(|\mathscr{N}|)$ for the each of the path-searching functions, representing the results for each possible input given the size of $\mathscr{G}$ and the subformulae of φ. Filling each of the path tables (assuming their arguments are already evaluated) takes $O(|\mathscr{N}|^2)$ time, and the tables can be reused to answer their queries for all vertices. Thus, the amortized running time for filling an entry in the table for SATIS is $O(|\mathscr{N}|)$, and the overall reduction runs in $O(|\phi||\mathscr{N}|^2)$ time. Once we have the characteristic formula provided by SATIS, if the basic language of atomic predicates is amenable to SMT solving, we can use a solver to compute the concrete set of satisfying models.

3.3 Executable Semantics for Strategies

Given the model finding algorithm described above, we can define a function get_models$(\tau, \mathscr{G}, \varphi)$ that computes the satisfying models of φ by generating a first-order formula that represents the set of substitutions that satisfy φ, 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 1 then assures us that get_models$(\tau, \mathscr{G}, \varphi) = \{\sigma \mid \mathscr{G}; \sigma \vDash \varphi \wedge \sigma|_{\text{dom}(\tau)} = \tau\}$, and so get_models serves as an executable method for finding satisfying models of PTRANS side conditions. Using get_models, we can write an executable function trans_sf that finds the semantics of a transformation, defined as follows (recall that the abstract semantic function for actions is already executable):

- trans_sf$(A_1, ..., A_k \text{ if } \varphi, \tau, \mathscr{G}) = \{\text{for each } \sigma \text{ in get_models}(\tau, \mathscr{G}, \varphi), [\![A_1, ..., A_k]\!](\sigma, \mathscr{G})\}$
- trans_sf$(\text{MATCH } \varphi \text{ IN } T, \tau, \mathscr{G}) = \{\text{for each } \sigma \text{ in get_models}(\tau, \mathscr{G}, \varphi), \text{trans_sf}(T, \sigma, \mathscr{G})\}$
- trans_sf$(T_1 \text{ THEN } T_2, \tau, \mathscr{G}) = \bigcup_{\mathscr{G}' \in \text{trans_sf}(T_1, \tau, \mathscr{G})} \text{trans_sf}(T_2, \tau, \mathscr{G}')$
- trans_sf$(T_1 \square T_2, \tau, \mathscr{G}) = \text{trans_sf}(T_1, \tau, \mathscr{G}) \cup \text{trans_sf}(T_2, \tau, \mathscr{G})$
- trans_sf$(\text{APPLY_ALL } T, \tau, \mathscr{G}) = \text{let } R = \text{trans_sf}(T, \tau, \mathscr{G}) \text{ in}$
  $\text{if } R = \{\mathscr{G}\} \text{ then } R \text{ else } \bigcup_{\mathscr{G}' \in R} \text{trans\_sf}(\text{APPLY\_ALL } T, \tau, \mathscr{G}')$

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 [![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 $\mathscr{G}$ indefinitely, the infinite sequence of rewrites would contribute nothing to $[\![\text{APPLY\_ALL } T]\!](\tau, \mathscr{G})$. The trans_sf function, on the other hand, attempts to apply T to $\mathscr{G}$ indefinitely, and so will never terminate. However, in all finite cases it can be shown that trans_sf$(T, \tau, \mathscr{G}) = [\![T]\!](\tau, \mathscr{G})$, 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). 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.

4 Designing and Prototyping Optimizations with PTRANS

4.1 A Sample Target Language: MiniLLVM

In this section, we will develop an optimization in PTRANS and show how its executable semantics can be of use in the design process. We will begin by defining a target language: MiniLLVM, a simplification of the LLVM intermediate language [7]. The syntax of MiniLLVM is as follows:

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

$$\begin{aligned} \textit{instr} ::= \quad & \%\mathrm{x} = \mathrm{op}\ \textit{type}\ \textit{expr}, \textit{expr} \mid \%\mathrm{x} = \texttt{icmp}\ \mathrm{cmp}\ \textit{type}\ \textit{expr}, \textit{expr} \mid \texttt{br i1}\ \textit{expr} \mid \texttt{br} \mid \\ & \%\mathrm{x} = \texttt{call}\ \textit{type}\ (\textit{expr}, ..., \textit{expr}) \mid \texttt{return}\ \textit{expr} \mid \%\mathrm{x} = \texttt{alloca}\ \textit{type} \mid \\ & \%\mathrm{x} = \texttt{load}\ \textit{type}^*\ \textit{expr} \mid \texttt{store}\ \textit{type}\ \textit{expr}, \textit{type}^*\ \textit{expr} \mid \texttt{is\_pointer}\ \textit{expr} \end{aligned}$$

MiniLLVM expressions are either local variables (%x), global variables (@x), or constants. 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, 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 encoded in the edges of the CFG, the label arguments to `br` instructions and function names in `call` instructions are omitted. Although each `alloca` instruction is executed by a single thread, the memory allocated can be exposed to other threads by storing its location in a global variable or fixed memory location.

For the purposes of this example, we will assume that MiniLLVM has a straightforward interleaved semantics, with a sequential consistency memory model: i.e., in each step one thread in the program executes, and any memory operations immediately update the shared memory and are visible to all other threads. More relaxed memory models, such as total or partial store ordering, are important to consider when designing compiler optimizations on parallel programs, and any operational memory model can be integrated into the semantics of MiniLLVM (and thus the optimization testing process) with little difficulty.

4.2 Writing PTRANS Optimizations

Now that we have a target language, we can begin to define PTRANS transformations on MiniLLVM. Our case study will be a *redundant store elimination* optimization (RSE), which removes stores that may be overwritten before they are used, as shown in Figure 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.

The rewrite involved is simple: 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

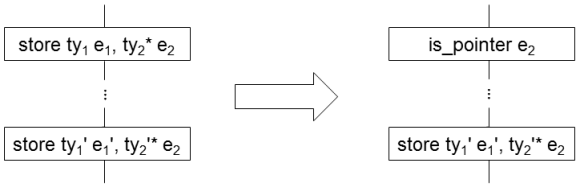


Figure 1: Redundant Store Elimination

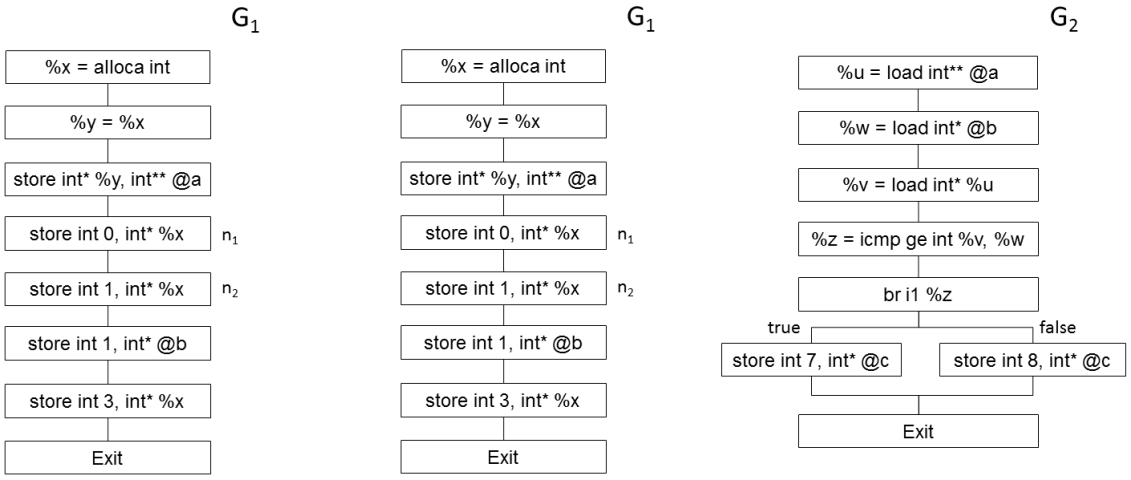
optimization safe, we must also require some property to hold on the instructions between n and the following store. For instance, if the value of e_2 is changed before the next store to e_2, removing the store at n would change the behavior of the program. Let φ be some restriction on the types of instructions that can appear between n and the stores that make it redundant; then the PTRANS specification of RSE can be written as:

$$\begin{aligned} RSE(\varphi) \triangleq\ & \text{replace } n \text{ with } \texttt{is\_pointer } e_2 \text{ if} \\ & \text{EF } \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\texttt{store } ty_1\ e_1, ty_2^*\ e_2) \wedge \\ & \text{A } \varphi \text{ U } (\neg \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\texttt{store } ty_1'\ e_1', ty_2'^*\ e_2)) \end{aligned}$$

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 our 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 indeed 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). 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 $\varphi_1 \triangleq \neg \mathsf{def}_t(e_2) \wedge \forall e.\ \mathsf{not\_loads}_t(e)$.

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 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 2b. Although the program is not well synchronized, we can see that the false branch in G_2 will never be taken, 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



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

Figure 2: An RSE example

$RSE(\varphi_1)$ 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 `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 Implementation

We have implemented the executable semantics of PTRANS described above in F# [15], taking advantage of its integration with the Z3 SMT solver [11]. The semantic functions for actions and strategies in Section 3.3 can be straightforwardly translated into F# code. We use the algorithm of Section 3.2 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 4.2 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.

6 Related Work

Our work builds on the TRANS approach of expressing optimizations as rewrites on control flow graphs with temporal logic side conditions due to Lacey et al. [6] and Kalvala et al. [4]. The most closely related tool is Cobalt [8], a system for specifying optimizations in a TRANS-like language. Cobalt optimizations are both executable and automatically verified, though it provides no support for iterative refinement of possibly incorrect specifications. Automation also comes at the cost of expressiveness: Cobalt is limited to a much smaller set of CTL side conditions than TRANS or PTRANS, and thus can express a smaller range of optimizations. To the best of our knowledge, neither Cobalt nor any other work stemming from the TRANS approach has yet addressed the question of parallelism.

He and Bowen [3] have also developed a language for specifying and prototyping compiler transformations, focusing particularly on the code generation phase of compilation. Their language consists of if-expressions analogous to our strategy-free transformations, and is implemented as a set of Horn clauses in Prolog. Rather than giving operational semantics for a real-world target language, they model programs directly as sequences of modifications to the machine state. Because they deal primarily with language-to-language translation, their transformations are not innately composable, and they deal largely with local peephole optimizations rather than those involving dataflow analysis.

CompCert [9], the definitive example of a proof of compiler correctness, includes a Coq-based framework for specifying and verifying compiler optimizations; executable semantics are obtained by extracting code from the Coq definition, guaranteeing its correctness. Their specifications follow the traditional algorithmic approach to dataflow analysis, with the conditions under which an optimization should be applied expressed as a set of transfer functions for dataflow equations, and must be written as instances of Coq functors rather than as a separate domain-specific language. The ongoing CompCertTSO project [14] seeks to add support for concurrency to CompCert, and has involved the specification and verification of a small number of concurrency-specific optimizations, as well as a range of sequential optimizations that can be lifted as-is to the concurrent case. CompCertTSO also verifies each translation in the chain from high-level source language to low-level machine code, while the work presented here is limited to transformations within a single target language. We believe that the most significant advantage of our approach is its language- and memory-model independence; we make no particular assumptions that restrict us to MiniLLVM or a particular treatment of concurrent memory models.

7 Conclusion and Future Work

In this paper, we show the use of the PTRANS specification language in designing and prototyping compiler optimizations for parallel programs in terms of graph transformations. By expressing optimizations as rewrites on control flow graphs with temporal logic side conditions, PTRANS allows for a more direct expression of the logic behind transformations. The mathematical semantics of PTRANS are accompanied by an executable semantics, allowing us to run PTRANS specifications directly on program graphs. The executable semantics relies on an algorithm for finding satisfying models of first-order CTL formulae on a graph, which in combination with an SMT solver can efficiently find all possible locations at which a transformation applies. PTRANS, with its combination of abstract and executable semantics, lays the groundwork for a unified platform for specifying, testing, and verifying optimizations.

While we have implemented the executable semantics of PTRANS in F# with Z3 integration, we are also interested in developing it in the K Framework [13] for programming language specification. A K implementation of PTRANS could take advantage of built-in state-space search functionality, as well as the wide range of languages that have been given formal semantics in K, including C, OCaml, and a fuller version of LLVM. We also intend to move forward with the formal verification of optimizations specified in PTRANS in the Isabelle theorem prover [12], and ultimately hope to link our executable semantics with our abstract semantics through a formal soundness proof.

Acknowledgements 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.

References

[1] Jürgen Bohn, Werner Damm, Orna Grumberg, Hardi Hungar & Karen Laster (1998): *First-Order-CTL Model Checking*. In Vikraman Arvind & Sundar Ramanujam, editors: *Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science* 1530, Springer Berlin Heidelberg, pp. 283–294, doi:10.1007/978-3-540-49382-2_27.

[2] E. M. Clarke, E. A. Emerson & A. P. Sistla (1986): *Automatic verification of finite-state concurrent systems using temporal logic specifications*. *ACM Trans. Program. Lang. Syst.* 8, pp. 244–263, doi:10.1145/5397.5399.

[3] He Jifeng & Jonathan Bowen (1994): *Specification, Verification and Prototyping of an Optimized Compiler*. *Formal Aspects of Computing* 6(6), pp. 643–658, doi:10.1007/BF03259390.

[4] Sara Kalvala, Richard Warburton & David Lacey (2009): *Program transformations using temporal logic side conditions*. *ACM Trans. Program. Lang. Syst.* 31(4), pp. 1–48, doi:10.1145/1516507.1516509.

[5] Jens Krinke (2003): *Context-sensitive slicing of concurrent programs*. *SIGSOFT Softw. Eng. Notes* 28(5), pp. 178–187, doi:10.1145/949952.940096.

[6] David Lacey, Neil D. Jones, Eric Van Wyk & Carl Christian Frederiksen (2002): *Proving correctness of compiler optimizations by temporal logic*. *SIGPLAN Not.* 37(1), pp. 283–294, doi:10.1145/565816.503299.

[7] C. Lattner & V. Adve (2004): *LLVM: a compilation framework for lifelong program analysis transformation*. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, doi:10.1109/CGO.2004.1281665.

[8] Sorin Lerner, Todd Millstein & Craig Chambers (2003): *Automatically proving the correctness of compiler optimizations*. *SIGPLAN Not.* 38, pp. 220–231, doi:10.1145/780822.781156.

[9] Xavier Leroy (2009): *A Formally Verified Compiler Back-end*. *J. Autom. Reason.* 43(4), pp. 363–446, doi:10.1007/s10817-009-9155-4.

[10] Robin Morisset, Pankaj Pawan & Francesco Zappa Nardelli (2013): *Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model*. *SIGPLAN Not.* 48(6), pp. 187–196, doi:10.1145/2499370.2491967.

[11] Leonardo Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer Berlin Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[12] Lawrence C. Paulson (1993): *Isabelle: The Next 700 Theorem Provers*. *CoRR* cs.LO/9301106. Available at `http://arxiv.org/abs/cs.LO/9301106`.

[13] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[14] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan & Peter Sewell (2011): *Relaxed-memory concurrency and verified compilation*. *SIGPLAN Not.* 46(1), pp. 43–54, doi:10.1145/1925844.1926393.

[15] Don Syme, Adam Granicz & Antonio Cisternino (2012): *Expert F# 3.0*, 3rd edition. Apress, Berkely, CA, USA, doi:10.1007/978-1-4302-4651-0.

[16] Xuejun Yang, Yang Chen, Eric Eide & John Regehr (2011): *Finding and understanding bugs in C compilers*. *SIGPLAN Not.* 46(6), pp. 283–294, doi:10.1145/1993316.1993532.