The PTRANS Specification Language

William Mansky

February 1, 2014

Abstract

The VeriF-OPT project seeks to provide a framework for stating and reasoning about compiler optimizations and transformations on parallel programs in the presence of relaxed memory models. The core of the framework is a domain-specific language for specifying compiler optimizations: PTRANS, in which program transformations are expressed as rewrites on control flow graphs with temporal logic side conditions. This document describes the syntax of PTRANS and its two semantics: the abstract semantics used to verify specifications, and the executable semantics used to prototype specifications.

1 PTRANS Syntax

The basic approach of the PTRANS specification language is that set out by Kalvala et al. in TRANS [1]: 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. Our starting point is our previous formalization of the syntax and semantics of TRANS for sequential programs [3]. As with our previous work, all formalizations and proofs have been developed in the Isabelle theorem prover [4], allowing us to provide strong guarantees of correctness for our verified optimizations.

The basic units of rewriting in PTRANS 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. Kalvala et al. have shown that a wide variety of common program transformations can be expressed using these basic rewrites.

The arguments to actions are given as *literals*, expressions representing graph objects but potentially containing *metavariables* that bind to values when the action is applied. A *node literal* is defined by:

$$\mathit{node\_lit} ::= \mathsf{Start}\ t \mid \mathsf{Exit}\ t \mid v$$

where t is a thread name and v is a (eventually node-valued) metavariable. An *edge type literal* is either a concrete edge type or a metavariable representing an edge type. Finally, a *pattern* represents a program instruction. In general we expect that patterns look like instructions with metavariables in place of various atomic elements (program variables, constants, operators, etc.), but in practice patterns are defined separately for each target language. In the grammar for actions, n and m are node literals, ℓ is an edge type literal, and $p, p_1, ...$ are patterns.

The side conditions in PTRANS are given in 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. Earlier presentations of TRANS 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. 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 $\mathcal{B}$ ("back-to") operators are the past-time counterparts to the $\mathcal{U}$ ("until") operators; for instance, $E\ \varphi_1\ \mathcal{B}\ \varphi_2$ asserts that there exists some path back 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 $\mathcal{U}$ operators in the usual way. Note the presence of the existential quantifier $\exists$, which is used to quantify over metavariables in a formula. These metavariables may then appear in the atomic predicates of a formula (atomic predicates, like patterns, are provided by the particular language under consideration).

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

$$T ::= A_1, ..., A_m \text{ if } \varphi \mid \text{MATCH } \varphi \text{ IN } T \mid T \text{ THEN } T \mid T \,\square\, T \mid \text{APPLY\_ALL } T$$

$A_1, ..., A_m$ 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 3.1). 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 $\square$ 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 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 [2], 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). Formally:

Definition 1. *A CFG is a labeled directed graph (N, E, I) where N is a set of nodes including a dedicated Start and Exit node, $E : 2^{N \times N \times ty}$ is a set of edges labeled with edge types, and $L : N \to instr$ is a labeling of the nodes with program instructions. Additionally, the outgoing edges of each node must correspond properly to the instruction label at that node. 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 a second parameter that must be provided by the target language: a correspondence between instruction labels and 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.

3 Verification Semantics

3.1 Temporal Logic on tCFGs

Suppose we have a notion of infinite paths through control flow graphs, in the form of a function *Paths* that takes a tCFG and a state (i.e., a vector of nodes, one in each CFG in the tCFG) and returns the set of infinite paths produced by following edges forward from each of those points (and a corresponding function *RPaths* for paths backwards). Then the semantics of CTL formulae are given by the relation $\mathcal{G}, \sigma, q \models \varphi$, where $\mathcal{G}$ is a tCFG, σ a substitution of values for metavariables, q a state, and φ a CTL formula, as follows:

- $\mathcal{G}, \sigma, q \models p$ if p is true in q under σ in the semantics for p provided by the target language

- $\mathcal{G}, \sigma, q \models \varphi_1 \wedge \varphi_2$ if $\mathcal{G}, \sigma, q \models \varphi_1$ and $\mathcal{G}, \sigma, q \models \varphi_2$
- $\mathcal{G}, \sigma, q \models \neg\varphi$ if $\mathcal{G}, \sigma, q \not\models \varphi$
- $\mathcal{G}, \sigma, q \models A\ \varphi_1\ \mathcal{U}\ \varphi_2$ if $\forall \lambda \in \mathit{Paths}(\mathcal{G}, q).\ \exists i.\ \mathcal{G}, \sigma, \lambda_{[i,\infty)} \models \varphi_2 \wedge \forall j < i.\ \mathcal{G}, \sigma, \lambda_{[j,\infty)} \models \varphi_1$
- $\mathcal{G}, \sigma, q \models E\ \varphi_1\ \mathcal{U}\ \varphi_2$ if $\exists \lambda \in \mathit{Paths}(\mathcal{G}, q).\ \exists i.\ \mathcal{G}, \sigma, \lambda_{[i,\infty)} \models \varphi_2 \wedge \forall j < i.\ \mathcal{G}, \sigma, \lambda_{[j,\infty)} \models \varphi_1$
- $\mathcal{G}, \sigma, q \models A\ \varphi_1\ \mathcal{B}\ \varphi_2$ if $\forall \lambda \in \mathit{RPaths}(\mathcal{G}, q).\ \exists i.\ \mathcal{G}, \sigma, \lambda_{[i,\infty)} \models \varphi_2 \wedge \forall j < i.\ \mathcal{G}, \sigma, \lambda_{[j,\infty)} \models \varphi_1$
- $\mathcal{G}, \sigma, q \models A\ \varphi_1\ \mathcal{B}\ \varphi_2$ if $\exists \lambda \in \mathit{RPaths}(\mathcal{G}, q).\ \exists i.\ \mathcal{G}, \sigma, \lambda_{[i,\infty)} \models \varphi_2 \wedge \forall j < i.\ \mathcal{G}, \sigma, \lambda_{[j,\infty)} \models \varphi_1$
- $\mathcal{G}, \sigma, q \models \exists x.\ \varphi$ if $\exists o.\ \mathcal{G}, \sigma(x \mapsto o), q \models \varphi$

We write $\mathcal{G}, \sigma \models \varphi$ to abbreviate $\mathcal{G}, \sigma, \mathit{start\_points}(\mathcal{G}) \models \varphi$, where $\mathit{start\_points}(\mathcal{G})$ is the vector that for each CFG in $\mathcal{G}$ gives that CFG's Start node.

3.2 Actions and Transformations

The semantics of actions are 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). 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 $\mathcal{G}_t = (N_t, E_t, I_t)$; then the semantics of actions are then defined as follows:

- $[\![\mathsf{add\_edge}(n, m, \ell)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t, E_t \cup \{(\sigma(n), \sigma(m), \sigma(\ell))\}, I_t))$
- $[\![\mathsf{remove\_edge}(n, m, \ell)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t, E_t - \{(\sigma(n), \sigma(m), \sigma(\ell))\}, I_t))$
- $[\![\mathsf{replace}\ n\ \mathsf{with}\ ]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t - \{\sigma(n)\}, E_t - \{(a, b, \ell) \mid a = \sigma(n) \vee b = \sigma(n)\}, I_t))$ (special case added to allow replace to remove nodes)
- $[\![\mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t \cup \{n_2, ..., n_m\}, \{\mathit{remap\_succ}(\sigma(n), n_m, e) \mid e \in E\} \cup \{(n_i, n_{i+1}, \mathsf{seq}) \mid 1 < i < m\}, I_t + (n_1 \mapsto \sigma(p_1), ..., n_m \mapsto \sigma(p_m)))$ where $n_1 = \sigma(n)$ and $n_2, ..., n_m$ are new nodes not in $\mathcal{G}$, and *remap_succ* is defined below
- $[\![\mathsf{split\_edge}(n, m, \ell, p)]\!](\sigma, \mathcal{G}) = \mathcal{G}(t \mapsto (N_t \cup \{n'\}, E_t - \{(\sigma(n), \sigma(m), \sigma(\ell))\} \cup \{(\sigma(n), n', \ell), (n', \sigma(m), \mathsf{seq})\}, I_t + (n' \mapsto \sigma(p))))$ where n' is a new node not in $\mathcal{G}$

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_m. To do this we use the auxiliary remap_succ function, defined as

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

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 sequence.

The semantics of strategies are 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:

$$\frac{}{\mathit{apply\_some}(T, \tau, G, G)} \qquad \frac{G' \in T(\tau, G) \quad \mathit{apply\_some}(T, \tau, G', G'')}{\mathit{apply\_some}(T, \tau, G, G'')}$$

Then the semantics of strategies are defined as follows:

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

- $[\![\text{MATCH } \varphi \text{ IN } T]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \exists \sigma.\ \sigma|_{\text{dom}(\tau)} = \tau \wedge \mathcal{G}, \sigma \models \varphi \wedge \mathcal{G}' \in [\![T]\!](\tau + \sigma|_{\text{fv}(\varphi)}, \mathcal{G})\}$ where $\text{fv}(\varphi)$ is the set of free variables of φ
- $[\![T_1 \text{ THEN } T_2]\!](\tau, \mathcal{G}) = \{\mathcal{G}'' \mid \exists \mathcal{G}'.\ \mathcal{G}' \in [\![T_1]\!](\tau, \mathcal{G}) \wedge \mathcal{G}'' \in [\![T_2]\!](\tau, \mathcal{G}')$
- $[\![T_1 \square T_2]\!](\tau, \mathcal{G}) = [\![T_1]\!](\tau, \mathcal{G}) \cup [\![T_2]\!](\tau, \mathcal{G})$
- $[\![\text{APPLY\_ALL } T]\!](\tau, \mathcal{G}) = \{\mathcal{G}' \mid \mathit{apply\_some}([\![T]\!], \tau, \mathcal{G}, \mathcal{G}')\} - \{\mathcal{G}' \mid \exists \mathcal{G}'' \neq \mathcal{G}'.\ \mathcal{G}'' \in [\![T]\!](\tau, \mathcal{G}')\}$

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.

4 Executable Semantics

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. As such, we need to modify the semantics in order to give an algorithm for computing the results of a transformation.

4.1 CTL Model Finding

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 $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, x, L)$ be a CFG, $v \in \mathcal{N}$ and φ a first-order CTL formula. Then $\{\sigma \mid \mathcal{G}; \sigma; v \vDash \varphi\} = \{\sigma \mid \sigma \vDash_{\text{FOL}} \mathit{Satis}(\varphi)(v)\}$.*

The Satis function relies on a collection of auxiliary functions for until-formulae:

- $\mathsf{Paths}_{\leftarrow}(I, F, 0, v) = F(v)$
- $\mathsf{Paths}_{\leftarrow}(I, F, n, v) = \mathsf{Paths}_{\leftarrow}(I, F, n-1, v) \vee \left(I(v) \wedge \bigvee_{v' \in \mathsf{succ}(\mathcal{E}, v)} \mathsf{Paths}_{\leftarrow}(I, F, n-1, v')\right)$
- $\mathsf{Paths}_{\wedge}(I, F, 0, v) = F(v)$
- $\mathsf{Paths}_{\wedge}(I, F, n, v) = \mathsf{Paths}_{\wedge}(I, F, n-1, v) \vee \left(I(v) \wedge \bigwedge_{v' \in \mathsf{succ}(\mathcal{E}, V)} \mathsf{Paths}_{\wedge}(I, F, n, v)\right)$
- $\mathsf{Paths}_{\rightarrow}(I, 0, v) = \text{true}$
- $\mathsf{Paths}_{\rightarrow}(I, n, v) = \bigvee_{v' \in \mathsf{succ}(\mathcal{E}, v)} (I(v') \wedge \mathsf{Paths}_{\rightarrow}(I, n-1, v'))$

Then Satis is defined by:

- $\mathsf{Satis}(p(\vec{x}))(v) = p(\vec{x})$
- $\mathsf{Satis}(\varphi_1 \wedge \varphi_2)(v) = \mathsf{Satis}(\varphi_1)(v) \wedge \mathsf{Satis}(\varphi_2)(v)$
- $\mathsf{Satis}(\neg\varphi)(v) = \neg\mathsf{Satis}(\varphi)(v)$
- $\mathsf{Satis}(\exists x.\ \varphi)(v) = \exists x.\ \mathsf{Satis}(\varphi)(v)$
- $\mathsf{Satis}(\text{E } \varphi_1 \text{ U } \varphi_2)(v) = \mathsf{Paths}_{\leftarrow}(\mathsf{Satis}(\varphi_2), \mathsf{Satis}(\varphi_1), |\mathcal{N}|, v)$
- $\mathsf{Satis}(\text{A } \varphi_1 \text{ U } \varphi_2)(v) = \neg\mathsf{Paths}_{\rightarrow}(\mathsf{Satis}(\varphi_1 \wedge \neg\varphi_2), |\mathcal{N}| + 1, v) \wedge \mathsf{Paths}_{\wedge}(\mathsf{Satis}(\varphi_2), \mathsf{Satis}(\varphi_1), |\mathcal{N}|, v)$
- $\mathsf{Satis}(\text{E } \varphi_1 \text{ B } \varphi_2)(v) = \mathsf{Paths}_{\not\leftarrow}(\mathsf{Satis}(\varphi_2), \mathsf{Satis}(\varphi_1), \mathcal{N}, v)$
- $\mathsf{Satis}(\text{A } \varphi_1 \text{ B } \varphi_2)(v) = \mathsf{Paths}_{\not\wedge}(\mathsf{Satis}(\varphi_2), \mathsf{Satis}(\varphi_1), \mathcal{N}, v)$

4.2 Executable PTRANS Semantics

Let $\mathsf{get\_models}(\tau, \mathcal{G}, \varphi)$ be the function 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 $\mathsf{get\_models}(\tau, \mathcal{G}, \varphi) = \{\sigma \mid \mathcal{G}; \sigma \vDash \varphi \wedge \sigma|_{\mathrm{dom}(\tau)} = \tau\}$, and so $\mathsf{get\_models}$ serves as an executable method for finding satisfying models of PTRANS side conditions. We can then write an executable function $\mathsf{trans\_sf}$ that finds the semantics of a transformation, defined as follows:

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

In order to define $\mathsf{trans\_sf}$ 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 $\mathsf{apply\_some}$ 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; thus, if T could be applied an infinite number of times to a graph $\mathcal{G}$, then $[\![\text{APPLY\_ALL } T]\!](\tau, \mathcal{G})$ would be empty. The $\mathsf{trans\_sf}$ function, on the other hand, attempts 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 $\mathsf{trans\_sf}$ is a viable executable semantics for PTRANS transformations.

5 Sample Atomic Predicates

In general, the atomic predicates available for use in our side conditions are another parameter provided by the target language under consideration; there is no standard set of atomic predicates. To provide some intuition for what atomic predicates might look like, we present here a simple, fairly general set of predicates. 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 $\mathcal{G}_t$.

State-independent predicates include:

- $\mathsf{conlit}(e)$, which is true when e represents a program constant (i.e., an integer).
- $\mathsf{varlit}(e)$, which is true when e represents a program variable.
- $\mathsf{freevar}(x, p)$, which is true when x is a free variable in the instruction p.
- $\mathsf{is}(x, y)$, which is true when the metavariables x and y represent the same program object (number, node, instruction, etc.).
- $\mathsf{sameval}(e, f)$, which is true when e and f are expressions that can be statically evaluated to the same value. (Note that $\neg\mathsf{sameval}(e, f)$ does not imply that e and f have different values; it simply means that they cannot be shown to have the same value at compile time.)
- $\mathsf{fresh}(e)$, which is true when e represents a program variable that appears nowhere in $\mathcal{G}$.

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] Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Trans. Program. Lang. Syst. 31(4), 1–48 (2009)

[2] Krinke, J.: Context-sensitive slicing of concurrent programs. SIGSOFT Softw. Eng. Notes 28(5), 178–187 (Sep 2003), http://doi.acm.org/10.1145/949952.940096

[3] Mansky, W., Gunter, E.: A framework for formal verification of compiler optimizations. In: Proceedings of the First international conference on Interactive Theorem Proving. pp. 371–386. ITP'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14052-5_26

[4] Paulson, L.C.: Isabelle: The next 700 theorem provers. In: Odifreddi, P. (ed.) Logic and Computer Science, pp. 361–386. Academic Press (1990)