Tracing Compilation by Abstract Interpretation

Stefano Dissegna
Dipartimento di Matematica
University of Padova, Italy

Francesco Logozzo
Microsoft Research
Redmond, WA, USA

Francesco Ranzato
Dipartimento di Matematica
University of Padova, Italy

Abstract

Tracing just-in-time compilation is a popular compilation schema for the efficient implementation of dynamic languages, which is commonly used for JavaScript, Python, and PHP. It relies on two key ideas. First, it monitors the execution of the program to detect so-called hot paths, *i.e.*, the most frequently executed paths. Then, it uses some store information available at runtime to optimize hot paths. The result is a residual program where the optimized hot paths are guarded by sufficient conditions ensuring the equivalence of the optimized path and the original program. The residual program is persistently mutated during its execution, *e.g.*, to add new optimized paths or to merge existing paths. Tracing compilation is thus fundamentally different than traditional static compilation. Nevertheless, despite the remarkable practical success of tracing compilation, very little is known about its theoretical foundations.

We formalize tracing compilation of programs using abstract interpretation. The monitoring (*viz.*, hot path detection) phase corresponds to an abstraction of the trace semantics that captures the most frequent occurrences of sequences of program points together with an abstraction of their corresponding stores, *e.g.*, a type environment. The optimization (*viz.*, residual program generation) phase corresponds to a transform of the original program that preserves its trace semantics up to a given observation as modeled by some abstraction. We provide a generic framework to express dynamic optimizations and to prove them correct. We instantiate it to prove the correctness of dynamic type specialization. We show that our framework is more general than a recent model of tracing compilation introduced in POPL 2011 by Guo and Palsberg (based on operational bisimulations). In our model we can naturally express hot path reentrance and common optimizations like dead-store elimination, which are either excluded or unsound in Guo and Palsberg's framework.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – correctness proofs, formal methods; D.3.4 [*Programming Languages*]: Processors – compilers, optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages – program analysis

Keywords tracing compilation; abstract interpretation; trace semantics

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535866

1. Introduction

Efficient traditional static compilation of popular dynamic languages like JavaScript, Python and PHP is very hard if not impossible. In fact those languages present so many dynamic features which make all traditional static analyses used for program optimization very imprecise. Therefore, practical implementations of dynamic languages should rely on dynamic information in order to produce an optimized version of the program. In particular, tracing just-in-time compilation (TJITC) [1, 3–6, 15, 16, 24] has emerged as a valuable implementation and optimization technique for dynamic languages. For instance, the Facebook HipHop virtual machine for PHP and the V8 JavaScript engine of Google Chrome use some form of tracing compilation [19, 20]. The Mozilla Firefox JavaScript engine used to have a tracing engine, TraceMonkey, which has been later substituted by whole-method just-in-time compilation engines (initially JägerMonkey and then IonMonkey) [13, 14].

1.1 The Problem

Tracing JIT compilers leverage runtime profiling of programs to detect and record often executed paths, called hot paths, and then they optimize and compile only these paths at runtime. A path is a linear sequence of instructions through the program. Profiling may also collect information about the values that the program variables may assume during the execution of that path, which is then used to specialize/optimize the code. Of course, this information is not guaranteed to hold for all the subsequent executions of the hot path. Since optimizations rely on that information, the hot path is augmented with guards that check the profiled conditions, such as, for example, variable types. When a guard fails, execution jumps back to the old, non-optimized code. The main hypotheses of tracing compilers, confirmed by the practice, are: (i) loop bodies are the only interesting code to optimize, so they only consider paths inside program loops; and (ii) optimizing straight-line code is easier than a whole-method analysis (involving loops, goto, *etc.*).

Hence, tracing compilers look quite different than traditional compilers. These differences raise some natural questions on trace compilation: (i) what is a viable formal model, which is generic yet realistic enough to capture the behavior of real optimizers? (ii) which optimizations are sound? (iii) how can one prove their soundness? In this paper we answer the above questions.

Our formal model is based on program trace semantics [9] and abstract interpretation [10, 12]. Hot path detection is modeled just as an abstraction of the trace semantics of the program, which only retains: (i) the sequences of program points which are repeated more than some threshold; (ii) an abstraction of the possible program stores, *e.g.*, the type of the variables instead of their concrete values. As a consequence, a hot path does not contain loops nor join points. Furthermore, in the hot path, all the correctness conditions (*i.e.*, guards) are explicit, for instance before performing integer addition, we should check that the operands are integers. If the guard condition is not satisfied then the execution leaves the hot path, reverting to the non-optimized code. Guards are essentially elements of some abstract domain, which is then left as a parameter in our framework. The hot path is then optimized using standard compilation techniques — we only require the optimization to be sound.

We define the correctness of the residual (or extracted) program in terms of abstraction of the trace semantics: the residual program is correct if it is indistinguishable, up to some abstraction of the trace semantics, from the original program. Examples of abstractions are the program store at the exit of a method, or the stores at loop entry and loop exit points.

1.2 Main Contributions

This paper puts forward a formal model of TJITC whose key features are as follows:

– We provide the first model of tracing compilation based on abstract interpretation of trace semantics of programs.

– We provide a more general and realistic framework than a recent model of TJITC by Guo and Palsberg [17] based on program bisimulations: we employ a less restrictive correctness criterion that enables the correctness proof of actually implemented optimizations; hot paths can be annotated with runtime information on the stores, notably type information; optimized hot loops can be re-entered.

– We formalize and prove the correctness of type specialization of hot paths.

Our model focusses on source-to-source program transformations and optimizations of a low level imperative language with untyped global variables, which may play the role of intermediate language of some virtual machine. Our starting point is that program optimizations can be seen as transformations that lose some information on the original program, so that optimizations can be viewed as approximations and in turn can be formalized by abstract interpretation. More precisely, we rely on the insight by Cousot and Cousot [12] that a program source can be seen as an abstraction of its trace semantics, *i.e.* the set of all possible execution sequences, so that a source-to-source optimization can be viewed as an abstraction of a transform of the program trace semantics. In our model, soundness of program optimizations is defined as program equivalence w.r.t. an observational abstract interpretation of the program trace semantics. Here, an observational abstraction induces a correctness criterion by describing what is observable about program executions, so that program equivalence means that two programs are indistinguishable by looking only at their observable behaviors.

A crucial part of tracing compilation is the selection of the hot path(s) to optimize. Of course, this choice is made at runtime based on program executions, so it can be seen once again as an abstraction of trace semantics. Here, a simple trace abstraction selects cyclic instruction sequences, *i.e.* loop paths, that appear at least N times within a single execution trace. These instruction sequences are recorded together with some property of the values assumed by program variables at that point, which is represented as a value of a suitable store abstraction, which in general depends on the successive optimization.

A program optimization can be seen as an abstraction of a semantic transformation of program execution traces, as described by the Cousots in [12]. The advantage of this approach is that optimization properties, such as their soundness, are easier to prove at a semantic level. The optimization itself can be defined on the whole program or, as in the case of real tracing JIT compilers, can be restricted to the hot path. This latter restriction is achieved by transforming the original program so that the hot path is extracted, *i.e.* made explicit: the hot path is added to the program as a path with no join points that jumps back to the original code when execution leaves it. A guard is placed before each command in this hot path that checks if the necessary conditions, as selected by the store abstraction, are satisfied. A program optimization can be then confined to the hot path only, making it linear, by ignoring the parts of the program outside it. The guards added to the hot path allows us to retain precision.

We apply our TJITC model to type specialization. Type specialization is definitely the key optimization for dynamic languages such as Javascript [15], as they provide generic operations whose execution depends on the type of run-time values of their operands.

1.3 Related Work

A formal model for tracing JIT compilation has been put forward in POPL 2011 by Guo and Palsberg [17]. It is based on operational bisimulation [23] to describe equivalence between source and optimized programs. In Section 11 we show how this model can be expressed within our framework through the following steps: Guo and Palsberg's language is compiled into ours; we then exhibit an observational abstraction which is equivalent to Guo and Palsberg's correctness criterion; finally, after some minor changes that address a few differences in path selection, the transformations performed on the source program turn out to be the same. Our framework overcomes some significant limitations in Guo and Palsberg's model. The bisimulation equivalence model used in [17] implies that the optimized program has to match every change to the store made by the original program, whereas in practice we only need this match to hold in certain program points and for some variables, such as in output instructions. This limits the number of real optimizations that can be modeled in the theoretical framework. For instance, dead store elimination is proven unsound in [17], while it is implemented in actual tracing compilers [15, Section 5.1]. Furthermore, their formalization fails to model some important features of actual TJITC implementation: (i) traces are simple linear paths of instructions, *i.e.*, they cannot be annotated with store properties; (ii) hot path selection is completely non-deterministic, they do not model a selection criterion; and, (iii) once execution leaves an optimized hot path the program will not be able to re-enter it.

It is also worth citing that abstract interpretation of program trace semantics roots at the foundational work by Cousot [8, 9] and has been widely used as a successful technique for defining a range of static program analyses [2, 7, 18, 22, 26–28]. Abstract interpretation has been used to describe static compilation and optimizations. In particular, Rival [25] describes various optimizations as trace abstractions they preserve. In the Cousot and Cousot terminology [12], Rival approach corresponds to offline transformations whereas tracing compilation is an online transformation.

2. Language and Concrete Semantics

2.1 Syntax

Following [12], we consider a basic low level language with untyped global variables, a kind of elementary dynamic language. Program commands range in $\mathbb{C}$ and consist of a labeled action which specifies a next label (Ł is the undefined label, where the execution becomes stuck).

$$\begin{aligned}
\text{Labels: } & L \in \mathbb{L} \qquad \text{Ł} \notin \mathbb{L} \\
\text{Values: } & v \in \text{Value} \\
\text{Variables: } & x \in \text{Var} \\
\text{Expressions: } & \text{Exp} \ni E ::= v \mid x \mid E_1 + E_2 \\
\text{Boolean Expressions: } & \text{BExp} \ni B ::= \text{tt} \mid \text{ff} \mid E_1 \leq E_2 \mid \\
& \qquad \neg B \mid B_1 \wedge B_2 \\
\text{Actions: } & \mathbb{A} \ni A ::= x := E \mid B \mid \text{skip} \\
\text{Commands: } & \mathbb{C} \ni C ::= L : A \rightarrow L' \quad (L' \in \mathbb{L} \cup \{\text{Ł}\})
\end{aligned}$$

For any command $C \equiv L : A \to L'$, we use the following notation: $lbl(C) \triangleq L$, $act(C) \triangleq A$ and $suc(C) \triangleq L'$. Commands $L : B \to L'$ whose action is a Boolean expression are called conditionals. A program $P \in \wp(\mathbb{C})$ is a (possibly infinite, at least in theory) set of commands, with a distinct initial label L_{in} from which execution starts, so that P_{in} denotes the commands in P labeled by L_{in} (P_{in} consists of two commands when the initial command is a conditional). If a program P includes a conditional $C \equiv L : B \to L'$ then P must also include a unique complement conditional $L : \neg B \to L''$, which is denoted by $cmpl(C)$ or C^c, where $\neg\neg B$ is taken to be equal to B, so that $cmpl(cmpl(C)) = C$. For simplicity, we consider deterministic programs, *i.e.*, we require that for any $C_1, C_2 \in P$ such that $lbl(C_1) = lbl(C_2)$: (1) if $act(C_1) \neq act(C_2)$ then $C_1 = cmpl(C_2)$; (2) if $act(C_1) = act(C_2)$ then $C_1 = C_2$. The set of well-formed programs is denoted by Program.

2.2 Transition Semantics

The language semantics relies on the following type values, where Char is a nonempty set of characters and $undef$ represents a generic error.

$$\text{Int} \triangleq \mathbb{Z} \quad \text{Bool} \triangleq \{true, false\}$$
$$\text{String} \triangleq \text{Char}^* \quad \text{Undef} \triangleq \{undef\}$$

In turn, Value and type names in Types are defined as follows:

$$\text{Value} \triangleq \text{Int} \cup \text{String} \cup \text{Undef}$$
$$\text{Types} \triangleq \{\text{Int}, \text{String}, \text{Undef}, \text{Any}, \varnothing\}$$

while $type$: Value $\to$ Types provides the type of any value. Here, the type name Any plays the role of top type, which is the supertype (*i.e.*, contains) all types, while $\varnothing$ is the bottom type, which is a subtype for all types.

Let $\rho \in \text{Store} \triangleq \text{Var} \to \text{Value}$ denote the set of possible program stores. The semantics of expressions and program actions is standard and goes as defined in Fig. 1. Let us remark that: the binary function $+_{\text{Int}}$ denotes integer addition; $\cdot$ is string concatenation; logical negation and conjunction are extended in order to handle $undef$ values, *i.e.*, $\neg undef = undef$ and $undef \wedge b = undef = b \wedge undef$. With a slight abuse of notation we also consider the so-called collecting versions of the semantic functions in Fig. 1:

$$\begin{array}{l}\mathbf{E} : \text{Exp} \to \wp(\text{Store}) \to \wp(\text{Value}) \\ \mathbf{E}[\![E]\!]S \triangleq \{\mathbf{E}[\![E]\!]\rho \mid \rho \in S\}\end{array}$$

$$\begin{array}{l}\mathbf{B} : \text{BExp} \to \wp(\text{Store}) \to \wp(\text{Store}) \\ \mathbf{B}[\![B]\!]S \triangleq \{\rho \in S \mid \mathbf{B}[\![B]\!]\rho = true\}\end{array}$$

$$\begin{array}{l}\mathbf{A} : \mathbb{A} \to \wp(\text{Store}) \to \wp(\text{Store}) \\ \mathbf{A}[\![A]\!]S \triangleq \{\mathbf{A}[\![A]\!]\rho \mid \rho \in S, \mathbf{A}[\![A]\!]\rho \notin \{\bot, undef\}\}\end{array}$$

Program states are pairs of stores and commands: State $\triangleq$ Store $\times \mathbb{C}$. If P is a program then $\text{State}_P \triangleq \text{Store} \times P$. We extend lbl, act and suc to be defined on states, meaning that they are defined on the command component of a state. Also, $store(s)$ returns the store of a state s. Given $P \in$ Program, the program transition relation $\mathbf{S}[\![P]\!] : \text{State}_P \to \wp(\text{State}_P)$ between states is defined as follows:

$$\begin{aligned}\mathbf{S}[\![P]\!]\langle\rho, C\rangle \triangleq \{\langle\rho', C'\rangle \in \text{State}_P \mid\ & \rho' \in \mathbf{A}[\![act(C)]\!]\{\rho\}, \\ & C' \in P,\ suc(C) = lbl(C')\}.\end{aligned}$$

It is worth remarking that, according to the above definition, if $C \equiv L : A \to L'$, $C_1 \equiv L' : B \to L''$ and $C_1^c \equiv L' : \neg B \to L'''$ are all commands that belong to P and $\rho' \in \mathbf{A}[\![A]\!]\rho$ then we have that $\mathbf{S}[\![P]\!]\langle\rho, C\rangle = \{\langle\rho', C_1\rangle, \langle\rho', C_1^c\rangle\}$.

2.2.1 Trace Semantics

A partial (forward) trace is a finite sequence of program states which are related by the transition relation $\mathbf{S}$. If P is a program then we define

$$\text{Trace}_P \triangleq \{\sigma \in (\text{State}_P)^* \mid \forall i \in [1, |\sigma|).\ \sigma_i \in \mathbf{S}[\![P]\!]\sigma_{i-1}\}$$

A trace $\sigma \in \text{Trace}_P$ is maximal if for any state $s \in \text{State}_P$, $\sigma s \notin \text{Trace}_P$. Let us note that according to the above definitions, if a trace $\sigma \in \text{Trace}_P$ has a last state $\sigma_{|\sigma|-1} = \langle\rho, L : B \to L'\rangle$ with a conditional command such that $\mathbf{B}[\![B]\!]\rho = false$ then σ is maximal. Also, if a trace $\sigma \in \text{Trace}_P$ has a last state $\sigma_{|\sigma|-1} = \langle\rho, L : A \to \text{Ł}\rangle$ whose next label is the undefined label then σ is maximal as well.

The trace semantics $\mathbf{T}[\![P]\!]$ is the set of all the partial (including maximal) traces of the program P. This set is defined as the least fixed point of a monotonic operator $F[P] : \wp(\text{Trace}_P) \to \wp(\text{Trace}_P)$, called trace transition operator, defined as follows:

$$\begin{aligned} F[P]X &\triangleq \{\langle\rho, C_{in}\rangle \mid \rho \in \text{Store}, C_{in} \in P_{in}\} \cup \\ &\quad\ \{\sigma s s' \in \text{Trace}_P \mid \sigma s \in X,\ s' \in \mathbf{S}[\![P]\!]s\} \\ \mathbf{T}[\![P]\!] &\triangleq \text{lfp}(F[P]) \in \wp(\text{Trace}_P)\end{aligned}$$

The function $F[P]$ is trivially monotone on the complete lattice $\langle\wp(\text{Trace}_P), \subseteq\rangle$, so that its least fixpoint $\mathbf{T}[\![P]\!]$ is well defined.

Example 2.1. Let us consider the program below written in some while-language:

```
$x := 0;$
**while** $(x \leq 20)$ **do**
    $x := x + 1;$
    **if** $(x\%3 = 0)$ **then** $x := x + 3;$
```

Its translation in our language is given below, where, with a little abuse, we assume that the syntax of arithmetic and Boolean expression is extended to allow expressions like $x\%3 = 0$.

$$\begin{array}{l} P = \big\{C_0 \equiv L_0 : x := 0 \to L_1, \\ \quad C_1 \equiv L_1 : x \leq 20 \to L_2,\ C_1^c \equiv L_1 : \neg(x \leq 20) \to L_5, \\ \quad C_2 \equiv L_2 : x := x + 1 \to L_3, \\ \quad C_3 \equiv L_3 : (x\%3 = 0) \to L_4,\ C_3^c \equiv L_3 : \neg(x\%3 = 0) \to L_1 \\ \quad C_4 \equiv L_4 : x := x + 3 \to L_1, \\ \quad C_5 \equiv L_5 : \text{skip} \to \text{Ł}\big\} \end{array}$$

The trace semantics $\mathbf{T}[\![P]\!]$ includes the following partial traces, where ? stands for any integer value and stores are denoted within square brackets.

$$\begin{array}{l}
\langle[x/?], C_0\rangle \quad \in F[P]\varnothing \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle \\
\langle[x/?], C_0\rangle\langle[x/0], C_1^c\rangle \quad \text{(maximal)} \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3\rangle \quad \text{(maximal)} \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle\langle[x/1], C_1\rangle \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle\langle[x/1], C_1^c\rangle \quad \text{(maximal)} \\
\langle[x/?], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle\langle[x/1], C_1\rangle\langle[x/2], C_2\rangle \\
\ldots \\
\ldots \\
\langle[x/?], C_0\rangle \cdots \langle[x/21], C_4\rangle\langle[x/24], C_1\rangle \quad \text{(maximal)} \\
\langle[x/?], C_0\rangle \cdots \langle[x/21], C_4\rangle\langle[x/24], C_1^c\rangle\langle[x/24], C_5\rangle \quad \text{(maximal)}
\end{array}$$

$\square$

$$\mathbf{E} : \text{Exp} \to \text{Store} \to \text{Value}$$

$$\mathbf{E}[\![v]\!]\rho \triangleq v \quad \mathbf{E}[\![x]\!]\rho \triangleq \rho(x) \quad \mathbf{E}[\![E_1 + E_2]\!]\rho \triangleq \begin{cases} \mathbf{E}[\![E_1]\!]\rho +_{\text{Int}} \mathbf{E}[\![E_2]\!]\rho & \text{if } \mathit{type}(\mathbf{E}[\![E_i]\!]\rho) = \text{Int} \\ \mathbf{E}[\![E_1]\!]\rho \cdot \mathbf{E}[\![E_2]\!]\rho & \text{if } \mathit{type}(\mathbf{E}[\![E_i]\!]\rho) = \text{String} \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$\mathbf{B} : \text{BExp} \to \text{Store} \to \{\mathit{true}, \mathit{false}, \mathit{undef}\}$$

$$\mathbf{B}[\![\,\text{tt}\,]\!]\rho \triangleq \mathit{true} \quad \mathbf{B}[\![\text{ff}]\!]\rho \triangleq \mathit{false} \quad \mathbf{B}[\![E_1 \le E_2]\!]\rho \triangleq \begin{cases} \mathbf{E}[\![E_1]\!]\rho \le \mathbf{E}[\![E_2]\!]\rho & \text{if } \mathit{type}(\mathbf{E}[\![E_i]\!]\rho) = \text{Int} \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$\mathbf{B}[\![\neg B]\!]\rho \triangleq \neg\mathbf{B}[\![B]\!]\rho \quad \mathbf{B}[\![B_1 \wedge B_2]\!]\rho \triangleq \mathbf{B}[\![B_1]\!]\rho \wedge \mathbf{B}[\![B_2]\!]\rho$$

$$\mathbf{A} : \mathbb{A} \to \text{Store} \to \text{Store} \cup \{\bot, \mathit{undef}\}$$

$$\mathbf{A}[\![x := E]\!]\rho \triangleq \rho[x := \mathbf{E}[\![E]\!]\rho] \quad \mathbf{A}[\![\text{skip}]\!]\rho \triangleq \rho \quad \mathbf{A}[\![B]\!]\rho \triangleq \begin{cases} \rho & \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{true} \\ \bot & \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{false} \\ \mathit{undef} & \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{undef} \end{cases}$$

Figure 1. Semantics of program expressions and actions.

3. Abstract Interpretation Background

In standard abstract interpretation [10, 11], abstract domains (or abstractions) are specified by Galois connections/insertions (GCs/GIs for short) or, equivalently, adjunctions. Concrete and abstract domains, $\langle C, \le_C\rangle$ and $\langle A, \le_A\rangle$, are assumed to be complete lattices which are related by abstraction and concretization maps $\alpha : C \to A$ and $\gamma : A \to C$ that give rise to an adjunction (α, C, A, γ), that is, for all a and c, $\alpha(c) \le_A a \Leftrightarrow c \le_C \gamma(a)$. A GC is a GI when $\alpha \circ \gamma = \lambda x.x$. It is well known that a join-preserving α uniquely determines γ as follows: $\gamma(a) = \vee\{c \in C \mid \alpha(c) \le_A a\}$; conversely, a meet-preserving γ uniquely determines α as follows: $\alpha(c) = \wedge\{a \in A \mid c \le_C \gamma(a)\}$.

Let $f : C \to C$ be some concrete monotone function — for simplicity, we consider 1-ary functions — and let $f^\sharp : A \to A$ be a corresponding monotone abstract function defined on some abstraction A related to C by a GC. Then, $f^\sharp$ is a correct abstract interpretation of f on A when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ holds, where $\sqsubseteq$ denotes the pointwise ordering. Moreover, the abstract function $f^A \triangleq \alpha \circ f \circ \gamma : A \to A$ is called the best correct approximation of f on A because any abstract function $f^\sharp$ is correct iff $f^A \sqsubseteq f^\sharp$. Hence, for any abstraction A, f^A plays the role of the best possible approximation of f on the abstract domain A.

4. Store Abstractions

As usual in abstract interpretation, a store property is modeled by some abstraction $\text{Store}^\sharp$ that we assume to be encoded through a Galois connection

$$(\alpha_{store}, \langle\wp(\text{Store}), \subseteq\rangle, \langle\text{Store}^\sharp, \le\rangle, \gamma_{store}).$$

For instance, as we will see later, the static types of program variables give rise to a simple store abstraction.

Given a program P, a store abstraction $\text{Store}^\sharp$ also induces a corresponding state abstraction $\text{State}^\sharp_P \triangleq \text{Store}^\sharp \times P$ and, in turn, a trace abstraction $\text{Trace}^\sharp_P \triangleq (\text{State}^\sharp_P)^*$.

4.1 Nonrelational Abstractions

Nonrelational store abstractions can be easily designed by a standard pointwise lifting of some value abstraction. Let $\text{Value}^\sharp$ be a value abstraction as formalized by a Galois connection

$$(\alpha_{value}, \langle\wp(\text{Value}), \subseteq\rangle, \langle\text{Value}^\sharp, \le_{\text{Value}^\sharp}\rangle, \gamma_{value}).$$

The abstract domain $\text{Value}^\sharp$ induces a nonrelational store abstraction

$$\text{Store}^\sharp_{value} \triangleq \langle\text{Var} \to \text{Value}^\sharp, \sqsubseteq\rangle$$

where $\sqsubseteq$ is the pointwise ordering induced by $\le_{\text{Value}^\sharp}$: $\rho^\sharp_1 \sqsubseteq \rho^\sharp_2$ iff for all $x \in \text{Var}$, $\rho^\sharp_1(x) \le_{\text{Value}^\sharp} \rho^\sharp_2(x)$. Here, the bottom and top abstract stores are, respectively, $\lambda x.\bot_{\text{Value}^\sharp}$ and $\lambda x.\top_{\text{Value}^\sharp}$. The abstraction map $\alpha^{\sqsubseteq}_{value} : \wp(\text{Store}) \to \text{Store}^\sharp_{value}$ is thus defined as follows:

$$\alpha^{\sqsubseteq}_{value}(S) \triangleq \lambda x.\alpha_{value}(\{\rho(x) \in \text{Value} \mid \rho \in S\})$$

while the corresponding concretization map $\gamma^{\sqsubseteq}_{value} : \text{Store}^\sharp_{value} \to \wp(\text{Store})$ is defined by adjunction from $\alpha^{\sqsubseteq}_{value}$ as recalled in Section 3 and it is easy to check that it turns out to be defined as follows:

$$\gamma^{\sqsubseteq}_{value}(\rho^\sharp) = \{\rho \in \text{Store} \mid \forall x \in \text{Var} \,.\, \rho(x) \in \gamma_{value}(\rho^\sharp(x))\}.$$

5. Hot Path Selection

A loop path is a sequence of program commands which is repeated in some execution of a program loop, together with a store property which is valid at the entry of each command in the path. A loop path becomes *hot* when, during the execution, it is repeated at least a fixed number N of times. In a TJITC, hot path selection is performed by a loop path monitor that also records store properties (see, *e.g.*, [15]). Here, hot path selection is not operationally defined, it is instead modeled as an abstraction map over program traces, *i.e.*, program executions.

We first define a mapping $\mathit{loop} : \text{Trace}_P \to \wp(\text{Trace}_P)$ that returns all the loop paths in some execution trace of a program P. Formally, a loop path is a substring (*i.e.*, a segment) τ of a trace σ such that: (1) the successor command in σ of the last state in τ exists and coincides with the command – or its complement, when this is the last loop iteration – of the first state in τ; (2) there is no other such command within τ (otherwise the sequence τ would contain multiple iterations); (3) the last state of τ performs a backward jump in the program P. To recognize backward jumps, we consider a topological order on the control flow graph of commands in P, here denoted by $\lessdot$:

$$\begin{aligned}&\mathit{loop}(\langle\rho_0,C_0\rangle\cdots\langle\rho_n,C_n\rangle)\triangleq\\ &\big\{\langle\rho_i,C_i\rangle\langle\rho_{i+1},C_{i+1}\rangle\cdots\langle\rho_j,C_j\rangle \mid 0\le i\le j<n,\ C_i\lessdot C_j,\\ &\quad \mathit{lbl}(C_{j+1})=\mathit{lbl}(C_i),\forall k\in(i,j].\,C_k\notin\{C_i,\mathit{cmpl}(C_i)\}\big\}.\end{aligned}$$

Let us remark that a loop path

$$\langle\rho_i,C_i\rangle\cdots\langle\rho_j,C_j\rangle\in\mathit{loop}(\langle\rho_0,C_0\rangle\cdots\langle\rho_n,C_n\rangle)$$

may contain some sub-loop path, namely it may happen that $\mathit{loop}(\langle\rho_i,C_i\rangle\cdots\langle\rho_j,C_j\rangle)\neq\varnothing$ so that some commands C_k, with $k\in[i,j]$, occur more than once in $\langle\rho_i,C_i\rangle\cdots\langle\rho_j,C_j\rangle$. We abuse notation by using α_{store} to denote a map $\alpha_{\mathit{store}}:\mathrm{Trace}_P\to\mathrm{Trace}^\sharp_P$ which "abstracts" program traces in $\mathrm{Trace}^\sharp_P$:

$$\begin{aligned}&\alpha_{\mathit{store}}(\langle\rho_0,C_0\rangle\cdots\langle\rho_n,C_n\rangle)\triangleq\\ &\qquad\langle\alpha_{\mathit{store}}(\{\rho_0\}),C_0\rangle\cdots\langle\alpha_{\mathit{store}}(\{\rho_n\}),C_n\rangle.\end{aligned}$$

Given a static parameter $N>0$, we define a function

$$\mathit{hot}^N:\mathrm{Trace}_P\to\wp(\mathrm{Trace}^\sharp_P)$$

which returns the set of $\mathrm{Store}^\sharp$-abstracted loop paths appearing at least N times in some program trace. To count the number of times a loop path appears within a trace we use an auxiliary function $\mathit{count}:\mathrm{Trace}^\sharp_P\times\mathrm{Trace}^\sharp_P\to\mathbb{N}$ such that $\mathit{count}(\sigma,\tau)$ yields the number of times an abstract path τ occurs in a abstract trace σ:

$$\begin{aligned}&\mathit{count}(\langle a_0,C_0\rangle\cdots\langle a_n,C_n\rangle,\langle b_0,C'_0\rangle\cdots\langle b_m,C'_m\rangle)\triangleq\\ &\sum_{i=0}^{n-m}\begin{cases}1 & \text{if } \langle a_i,C_i\rangle\cdots\langle a_{i+m},C_{i+m}\rangle=\langle b_0,C'_0\rangle\cdots\langle b_m,C'_m\rangle\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

Hence, hot^N can be defined as follows:

$$\begin{aligned}&\mathit{hot}^N(\sigma\equiv\langle\rho_0,C_0\rangle\cdots\langle\rho_n,C_n\rangle)\triangleq\\ &\big\{\langle a_i,C_i\rangle\cdots\langle a_j,C_j\rangle \mid \exists\langle\rho_i,C_i\rangle\cdots\langle\rho_j,C_j\rangle\in\mathit{loop}(\sigma)\text{ s.t.}\\ &\ i\le j,\ \alpha_{\mathit{store}}(\langle\rho_i,C_i\rangle\cdots\langle\rho_j,C_j\rangle)=\langle a_i,C_i\rangle\cdots\langle a_j,C_j\rangle,\\ &\qquad\qquad \mathit{count}(\alpha_{\mathit{store}}(\sigma),\langle a_i,C_i\rangle\cdots\langle a_j,C_j\rangle)\ge N\big\}.\end{aligned}$$

Finally, an abstraction map $\alpha^N_{\mathit{hot}}:\wp(\mathrm{Trace}_P)\to\wp(\mathrm{Trace}^\sharp_P)$ collects the results of applying hot^N to a set of traces:

$$\alpha^N_{\mathit{hot}}(T)\triangleq\cup_{\sigma\in T}\,\mathit{hot}^N(\sigma).$$

A hot path $hp\in\alpha^N_{\mathit{hot}}(\mathbf{T}[\![P]\!])$ is also called a N-hot path and is compactly denoted as $hp=\langle a_0,C_0,...,a_n,C_n\rangle$. Let us observe that if the hot path is the body of some while loop then its first command C_0 is a conditional, namely C_0 is the Boolean guard of the while loop. We define the following successor function for indices in hot paths: $\mathit{next}\triangleq\lambda i.\ i=n\,?\,0:i+1$. For a N-hot path $\langle a_0,C_0,...,a_n,C_n\rangle\in\alpha^N_{\mathit{hot}}(\mathbf{T}[\![P]\!])$, for any $i\in[0,n]$, if C_i is a conditional command $L_i:B_i\to L_{\mathit{next}(i)}$ then throughout the paper its complement $C^c_i=\mathit{cmpl}(C_i)$ will be also denoted by $L_i:\neg B_i\to L^c_{\mathit{next}(i)}$.

Example 5.1. Let us consider the program P in Example 2.1. We consider a trivial one-point store abstraction $\mathrm{Store}^\sharp=\{\top\}$, where all the stores are abstracted to the same abstract store $\top$, *i.e.*, $\alpha_{\mathit{store}}=\lambda S.\top$. Here, we have two 2-hot paths in P, that is, it turns out that $\alpha^2_{\mathit{hot}}(\mathbf{T}[\![P]\!])=\{hp_1,hp_2\}$ where:

$$\begin{aligned}hp_1=\langle&\top,C_1\equiv L_1:x\le 20\to L_2,\\ &\top,C_2\equiv L_2:x:=x+1\to L_3,\\ &\top,C^c_3\equiv L_3:\neg(x\%3=0)\to L_1\rangle;\end{aligned}$$

$$\begin{aligned}hp_2=\langle&\top,C_1\equiv L_1:x\le 20\to L_2,\\ &\top,C_2\equiv L_2:x:=x+1\to L_3,\\ &\top,C_3\equiv L_3:(x\%3=0)\to L_4,\\ &\top,C_4\equiv x:=x+3\to L_1\rangle.\end{aligned}$$

□

6. Trace Extraction

For any abstract store $a\in\mathrm{Store}^\sharp$, a corresponding Boolean expression "guard E_a" $\in$ BExp is defined (where the notation E_a should hint at an expression which is induced by the abstract store a) , whose semantics is as follows: for any $\rho\in\mathrm{Store}$,

$$\mathbf{B}[\![\text{guard }E_a]\!]\rho\triangleq\rho\in\gamma_{\mathit{store}}(a).$$

Thus, in turn, we also have program actions guard E_a such that:

$$\mathbf{A}[\![\text{guard }E_a]\!]\rho\triangleq\begin{cases}\rho & \text{if }\rho\in\gamma_{\mathit{store}}(a)\\ \bot & \text{if }\rho\notin\gamma_{\mathit{store}}(a)\end{cases}$$

Let P be a program and $hp=\langle a_0,C_0,...,a_n,C_n\rangle\in\alpha^N_{\mathit{hot}}(\mathbf{T}[\![P]\!])$ be a hot path on some store abstraction $\mathrm{Store}^\sharp$. We define a synctatic transform of P where the hot path hp is explicitly extracted from P. This is implemented by a suitable relabeling of each command C_i in hp which is in turn preceded by the conditional guard E_{a_i} induced by the store property a_i. To this aim, we consider three *injective* relabeling functions

$$\begin{aligned}\ell&:[0,n]\to\mathbb{L}_1\\ \mathbb{1}&:[1,n]\to\mathbb{L}_2\\ \overline{(\cdot)}&:\mathbb{L}\to\overline{\mathbb{L}}\end{aligned}$$

where $\mathbb{L}_1$, $\mathbb{L}_2$ and $\overline{\mathbb{L}}$ are pairwise disjoint sets of fresh labels, so that $\mathit{labels}(P)\cap(\mathbb{L}_1\cup\mathbb{L}_2\cup\overline{\mathbb{L}})=\varnothing$. The transformed program $\mathit{extr}_{hp}(P)$ for the hot path hp is defined as follows and a graphical example of this transform is depicted in Fig. 2.

Definition 6.1 (Trace extraction transform). The *trace extraction transform* of P for the hot path hp is:

$$\begin{aligned}\mathit{extr}_{hp}(P)\triangleq P\smallsetminus&\big(\{C_0\}\cup\{\mathit{cmpl}(C_0)\mid\mathit{cmpl}(C_0)\in P\}\big)\\ \cup&\,\{\overline{L_0}:\mathit{act}(C_0)\to L_1\}\\ \cup&\,\{\overline{L_0}:\neg\mathit{act}(C_0)\to L^c_1\mid\mathit{cmpl}(C_0)\in P\}\\ \cup&\,\mathit{stitch}_P(hp)\end{aligned}$$

where the stitch of hp into P is defined as follows:

$$\begin{aligned}&\mathit{stitch}_P(hp)\triangleq\\ &\cup\{L_0:\text{guard }E_{a_0}\to\ell_0,\ L_0:\neg\text{guard }E_{a_0}\to\overline{L_0}\}\\ &\cup\{\ell_i:\mathit{act}(C_i)\to\mathbb{1}_{i+1}\mid i\in[0,n-1]\}\cup\{\ell_n:\mathit{act}(C_n)\to L_0\}\\ &\cup\{\ell_i:\neg\mathit{act}(C_i)\to L^c_{\mathit{next}(i)}\mid i\in[0,n],\ \mathit{cmpl}(C_i)\in P\}\\ &\cup\{\mathbb{1}_i:\text{guard }E_{a_i}\to\ell_i,\ \mathbb{1}_i:\neg\text{guard }E_{a_i}\to L_i\mid i\in[1,n]\}.\end{aligned}$$

□

The new command L_0 : guard $E_{a_0}\to\ell_0$ is therefore the entry conditional of the stitched hot path $\mathit{stitch}_P(hp)$, while any command $C\in\mathit{stitch}_P(hp)$ such that $\mathit{suc}(C)\in\mathit{labels}(P)\cup\overline{\mathbb{L}}$ is a potential exit (or bail out) command of $\mathit{stitch}_P(hp)$.

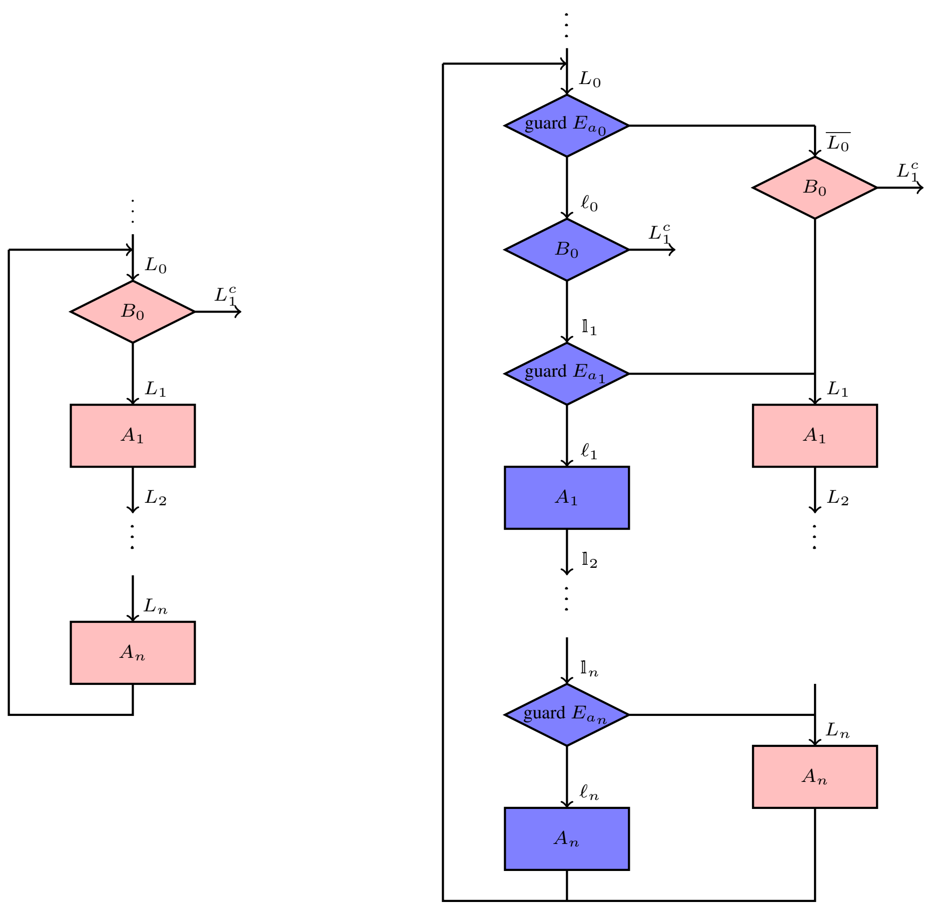


Figure 2. An example of trace extraction transform: on the left, a hot path hp with commands in pink (in black/white: loosely dotted) shapes; on the right, the corresponding trace transform $extr_{hp}(P)$ with new commands in blue (in black/white: densely dotted) shapes.

Lemma 6.2. *If P is well-formed then, for any hot path hp, $extr_{hp}(P)$ is well-formed.*

Let us remark that the stitch of the hot path hp into P is always a linear sequence of different commands, namely, $stitch_P(hp)$ does not contain loops nor join points. Furthermore, this happens even if the hot path hp does contain some inner sub-loop. Technically, this comes as a consequence of the fact that the above relabeling functions are required to be injective. Hence, even if some command C occurs more than once inside hp then these multiple occurrences of C in hp are transformed into differently labeled commands in $stitch_P(hp)$.

Example 6.3. Let us consider the program P in Example 2.1 and the hot path $hp_1 = \langle \top, C_1, \top, C_2, \top, C_3^c\rangle$ in Example 5.1, where stores are abstracted to the trivial one-point abstraction $\text{Store}^\sharp = \{\top\}$. Here, we have that for any store $\rho \in$ Store, $\mathbf{B}[\![\text{guard } E_\top]\!]\rho = \mathit{true}$. The trace extraction transform of P w.r.t. hp is therefore as follows:

$$\begin{aligned} extr_{hp}(P) = P \smallsetminus \{&C_1, C_1^c\} \\ &\cup \{\overline{L_1} : x \le 20 \to L_2,\ \overline{L_1} : \neg(x \le 20) \to L_5\} \\ &\cup stitch_P(hp) \end{aligned}$$

where $stitch_P(hp) =$

$$\begin{aligned} &\{H_0 \equiv L_1 : \text{guard } E_\top \to \ell_0,\ H_0^c \equiv L_1 : \neg\text{guard } E_\top \to \overline{L_1}\} \\ \cup\, &\{H_1 \equiv \ell_0 : x \le 20 \to \mathbb{l}_1,\ H_1^c \equiv \ell_0 : \neg(x \le 20) \to L_5\} \\ \cup\, &\{H_2 \equiv \mathbb{l}_1 : \text{guard } E_\top \to \ell_1,\ H_2^c \equiv \mathbb{l}_1 : \neg\text{guard } E_\top \to L_2\} \\ \cup\, &\{H_3 \equiv \ell_1 : x := x+1 \to \mathbb{l}_2\} \\ \cup\, &\{H_4 \equiv \mathbb{l}_2 : \text{guard } E_\top \to \ell_2,\ H_4^c \equiv \mathbb{l}_2 : \neg\text{guard } E_\top \to L_3\} \\ \cup\, &\{H_5 \equiv \ell_2 : \neg(x\%3 = 0) \to L_1, H_5^c \equiv \ell_2 : (x\%3 = 0) \to L_4\}. \end{aligned}$$

Hence, $extr_{hp}(P)$ can be rewritten at a high-level representation using while loops and gotos as follows:

```
$x := 0$;
$L_1$: **while** guard $E_\top$ **do**
        **if** $\neg(x \le 20)$ **then goto** $L_5$;
        **if** $\neg$guard $E_\top$ **then goto** $L_2$;
        $x := x + 1$;
        **if** $\neg$guard $E_\top$ **then goto** $L_3$;
        **if** $(x\%3 = 0)$ **then goto** $L_4$;
**if** $\neg(x \le 20)$ **then goto** $L_5$;
$L_2$: $x := x + 1$;
$L_3$: **if** $\neg(x\%3 = 0)$ **then goto** $L_1$;
$L_4$: $x := x + 3$;
**goto** $L_1$;
$L_5$: **skip**;
```

□

7. Correctness

As advocated by Cousot and Cousot [12, par. 3.8], correctness of dynamic program transformations and optimizations should be defined with respect to some observational abstraction of program trace semantics: a program transform is correct when, at some level of abstraction, the observation of the execution of the subject program is equivalent to the observation of the execution of the transformed program. The approach by Guo and Palsberg [17] basically relies on a notion of correctness that requires the same *store changes* in both the transformed/optimized program and the original program. This can be easily encoded by an observational abstraction $\alpha_{sc} : \wp(\text{Trace}_P) \to \wp(\text{Store}^*)$ of trace semantics that observes store changes in execution traces of a program P:

$$sc : \text{Trace}_P \to \text{Store}^*$$

$$sc(\sigma) \triangleq \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \rho & \text{if } \sigma = \langle \rho, C \rangle \\ sc(\langle \rho, C_1 \rangle \sigma') & \text{if } \sigma = \langle \rho, C_0 \rangle \langle \rho, C_1 \rangle \sigma' \\ \rho_0\, sc(\langle \rho_1, C_1 \rangle \sigma') & \text{if } \sigma = \langle \rho_0, C_0 \rangle \langle \rho_1, C_1 \rangle \sigma', \rho_0 \neq \rho_1 \end{cases}$$

$$\alpha_{sc}(T) \triangleq \{sc(\sigma) \mid \sigma \in T\}$$

Since α_{sc} obviously preserves arbitrary set unions, it admits a right adjoint $\gamma_{sc} : \wp(\text{Store}^*) \to \wp(\text{Trace}_P)$ defined as $\gamma_{sc}(S) \triangleq \cup\{T \in \wp(\text{Trace}_P) \mid \alpha_{sc}(T) \subseteq S\}$, that gives rise to a GC $(\alpha_{sc}, \langle \wp(\text{Trace}_P), \subseteq \rangle, \langle \wp(\text{Store}^*), \subseteq \rangle, \gamma_{sc})$.

However, the store changes abstraction α_{sc} may be too strong in practice. This condition can be thus relaxed and generalized to an observational abstraction that demands to have the same stores (possibly just for some subset of variables) only at some specific program points. For example, these program points may depend on the language. In a language without output primitives and functions, as that considered in [17], we could be interested just in the final store of the program (when it terminates), or in the entry and exit stores of any loop containing an extracted hot path. If a more general language includes a sort of primitive "put $\mathcal{X}$" that "outputs" the value of program variables ranging in some set $\mathcal{X}$ then we may want to have stores with the same values for variables in $\mathcal{X}$ at each output point. Moreover, the same sequence of outputs should be preserved, *i.e.* optimizations must not modify the order of output instructions.

We therefore consider an additional sort of actions: put $\mathcal{X} \in \mathbb{A}$, where $\mathcal{X} \subseteq \text{Var}$ is a set of program variables. The semantics of put $\mathcal{X}$ obviously does not affect program stores, *i.e.*, $\mathbf{A}[\![\text{put } \mathcal{X}]\!]\rho \triangleq \rho$. Correspondingly, an observational abstraction $\alpha_o : \wp(\text{Trace}_P) \to \wp(\text{Store}^*)$ of trace semantics observes program stores at output program points only (we use $\rho_{|\mathcal{X}}$ to denote store restriction to variables in $\mathcal{X}$):

$$out : \text{Trace}_P \to \text{Store}^*$$

$$out(\sigma) \triangleq \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ out(\sigma') & \text{if } \sigma = s\sigma' \wedge act(s) \neq \text{put } \mathcal{X} \\ \rho_{|\mathcal{X}}\ out(\sigma') & \text{if } \sigma = \langle \rho, L : \text{put } \mathcal{X} \to L' \rangle \sigma' \end{cases}$$

$$\alpha_o(T) \triangleq \{out(\sigma) \mid \sigma \in T\}$$

Similarly to α_{sc}, here again we have a GC $(\alpha_o, \langle \wp(\text{Trace}_P), \subseteq \rangle, \langle \wp(\text{Store}^*), \subseteq \rangle, \gamma_o)$. This approach is clearly more general because the above store changes abstraction α_{sc} is more precise than α_o, *i.e.*, for any set of traces T, $\gamma_{sc}(\alpha_{sc}(T)) \subseteq \gamma_o(\alpha_o(T))$, or, equivalently, $\alpha_{sc}(T_1) = \alpha_{sc}(T_2) \Rightarrow \alpha_o(T_1) = \alpha_o(T_2)$.

Example 7.1 (**Dead store elimination**). The above approach based on a generic observational abstraction allows us to prove the correctness of program optimizations that are unsound in Guo and Palsberg [17]'s framework, such as dead store elimination. For example, in a program fragment such as

```
**while** $(x \leq 0)$ **do**
    $z := 0;$
    $x := x + 1;$
    $z := 1;$
```

one can extract the hot path

$$hp = \langle x \leq 0, z := 0, x := x + 1, z := 1 \rangle$$

and perform dead store elimination by optimizing hp to $hp' = \langle x \leq 0, x := x+1, z := 1 \rangle$. As observed by Guo and Palsberg [17, Section 4.3], this is clearly unsound in bisimulation-based correctness because this hot path optimization does not output bisimilar code. By contrast, this optimization can be made sound in our framework by choosing an observational abstraction that records store changes at the beginning and at the exit of loops containing extracted hot paths. □

7.1 Correctness Proof

It turns out that observational correctness of the hot path extraction transform can be proved w.r.t. the more precise observational abstraction α_{sc}.

Theorem 7.2 (**Correctness of trace extraction**). *For any* $P \in \text{Program}$, $hp \in \alpha_{hot}^N(\mathbf{T}[\![P]\!])$, *we have that* $\alpha_{sc}(\mathbf{T}[\![extr_{hp}(P)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$.

In the rest of this section we outline a proof sketch of this result. Let us fix a hot path $hp = \langle a_0, C_0, ..., a_n, C_n \rangle \in \alpha_{hot}^N(\mathbf{T}[\![P]\!])$ and let $P_{hp} \triangleq extr_{hp}(P)$. The proof relies on a mapping of traces of the program P into corresponding traces of P_{hp} that unfolds the hot path hp (or any its initial fragment) according to the hot path extraction strategy given by Definition 6.1.

We define two functions $\text{tr}_{hp}^{in}, \text{tr}_{hp}^{out} : \text{Trace}_P \to \text{Trace}_{P_{hp}}$ in Fig. 3. The first function, $\text{tr}_{hp}^{out}(s\sigma)$, on the trace $s\sigma$ begins to unfold in P_{hp} the hot path hp when: (i) $s = \langle \rho, C_0 \rangle$ where C_0 is the first command of hp; and (ii) the condition guard E_{a_0} is satisfied in the store ρ. If this unfolding for the trace $s\sigma$ is actually started by applying $\text{tr}_{hp}^{out}(s\sigma)$ then it is carried on by applying $\text{tr}_{hp}^{in}(\sigma)$, *i.e.*, with a *in*-modality. The second function application, $\text{tr}_{hp}^{in}(s\sigma)$, carries on the unfolding of hp in P_{hp} when: (i) $s = \langle \rho, C_i \rangle$ where $i \in [1, n-1]$, namely the command C_i in hp is different from C_0 and C_n; and (ii) the condition guard E_{a_i} holds for the store ρ. If this is not the case then $\text{tr}_{hp}^{in}(\langle \rho, C_i \rangle \sigma)$, after a suitable unfolding step for $\langle \rho, C_i \rangle$, jumps back to the *out*-modality by progressing with $\text{tr}_{hp}^{out}(\sigma)$. It turns out that these two functions are well defined and tr_{hp}^{out} does not alter store change sequences.

Lemma 7.3.

(1) tr_{hp}^{out} *and* tr_{hp}^{in} *are well-defined,* i.e.*, for any* $\sigma \in \text{Trace}_P$, $\text{tr}_{hp}^{out}(\sigma), \text{tr}_{hp}^{in}(\sigma) \in \text{Trace}_{P_{hp}}$.
(2) *For any* $\sigma \in \text{Trace}_P$, $sc(\text{tr}_{hp}^{out}(\sigma)) = sc(\sigma)$.

Proof sketch of Theorem 7.2. Let us define $\text{tr}_{hp} : \wp(\text{Trace}_P) \to \wp(\text{Trace}_{P_{hp}})$ as $\text{tr}_{hp}(T) \triangleq \{\text{tr}_{hp}^{out}(\sigma) \mid \sigma \in T\}$. Technically, the proof consists in showing the following two points.

(A) $\text{tr}_{hp}(\mathbf{T}[\![P]\!]) \subseteq \mathbf{T}[\![P_{hp}]\!]$: this shows that for any execution trace σ of P, $\text{tr}_{hp}^{out}(\sigma)$ is an execution trace of P_{hp}; this is not hard to prove.

(B) $\alpha_{sc}(\mathbf{T}[\![P_{hp}]\!]) \subseteq \alpha_{sc}(\mathbf{T}[\![P]\!])$: this is proved by the following statement: $\sigma \in \mathbf{T}[\![P_{hp}]\!] \smallsetminus \text{tr}_{hp}(\mathbf{T}[\![P]\!]) \Rightarrow F[P_{hp}]\{\sigma\} \in \text{tr}_{hp}(\mathbf{T}[\![P]\!])$. The proof relies on the fact that one such trace σ is necessarily of the following shape: $\sigma = \sigma' \langle \rho, C \rangle$ where

$$\mathrm{tr}^{out}_{hp}(\epsilon) \triangleq \epsilon$$

$$\mathrm{tr}^{out}_{hp}(s\sigma) \triangleq \begin{cases} \langle \rho, L_0 : \text{guard } E_{a_0} \to \ell_0 \rangle \langle \rho, \ell_0 : act(C_0) \to \mathbb{l}_1 \rangle\, \mathrm{tr}^{in}_{hp}(\sigma) & \text{if } s = \langle \rho, C_0 \rangle,\ \alpha_{store}(\{\rho\}) \leq a_0 \\ \langle \rho, L_0 : \neg\text{guard } E_{a_0} \to \overline{L_0} \rangle \langle \rho, \overline{L_0} : act(C_0) \to L_1 \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, C_0 \rangle,\ \alpha_{store}(\{\rho\}) \not\leq a_0 \\ \langle \rho, L_0 : \text{guard } E_{a_0} \to \ell_0 \rangle \langle \rho, \ell_0 : \neg act(C_0) \to L_1^c \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, cmpl(C_0) \rangle,\ \alpha_{store}(\{\rho\}) \leq a_0 \\ \langle \rho, L_0 : \neg\text{guard } E_{a_0} \to \overline{L_0} \rangle \langle \rho, \overline{L_0} : \neg act(C_0) \to L_1^c \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, cmpl(C_0) \rangle,\ \alpha_{store}(\{\rho\}) \not\leq a_0 \\ s \cdot \mathrm{tr}^{out}_{hp}(\sigma) & \text{otherwise} \end{cases}$$

$$\mathrm{tr}^{in}_{hp}(\epsilon) \triangleq \epsilon$$

$$\mathrm{tr}^{in}_{hp}(s\sigma) \triangleq \begin{cases} \langle \rho, \mathbb{l}_i : \text{guard } E_{a_i} \to \ell_i \rangle \langle \rho, \ell_i : act(C_i) \to \mathbb{l}_{i+1} \rangle\, \mathrm{tr}^{in}_{hp}(\sigma) & \text{if } s = \langle \rho, C_i \rangle,\ i \in [1, n-1],\ \alpha_{store}(\{\rho\}) \leq a_i \\ \langle \rho, \mathbb{l}_n : \text{guard } E_{a_n} \to \ell_n \rangle \langle \rho, \ell_n : act(C_n) \to L_0 \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, C_n \rangle,\ \alpha_{store}(\{\rho\}) \leq a_n \\ \langle \rho, \mathbb{l}_i : \neg\text{guard } E_{a_i} \to L_i \rangle \langle \rho, C_i \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, C_i \rangle,\ i \in [1, n],\ \alpha_{store}(\{\rho\}) \not\leq a_i \\ \langle \rho, \mathbb{l}_i : \text{guard } E_{a_i} \to \ell_i \rangle \langle \rho, \ell_i : \neg act(C_i) \to L^c_{next(i)} \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, cmpl(C_i) \rangle,\ i \in [1, n],\ \alpha_{store}(\{\rho\}) \leq a_i \\ \langle \rho, \mathbb{l}_i : \neg\text{guard } E_{a_i} \to L_i \rangle \langle \rho, cmpl(C_i) \rangle\, \mathrm{tr}^{out}_{hp}(\sigma) & \text{if } s = \langle \rho, cmpl(C_i) \rangle,\ i \in [1, n],\ \alpha_{store}(\{\rho\}) \not\leq a_i \\ s \cdot \mathrm{tr}^{out}_{hp}(\sigma) & \text{otherwise} \end{cases}$$

Figure 3. Definition of tr^{out}_{hp} and tr^{in}_{hp}.

$act(C) \in \{\text{guard } E_{a_i}, \neg\text{guard } E_{a_i}\}$; then, it is not hard to prove that $F[P_{hp}]\{\sigma'\langle\rho, C\rangle\} \in \mathrm{tr}_{hp}(\mathbf{T}[\![P]\!])$. In words, one such trace σ of P_{hp} can be extended through an execution step in P_{hp} to a trace in $\mathrm{tr}_{hp}(\mathbf{T}[\![P]\!])$.

We therefore obtain:

$$\begin{aligned} \alpha_{sc}(\mathbf{T}[\![P]\!]) = \quad & [\text{By Lemma 7.3 (2), } \alpha_{sc} \circ \mathrm{tr}_{hp} = \alpha_{sc}] \\ \alpha_{sc}(\mathrm{tr}_{hp}(\mathbf{T}[\![P]\!])) \subseteq \quad & [\text{By point (A)}] \\ \alpha_{sc}(\mathbf{T}[\![P_{hp}]\!]) \subseteq \quad & [\text{By point (B)}] \\ \alpha_{sc}(\mathbf{T}[\![P]\!]) \quad & \end{aligned}$$

and this closes the proof. ☐

8. Type Specialization

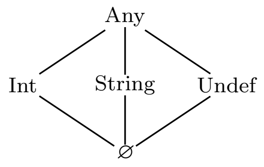
One key optimization for dynamic languages like JavaScript and PHP is type specialization, that is, using type-specific primitives in place of generic untyped operations whose runtime execution can be very costly. As a paradigmatic example, a generic addition operation could be defined on more than one type, so that the execution environment must check the type of its operands and execute a different operation depending on these types: this is the case of the addition operation in JavaScript (see its semantics in the ECMA-262 standard [21, Section 11.6]) and of the semantics of $+$ in our language as given in Section 2.2. Of course, type specialization avoids the overhead of dynamic type checking and dispatch of generic untyped operations. When a type is associated to each variable before the execution of a command in some hot path, this type environment can be used to replace generic operations with type-specific primitives.

8.1 Type Abstraction

Let us recall that the set of type names is

$$\text{Types} = \{\text{Int}, \text{String}, \text{Undef}, \text{Any}, \varnothing\}.$$

Type names can be therefore viewed as the following finite lattice $\langle \text{Types}, \subseteq \rangle$:



The abstraction map $\alpha_{type} : \wp(\text{Value}) \to \text{Types}$ takes a set of values and returns the smallest type containing it. Since Types viewed as a subset of $\wp(\text{Value})$ is closed under intersections (where Any is interpreted as the top element Value and $\varnothing$ is the bottom element), α_{type} can be indeed defined as a simple closure operator (*i.e.*, a monotonic, increasing and idempotent function) on $\langle \wp(\text{Value}), \subseteq \rangle$:

$$\alpha_{type}(V) \triangleq \cap\{T \in \text{Types} \mid V \subseteq T\}.$$

Given a value $v \in \text{Value}$, $\alpha_{type}(\{v\})$ thus coincides with $type(v)$. Here, the concretization function $\gamma_{type} : \text{Types} \to \wp(\text{Value})$ is simply the identity map (with $\text{Any} = \text{Value}$).

Following the general approach described in Section 4.1, we consider a simple nonrelational store abstraction for types

$$\text{Store}^t \triangleq \langle \text{Var} \to \text{Types}, \dot\subseteq \rangle$$

where $\dot\subseteq$ is the usual pointwise lifting of the ordering $\subseteq$ for Types, so that $\lambda x.\varnothing$ and $\lambda x.\text{Any}$ are, respectively, the bottom and top abstract stores in Store^t. The abstraction and concretization maps $\alpha_{store} : \wp(\text{Store}) \to \text{Store}^t$ and $\gamma_{store} : \text{Store}^t \to \wp(\text{Store})$ are defined as a straight instantiation of the definitions in Section 4.1.

The abstract type semantics $\mathbf{E}^t : \text{Exp} \to \text{Store}^t \to \text{Types}$ of expressions is defined as best correct approximation of the corresponding concrete semantics $\mathbf{E}$ on the type abstractions Store^t and Types, *i.e.*, $\mathbf{E}^t[\![E]\!]\rho^t \triangleq \alpha_{type}(\mathbf{E}[\![E]\!]\gamma_{store}(\rho^t))$. This definition leads to the following equalities:

$$\mathbf{E}^t[\![v]\!]\rho^t = type(v) \qquad \mathbf{E}^t[\![x]\!]\rho^t = \rho^t(x)$$

$$\mathbf{E}^t[\![E_1 + E_2]\!]\rho^t = \begin{cases} \varnothing & \text{if } \exists i.\, \mathbf{E}^t[\![E_i]\!]\rho^t = \varnothing \\ \mathbf{E}^t[\![E_1]\!]\rho^t & \text{else if } \mathbf{E}^t[\![E_1]\!]\rho^t = \mathbf{E}^t[\![E_2]\!]\rho^t \in \{\text{Int}, \text{String}\} \\ \text{Undef} & \text{else if } \forall i.\, \mathbf{E}^t[\![E_i]\!]\rho^t < \text{Any} \\ \text{Any} & \text{otherwise} \end{cases}$$

For instance, we have that:

$$\begin{aligned} &\mathbf{E}^t[\![x + y]\!][x/\text{String}, y/\varnothing] = \varnothing, \\ &\mathbf{E}^t[\![x + y]\!][x/\text{String}, y/\text{String}] = \text{String}, \\ &\mathbf{E}^t[\![x + y]\!][x/\text{Int}, y/\text{String}] = \text{Undef}, \\ &\mathbf{E}^t[\![x + y]\!][x/\text{Int}, y/\text{Any}] = \text{Any}. \end{aligned}$$

According to Section 6, for any abstract type store $[x_i/T_i \mid x_i \in \text{Var}]$ we consider a corresponding Boolean action guard

$$\text{guard } x_0 : T_0 \cdots x_n : T_n \in \text{BExp}$$

whose corresponding program action has the following semantics, which is automatically induced (as defined in Section 6) by the Galois connection $(\alpha_{store}, \wp(\text{Store}), \text{Store}^t, \gamma_{store})$: for any $\rho \in \text{Store}$,

$$\mathbf{A}[\![\text{guard } x_0 : T_0 \cdots x_n : T_n]\!]\rho \triangleq \begin{cases} \rho & \text{if } \forall i.\, \mathit{type}(\rho(x_i)) \subseteq T_i \\ \bot & \exists i.\, \mathit{type}(\rho(x_i)) \not\subseteq T_i \end{cases}$$

8.2 Type Specialization of Hot Paths

Let us consider some hot path $hp = \langle \rho_0^t, C_0, \ldots, \rho_n^t, C_n \rangle \in \alpha_{hot}^N(\mathbf{T}[\![P]\!])$ on the type abstraction $\langle \text{Store}^t, \dot{\subseteq} \rangle$, where each ρ_i^t is therefore a type map. The trace extraction transform $\mathit{extr}_{hp}(P)$ of P for hp gives rise to the set $\mathit{stitch}_P(hp)$ of commands that stitches the hot path hp into P. Hence, for any $i \in [0, n]$, $\mathit{stitch}_P(hp)$ contains a typed guard that we simply denote as guard ρ_i^t. Typed guards allow us to define type specialization of commands in the stitched hot path: this is defined as a program transform that instantiates most type-specific addition operations in place of generic untyped additions by exploiting the type information dynamically recorded by typed guards in $\mathit{stitch}_P(hp)$. Note that if $C \in \mathit{stitch}_P(hp)$ and $\mathit{act}(C) \equiv x := E_1 + E_2$ then $C \equiv \ell_i : x := E_1 + E_2 \to L'$, for some $i \in [0, n]$, where $L' \in \{\mathbb{l}_{i+1}, L_0\}$. Let $\mathbb{C}^t$ denote the set of commands that permits type specific additions $+_{\text{Int}}$ and $+_{\text{String}}$ and, in turn, Program^t denote the possible type specialized programs over $\mathbb{C}^t$. The function $\text{ts}_{hp} : \mathit{stitch}_P(hp) \to \mathbb{C}^t$ is defined as follows:

$$\text{ts}_{hp}(C) \triangleq C \qquad \text{if } \mathit{act}(C) \neq x := E_1 + E_2$$

$$\text{ts}_{hp}(\ell_i : x := E_1 + E_2 \to L') \triangleq$$

$$\begin{cases} \ell_i : x := E_1 +_{\text{Int}} E_2 \to L' & \text{if } \mathbf{E}^t[\![E_1 + E_2]\!]\rho_i^t = \text{Int} \\ \ell_i : x := E_1 +_{\text{String}} E_2 \to L' & \text{if } \mathbf{E}^t[\![E_1 + E_2]\!]\rho_i^t = \text{String} \\ \ell_i : x := E_1 + E_2 \to L' & \text{otherwise} \end{cases}$$

Hence, hot path type specialization TS is defined by

$$\text{TS}(\mathit{stitch}_P(hp)) \triangleq \{\text{ts}_{hp}(C) \mid C \in \mathit{stitch}_P(hp)\} \in \text{Program}^t.$$

The correctness of this program transform is quite straightforward. Let Trace^t be the set of traces for type specialized programs in Program^t and let $tt : \text{Trace}^t \to \text{Trace}$ be defined as follows:

$$tt(\epsilon) \triangleq \epsilon$$

$$tt(s\sigma) \triangleq$$

$$\begin{cases} \langle \rho, L : x := E_1 + E_2 \to L' \rangle tt(\sigma) & \\ & \text{if } s = \langle \rho, L : x := E_1 +_{\text{Type}} E_2 \to L' \rangle \\ s \cdot tt(\sigma) & \text{otherwise} \end{cases}$$

Theorem 8.1 (Correctness of type specialization). *For any typed $hp \in \alpha_{hot}^N(\mathbf{T}[\![P]\!])$, we have that $tt(\mathbf{T}[\![\text{TS}(\mathit{stitch}_P(hp))]\!]) = \mathbf{T}[\![\mathit{stitch}_P(hp)]\!]$.*

Typed trace extraction $\mathit{extr}_{hp}^t(P)$ consists in extracting and simultaneously type specializing a typed hot path hp in a program P, *i.e.*, it can be defined as follows:

$$\mathit{extr}_{hp}^t(P) \triangleq \mathit{extr}_{hp}(P) \smallsetminus \mathit{stitch}_P(hp) \cup \text{TS}(\mathit{stitch}_P(hp)).$$

Correctness of typed trace extraction extr_{hp}^t is a straight consequence of Theorems 7.2 and 8.1.

Corollary 8.2 (Correctness of typed trace extraction). *For any typed $hp \in \alpha_{hot}^N(\mathbf{T}[\![P]\!])$, we have that $\alpha_{sc}(\mathbf{T}[\![\mathit{extr}_{hp}^t(P)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$.*

Example 8.3. Let us consider the following sieve of Eratosthenes in a Javascript-like language – this is taken from the running example in [15] – where *primes* is initialized to an array of 100 *true* values. With a slight abuse, we assume that our language is extended with Boolean values and arrays. The semantics of arrays load and stores is as usual: first the index expression is checked to be in bounds, then the value is read or stored into the array. If the index is out of bounds, we assume the program is aborted.

```
for (var $i = 2;\ i < 100;\ i = i + 1$) do
    if (!$primes[i]$) then continue;
    for (var $k = i + i;\ k < 100;\ k = k + i$) do $primes[k] = false$;
```

This program is encoded in our language as follows:

$$\begin{aligned} P = \{ & C_0 \equiv L_0 : i := 2 \to L_1, \\ & C_1 \equiv L_1 : i < 100 \to L_2,\ C_1^c \equiv L_1 : \neg(i < 100) \to L_8, \\ & C_2 \equiv L_2 : \mathit{primes}[i] = \text{ tt } \to L_3, \\ & C_2^c \equiv L_2 : \neg(\mathit{primes}[i] = \text{ tt }) \to L_7, \\ & C_3 \equiv L_3 : k := i + i \to L_4, \\ & C_4 \equiv L_4 : k < 100 \to L_5,\ C_4^c \equiv L_4 : \neg(k < 100) \to L_7, \\ & C_5 \equiv L_5 : \mathit{primes}[k] := \text{ ff } \to L_6, \\ & C_6 \equiv L_6 : k := k + i \to L_4, \\ & C_7 \equiv L_7 : i := i + 1 \to L_1,\ C_8 \equiv L_8 : \text{skip} \to \text{Ł} \}. \end{aligned}$$

Let us consider the type environment ρ^t defined as

$$\rho^t \triangleq \{\mathit{primes}[n]/\,\text{Bool}, i/\,\text{Int}, k/\,\text{Int}\} \in \text{Store}^t$$

where $\mathit{primes}[n]/\,\text{Bool}$ is a shorthand for $\mathit{primes}[0]/\,\text{Bool}, \ldots, \mathit{primes}[99]/\,\text{Bool}$. Then the first traced 2-hot path on the type abstraction Store^t is:

$$hp_1 \triangleq \langle \rho^t, C_4, \rho^t, C_5, \rho^t, C_6 \rangle.$$

As a consequence, the typed transform extraction of hp_1 yields:

$$\begin{aligned} P_1 \triangleq \mathit{extr}_{hp_1}^t(P) = P & \smallsetminus \{C_4, C_4^c\} \\ & \cup \{\overline{L_4} : k < 100 \to L_5,\ \overline{L_4} : \neg(k < 100) \to L_7\} \\ & \cup \text{TS}(\mathit{stitch}_P(hp)) \end{aligned}$$

$$\begin{aligned} \text{where } \text{TS}(\mathit{stitch}_P(hp)) = \{ & \\ & H_0 \equiv L_4 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_0, \\ & H_0^c \equiv L_4 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \overline{L_4}, \\ & H_1 \equiv \ell_0 : k < 100 \to \mathbb{l}_1,\ H_1^c \equiv \ell_0 : \neg(k < 100) \to L_7, \\ & H_2 \equiv \mathbb{l}_1 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_1, \\ & H_2^c \equiv \mathbb{l}_1 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to L_5, \\ & H_3 \equiv \ell_1 : \mathit{primes}[k] := \text{ ff } \to \mathbb{l}_2, \\ & H_4 \equiv \mathbb{l}_2 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_2, \\ & H_4^c \equiv \mathbb{l}_2 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to L_6, \\ & H_5 \equiv \ell_2 : k := k +_{\text{Int}} i \to L_4 \}. \quad \square \end{aligned}$$

9. A General Correctness Criterion

Abstract interpretation allows us to view type specialization in Section 8 just as a particular *correct* hot path optimization that can be easily generalized. Guarded hot paths are a key feature of our tracing compilation model, where guards are dynamically recorded by the hot path monitor and range over abstract values in some store abstraction. An abstract guard for a command C in some hot path hp thus encodes a store property which is modeled in some abstract domain $\text{Store}^\sharp$ and is guaranteed to hold at the entry of C. This store information encapsulated by abstract guards can then be used to transform and optimize hp, *i.e.*, all the commands in the stitched hot path $\mathit{stitch}_P(hp)$.

This provides a modular approach to proving the correctness of some hot path optimization O. In fact, since correctness has to be proved w.r.t. some observational abstraction α_o of trace semantics and Theorem 7.2 ensures that this correctness holds for the store changes abstraction α_{sc} of the unoptimized trace extraction transform, we just need to prove the correctness of the optimization O on the whole stitched hot path $\mathit{stitch}_P(hp)$, which thus includes the abstract guards of the hot path hp. Hence, fixing a program P, a hot path optimization O is modeled as a program transform

$$O : \{\mathit{stitch}_P(hp) \mid hp \in \alpha^N_{hot}(\mathbf{T}[\![P]\!])\} \to \mathrm{Program}$$

where Program may permit new expressions and/or actions, like the case of type-specific addition operations in type specialization. O is required to be correct according to the following definition.

Definition 9.1 (Correctness of hot path optimization). O is *correct* if for any $P \in$ Program and for any $hp \in \alpha^N_{hot}(\mathbf{T}[\![P]\!])$, $\alpha_o(\mathbf{T}[\![O(\mathit{stitch}_P(hp))]\!]) = \alpha_o(\mathbf{T}[\![\mathit{stitch}_P(hp)]\!])$. ☐

As an example, it would be quite simple to formalize the variable folding optimization of hot paths considered by Guo and Palsberg [17] and to prove it correct in our framework w.r.t. the store changes abstraction α_{sc}.

10. Nested Hot Paths

Once a first hot path hp_1 has been extracted by transforming P to $P_1 \triangleq \mathit{extr}_{hp_1}(P)$, it may well happen that a new hot path hp_2 in P_1 contains hp_1 as a nested sub-path. Following TraceMonkey's trace recording strategy [15], we attempt to nest an inner hot path inside the current trace: during trace recording, an inner hot path is called as a subroutine, this executes a loop to a successful completion and then returns to the trace recorder that may therefore register the inner hot path as part of a new hot path.

To this aim, let us reshape the definitions in Section 5. Let P be the original program and P' be some hot path transform of P so that $P' \smallsetminus P$ contains all the commands (guards included) in the hot path. We define a function $\mathit{hotcut} : \mathrm{Trace}_{P'} \to (\mathrm{State}_{P'})^*$ that cuts from a trace in P' all the states whose commands appear in (some previous) hot path hp except the entry and exit states of hp:

$$\mathit{hotcut}(\sigma) \triangleq \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \mathit{hotcut}(\langle\rho_1, C_1\rangle\langle\rho_3, C_3\rangle\sigma') & \\ \qquad \text{if } \sigma = \langle\rho_1, C_1\rangle\langle\rho_2, C_2\rangle\langle\rho_3, C_3\rangle\sigma' \;\&\; C_1, C_2, C_3 \notin P & \\ \sigma_0\, \mathit{hotcut}(\sigma_1 \cdots \sigma_{|\sigma|-1}) & \text{otherwise} \end{cases}$$

In turn, we define $\mathit{outerhot}^N : \mathrm{Trace}_{P'} \to \wp((\mathrm{State}^\sharp_{P'})^*)$ as follows:

$$\begin{aligned} \mathit{outerhot}^N(\sigma) \triangleq \{&\langle a_i, C_i\rangle \cdots \langle a_j, C_j\rangle \in (\mathrm{State}^\sharp_{P'})^* \mid \\ &\exists\langle\rho_i, C_i\rangle \cdots \langle\rho_j, C_j\rangle \in \mathit{loop}(\mathit{hotcut}(\sigma)) \text{ s.t. } i \le j, \\ &\alpha_{store}(\langle\rho_i, C_i\rangle \cdots \langle\rho_j, C_j\rangle) = \langle a_i, C_i\rangle \cdots \langle a_j, C_j\rangle, \\ &\mathit{count}(\alpha_{store}(\mathit{hotcut}(\sigma)), \langle a_i, C_i\rangle \cdots \langle a_j, C_j\rangle) \ge N\}. \end{aligned}$$

Clearly, when $P' = P$ we have that $\mathit{hotcut} = \lambda\sigma.\sigma$ so that $\mathit{outerhot}^N = \mathit{hot}^N$. Finally, we define the collecting version $\alpha^N_{outerhot} \triangleq \lambda T. \cup_{\sigma\in T} \mathit{outerhot}^N(\sigma)$.

Example 10.1. Let us consider again Example 6.3, where $\mathrm{Store}^\sharp$ is the trivial one-point store abstraction $\{\top\}$. In Example 6.3, we first extracted $hp_1 = \langle\top, C_1, \top, C_2, \top, C_3^c\rangle$ by transforming P to $P_1 \triangleq \mathit{extr}_{hp}(P)$.

We then consider the following trace in $\mathbf{T}[\![P_1]\!]$:

$$\begin{aligned} \sigma = {} &\langle[x/?], C_0\rangle\langle[x/0], H_0\rangle\langle[x/0], H_1\rangle\langle[x/0], H_2\rangle\langle[x/0], H_3\rangle \\ &\langle[x/1], H_4\rangle\langle[x/1], H_5\rangle \cdots \langle[x/2], H_3\rangle\langle[x/3], H_4\rangle\langle[x/3], H_5^c\rangle \\ &\langle[x/3], C_4\rangle\langle[x/6], H_0\rangle \cdots \langle[x/9], H_5^c\rangle\langle[x/9], C_4\rangle\langle[x/12], H_0\rangle \cdots \end{aligned}$$

Thus, here we have that

$$\begin{aligned} \mathit{hotcut}(\sigma) = {} &\langle[x/?], C_0\rangle\langle[x/0], H_0\rangle\langle[x/3], H_5^c\rangle\langle[x/3], C_4\rangle \\ &\langle[x/6], H_0\rangle\langle[x/9], H_5^c\rangle\langle[x/9], C_4\rangle \cdots \end{aligned}$$

so that

$$hp_2 = \langle\top, H_0, \top, H_5^c, \top, C_4\rangle \in \alpha^2_{outerhot}(\mathbf{T}[\![P_1]\!])$$

Hence, hp_2 contains a nested hot path, which is called at the beginning of hp_2 and whose entry and exit commands are, respectively, H_0 and H_5^c. ☐

Let $hp = \langle a_0, C_0, \ldots, a_n, C_n\rangle \in \alpha^N_{outerhot}(\mathbf{T}[\![P']\!])$ be a N-hot path in P', where, for all $i \in [0, n]$, $C_i \equiv L_i : A_i \to L_{next(i)}$. Let us note that:

- If for all $i \in [0, n]$, $C_i \in P$ then hp actually is a hot path in P, *i.e.*, $hp \in \alpha^N_{hot}(\mathbf{T}[\![P]\!])$.
- Otherwise, there exists some $C_k \notin P$. If $C_i \in P$ and $C_{i+1} \notin P$ then C_{i+1} is the entry command of some inner hot path; on the other hand, if $C_i \notin P$ and $C_{i+1} \in P$ then C_i is the exit command of some inner hot path.

The transform of P' for extracting hp is a generalization of Definition 6.1.

Definition 10.2 (Nested trace extraction transform). The *nested trace extraction transform* of P' for the hot path hp is:

$$\begin{aligned} &\mathit{extr}_{hp}(P') \triangleq P \\ (1)\;& \smallsetminus (\{C_0 \mid C_0 \in P\} \cup \{\mathit{cmpl}(C_0) \mid \mathit{cmpl}(C_0) \in P\}) \\ (2)\;& \cup \{\overline{L_0} : \mathit{act}(C_0) \to L_1 \mid C_0 \in P\} \\ & \cup \{\overline{L_0} : \neg\mathit{act}(C_0) \to L_1^c \mid \mathit{cmpl}(C_0) \in P\} \\ (3)\;& \cup \{L_0 : \text{guard } E_{a_0} \to \ell_0,\, L_0 : \neg\text{guard } E_a \to \overline{L_0} \mid C_0 \in P\} \\ (4)\;& \cup \{\ell_i : \mathit{act}(C_i) \to \mathbb{l}_{i+1} \mid i \in [0, n-1], C_i, C_{i+1} \in P\} \\ & \cup \{\ell_n : \mathit{act}(C_n) \to L_0 \mid C_n \in P\} \\ (5)\;& \cup \{\ell_i : \neg\mathit{act}(C_i) \to L^c_{next(i)} \mid i \in [0, n], C_i, \mathit{cmpl}(C_i) \in P\} \\ (6)\;& \cup \{\mathbb{l}_i : \text{guard } E_{a_i} \to \ell_i, \mathbb{l}_i : \neg\text{guard } E_{a_i} \to L_i \mid \\ & \qquad\qquad i \in [1, n], C_i \in P\} \\ (7)\;& \cup \{\ell_i : \mathit{act}(C_i) \to L_{i+1} \mid i \in [0, n-1], C_i \in P, C_{i+1} \notin P\} \\ (8)\;& \smallsetminus \{C_i \mid i \in [0, n-1], C_i \notin P, C_{i+1} \in P\} \\ (9)\;& \cup \{L_i : \mathit{act}(C_i) \to \mathbb{l}_{i+1} \mid i \in [0, n-1], C_i \notin P, C_{i+1} \in P\} \end{aligned}$$

while $\mathit{stitch}_{P'}(hp) \triangleq (3) \cup (4) \cup (5) \cup (6) \cup (7) \cup (9)$. ☐

Let us observe that:

- Clauses (1)–(6) are the same clauses of Definition 6.1, with the additional constraints that C_i and $\mathit{cmpl}(C_i)$ are all commands in P, conditions which are trivially satisfied in Definition 6.1.
- Clause (7) where $C_i \in P$ and $C_{i+1} \notin P$, namely $next(C_i)$ is the call program point of a nested hot path nhp and C_{i+1} is the entry command of nhp, performs a relabeling that allows to correctly nest nhp in hp.
- Clauses (8)–(9) where $C_i \notin P$ and $C_{i+1} \in P$, *i.e.*, C_i is the exit command of a nested hot path nhp that returns to the program point $\mathit{lbl}(C_{i+1})$, performs the relabeling of $\mathit{suc}(C_i)$ in C_i in order to return from nhp to hp;

– $\overline{L_0}$, ℓ_i and $\mathbb{l}_i$ are fresh labels, *i.e.*, they have not been used in P'.

Example 10.3. Let us go on with Example 10.1. The second traced hot path in $\alpha^2_{outerhot}(\mathbf{T}[\![P_1]\!])$ is:

$$hp_2 = \langle \top, H_0 \equiv L_1 : \text{guard } E_\top \to \ell_0,$$
$$\top, H^c_5 \equiv \ell_2 : (x\%3 = 0) \to L_4, \top, C_4 \equiv L_4 : x := x{+}3 \to L_1\rangle.$$

According to Definition 10.2, trace extraction of hp_2 in P_1 yields the following transform:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{extr}_{hp_2}(P_1) \triangleq} \\
\text{[by clause (8)]} & P_1 \smallsetminus \{H^c_5\} \\
\text{[by clause (9)]} & \cup\,\{\ell_2 : (x\%3 = 0) \to \mathbb{h}_2\} \\
\text{[by clause (6)]} & \cup\,\{\mathbb{h}_2 : \text{guard } E_\top \to \hbar_2, \mathbb{h}_2 : \neg\text{guard } E_\top \to L_4\} \\
\text{[by clause (4)]} & \cup\,\{\hbar_2 : x := x + 3 \to L_1\}
\end{array}
$$

where we used additional fresh labels in $\mathbb{h}_2$ and $\hbar_2$. □

Example 10.4. Let us consider again Example 8.3. After the trace extraction of hp_1 that transforms P to P_1, a second traced 2-hot path is the following:

$$hp_2 \triangleq \langle \rho^t, C_1, \rho^t, C_2, \rho^t, C_3, \rho^t, H_0, \rho^t, H^c_1, \rho^t, C_7\rangle$$

where $\rho^t = \{\mathit{primes}[n]/\,\text{Bool}, i/\,\text{Int}, k/\,\text{Int}\} \in \text{Store}^t$. hp_2 contains a nested hot path which is called at $suc(C_3) = L_4$ and whose entry and exit commands are, respectively, H_0 and H^c_1. Here, typed trace extraction according to Definition 10.2 provides the following transform of P_1:

$$
\begin{array}{l}
P_2 \triangleq \mathit{extr}^t_{hp_2}(P_1) \triangleq P_1 \smallsetminus \{C_1, C^c_1\} \cup \big\{ \\
\quad \overline{L_1} : i < 100 \to L_2,\ \overline{L_1} : \neg(i < 100) \to L_8, \\
\quad H_6 \equiv L_1 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_3, \\
\quad H^c_6 \equiv L_1 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \overline{L_1}, \\
\quad H_7 \equiv \ell_3 : i < 100 \to \mathbb{l}_4,\ H^c_7 \equiv \ell_3 : \neg(i < 100) \to L_8, \\
\quad H_8 \equiv \mathbb{l}_4 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_4, \\
\quad H^c_8 \equiv \mathbb{l}_4 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to L_2, \\
\quad H_9 \equiv \ell_4 : \mathit{primes}[i] = \text{tt} \to \mathbb{l}_5, \\
\quad H^c_9 \equiv \ell_4 : \neg(\mathit{primes}[i] = \text{tt}) \to L_7, \\
\quad H_{10} \equiv \mathbb{l}_5 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_5, \\
\quad H^c_{10} \equiv \mathbb{l}_5 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to L_3, \\
\quad H_{11} \equiv \ell_5 : k := i +_{\text{Int}} i \to L_4\big\} \\
\quad \smallsetminus \{H^c_1\} \cup \big\{(H^c_1)' \equiv \ell_0 : \neg(k < 100) \to \mathbb{l}_6, \\
\quad H_{12} \equiv \mathbb{l}_6 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_6, \\
\quad H^c_{12} \equiv \mathbb{l}_6 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to L_7, \\
\quad H_{13} \equiv \ell_6 : i := i +_{\text{Int}} 1 \to L_1\big\}.
\end{array}
$$

Finally, a third traced 2-hot path in P_2 is the following:

$$hp_3 \triangleq \langle \rho^t, H_6, \rho^t, H^c_9, \rho^t, C_7\rangle$$

which contains a nested hot path which is called at the beginning of hp_3 and whose entry and exit commands are, respectively, H_6 and H^c_9. Here, typed trace extraction of hp_3 yields:

$$
\begin{array}{l}
P_3 \triangleq \mathit{extr}_{hp_3}(P_2) \triangleq P_2 \smallsetminus \{H^c_9\} \cup \big\{ \\
\quad (H^c_9)' \equiv \ell_4 : \neg(\mathit{primes}[i] = \text{tt}) \to \mathbb{l}_7, \\
\quad \mathbb{l}_7 : \text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to \ell_7, \\
\quad \mathbb{l}_7 : \neg\text{guard } (\mathit{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \to L_7, \\
\quad \ell_7 : i := i +_{\text{Int}} 1 \to L_1\big\}.
\end{array}
$$

We have thus obtained the same three trace extraction steps as described by Gal et al. [15, Section 2]. In particular, in P_1 we specialized the typed addition operation $k := k +_{\text{Int}} i$, in P_2 we specialized $k := i +_{\text{Int}} i$ and $i := i +_{\text{Int}} 1$, while in P_3 we specialized once again $i := i +_{\text{Int}} 1$ in a different hot path. Thus, in P_3 all the addition operations have been type specialized. □

11. Comparison with Guo and Palsberg's Framework

A formal model for tracing JIT compilation has been put forward in POPL 2011 by Guo and Palsberg [17]. Its main distinctive feature is the use of bisimulation [23] to describe operational equivalence between source and optimized programs. In this section we show how this model can be expressed within our framework.

11.1 Language and Semantics

Guo and Palsberg [17] employ a simple imperative language with while loops and a so-called bail construct.

$$
\begin{array}{l}
E ::= n \mid x + 1 \\
B ::= x = 0 \mid x \neq 0 \\
\text{Cmd} \ni c ::= \textbf{skip} \mid x := E \mid \textbf{if } B \textbf{ then } S \mid \\
\qquad\qquad\quad \textbf{while } B \textbf{ do } S \mid \textbf{bail } B \textbf{ to } S \\
\text{Stm} \ni S ::= \epsilon \mid c; S
\end{array}
$$

The baseline small-step operational semantics $\to_B \subseteq \text{State} \times \text{State}$, where $\text{State} \triangleq \text{Store} \times \text{Stm}$, is standard. Let us just recall the semantics of bail commands:

$$
\begin{array}{ll}
\langle \rho, (\textbf{bail } B \textbf{ to } S); K\rangle \to_B \langle \rho, K\rangle & \text{if } [\![B]\!]\rho = \mathit{false} \\
\langle \rho, (\textbf{bail } B \textbf{ to } S); K\rangle \to_B \langle \rho, S\rangle & \text{if } [\![B]\!]\rho = \mathit{true}
\end{array}
$$

If $\text{Trace}_{GP} \triangleq \{\sigma \in \text{State}^* \mid \forall i \in [0, |\sigma|).\, \sigma_i \to_B \sigma_{i+1}\}$ denotes the set of program traces for Guo and Palsberg's language then given a program $S \in \text{Stm}$, the trace transition operator $GP[S] : \wp(\text{Trace}_{GP}) \to \wp(\text{Trace}_{GP})$ is defined as usual:

$$GP[S](X) \triangleq \{\langle \rho, S\rangle \mid \rho \in \text{Store}\} \cup \{\sigma s s' \mid \sigma s \in X,\ s \to_B s'\}$$

so that the trace semantics of S is $\mathbf{T}_{GP}[\![S]\!] \triangleq \text{lfp}(GP[S]) \in \wp(\text{Trace}_{GP})$.

11.2 Language Compilation

Programs in Stm can be easily compiled into Program by resorting to an injective labeling function $\ell : \text{Stm} \to \mathbb{L}$ that assigns different labels to different statements.

Definition 11.1 (Language compilation). The compilation function $\mathcal{C} : \text{Stm} \to \wp(\mathbb{C})$ is recursively defined by the following clauses:

$$
\begin{array}{l}
\mathcal{C}(\epsilon) \triangleq \{\ell(\epsilon) : \textbf{skip} \to \text{\L}\} \\
\mathcal{C}(S \equiv \textbf{skip}; K) \triangleq \{\ell(S) : \textbf{skip} \to \ell(K)\} \cup \mathcal{C}(K) \\
\mathcal{C}(S \equiv x := E; K) \triangleq \{\ell(S) : x := E \to \ell(K)\} \cup \mathcal{C}(K) \\
\mathcal{C}(S \equiv (\textbf{if } B \textbf{ then } S'); K) \triangleq \\
\quad \{\ell(S) : B \to \ell(S'; K),\ \ell(S) : \neg B \to \ell(K)\} \cup \mathcal{C}(S'; K) \cup \mathcal{C}(K) \\
\mathcal{C}((\textbf{while } B \textbf{ do } S'); K) \triangleq \mathcal{C}((\textbf{if } B \textbf{ then } (S'; \textbf{while } B \textbf{ do } S')); K) \\
\mathcal{C}(S \equiv (\textbf{bail } B \textbf{ to } S'); K) \triangleq \\
\quad \{\ell(S) : B \to \ell(S'),\ \ell(S) : \neg B \to \ell(K)\} \cup \mathcal{C}(S') \cup \mathcal{C}(K)
\end{array}
$$

□

Example 11.2. Consider the following program $S \in \text{Stm}$ in Guo and Palsberg's syntax:

$x := 0;$
while B_1 **do** $x := 1;$
$x := 2;$
bail B_2 **to** $x := 3;$
$x := 4;$
ϵ

S is then compiled in our language by $\mathcal{C}$ in Definition 11.1 as follows:

$$\begin{aligned}\mathcal{C}(S) = \big\{&\ell_0 : x := 0 \to \ell_{\mathbf{while}},\\ &\ell_{\mathbf{while}} : B_1 \to \ell_1,\ \ell_{\mathbf{while}} : \neg B_1 \to \ell_2,\\ &\ell_1 : x := 1 \to \ell_{\mathbf{while}},\ \ell_2 : x := 2 \to \ell_{\mathbf{bail}},\\ &\ell_{\mathbf{bail}} : B_2 \to \ell_3,\ \ell_{\mathbf{bail}} : \neg B_2 \to \ell_4,\\ &\ell_3 : x := 3 \to \ell_\epsilon,\ \ell_4 : x := 4 \to \ell_\epsilon,\ \ell_\epsilon : \mathbf{skip} \to \text{Ł}\big\}.\end{aligned}$$

$\square$

Correctness for the above compilation function $\mathcal{C}$ means that for any $S \in \text{Stm}$: (i) $\mathcal{C}(S) \in \text{Program}$ and (ii) program traces of S and $\mathcal{C}(S)$ have the same store sequences. If $st : \text{Trace}_{GP} \cup \text{Trace} \to \text{Store}^*$ returns the store sequence of a trace, *i.e.*, $st(\epsilon) \triangleq \epsilon$ and $st(\langle\rho, S\rangle\sigma) \triangleq \rho \cdot st(\sigma)$, and, for a set X of traces $\alpha_{st}(X) \triangleq \{st(\sigma) \mid \sigma \in X\}$, then correctness goes as follows:

Theorem 11.3 (Correctness of language compilation). *For any* $S \in \text{Stm}$, $\mathcal{C}(S) \in \text{Program}$ *and* $\alpha_{st}(\mathbf{T}_{GP}[\![S]\!]) = \alpha_{st}(\mathbf{T}[\![\mathcal{C}(S)]\!])$.

11.3 Bisimulation

Correctness of trace extraction in [17] relies on the following notion of bisimulation, which is parameterized by program stores.

Definition 11.4 ([17]). A relation $R \subseteq \text{Store} \times \text{Stm} \times \text{Stm}$ is a *bisimulation* when $R(\rho, S_1, S_2)$ implies: (1) if $\langle\rho, S_1\rangle \to_B \langle\rho', S_1'\rangle$ then $\langle\rho, S_2\rangle \to_B^* \langle\rho', S_2'\rangle$, for some $\langle\rho', S_2'\rangle$ such that $R(\rho', S_1', S_2')$; (2) $R(\rho, S_2, S_1)$. $\square$

S_1 is bisimilar to S_2 for a given store ρ, denoted by $S_1 \approx_\rho S_2$, if $R(\rho, S_1, S_2)$ for some bisimulation R. It is simple to characterize this program equivalence through an abstraction map of traces that observes store changes (this is analogous to the definition of *sc* in Section 7).

$$stc : \text{Store}^* \to \text{Store}^* \qquad stc(\epsilon) \triangleq \epsilon \quad stc(\rho) \triangleq \rho$$

$$stc(\rho_1\rho_2\sigma) \triangleq \begin{cases} stc(\rho_2\sigma) & \text{if } \rho_1 = \rho_2 \\ \rho_1 \cdot stc(\rho_2\sigma) & \text{if } \rho_1 \neq \rho_2 \end{cases}$$

Given $\rho \in \text{Store}$, $\alpha^\rho_{stc} : \wp(\text{Trace}_{GP}) \to \wp(\text{Store}^*)$ is defined as follows: $\alpha^\rho_{stc}(X) \triangleq \{stc(\sigma) \mid \sigma \in \alpha_{st}(X),\ \sigma_0 = \rho\}$.

Theorem 11.5. *For any* $S_1, S_2 \in \text{Stm}$, $\rho \in \text{Store}$, *we have that* $S_1 \approx_\rho S_2$ *iff* $\alpha^\rho_{stc}(\mathbf{T}_{GP}[\![S_1]\!]) = \alpha^\rho_{stc}(\mathbf{T}_{GP}[\![S_2]\!])$.

11.4 Hot Paths and Trace Extraction

In Guo and Palsberg's model [17]: (i) hot paths always begin with an entry while-loop conditional, which is however not included in the hot path; (ii) the store of a hot path is recorded at the end of the first loop iteration and is a concrete store; and (iii) hot paths actually are 1-hot paths according to our definition. Guo and Palsberg's hot loops can be modeled in our framework by relying on a loop selection map $loop_{GP} : \text{Trace} \to \wp(\mathbb{C}^* \times \text{Store})$ defined as follows:

$$\begin{aligned} loop_{GP}(\langle\rho_0, C_0\rangle \cdots \langle\rho_n, C_n\rangle) \triangleq \big\{&\langle C_i C_{i+1} \cdots C_j, \rho_{j+1}\rangle \mid \\ &0 \le i \le j < n,\ C_i \lessdot C_j,\ lbl(C_{j+1}) = lbl(C_i),\\ &\forall k \in (i, j].\, C_k \notin \{C_i, cmpl(C_i)\}\big\}. \end{aligned}$$

Notice that, for simplicity, the above definition includes the entry loop conditional in the hot path. The map $\alpha^{GP}_{hot} : \wp(\text{Trace}) \to \wp(\mathbb{C}^* \times \text{Store})$ then lifts $loop_{GP}$ to sets of traces: $\alpha^{GP}_{hot}(T) \triangleq \cup_{\sigma \in T}\, loop_{GP}(\sigma)$.

Let us thus consider a hot path $hp = \langle C_0 C_1 \cdots C_n, \rho\rangle \in \alpha^{GP}_{hot}(\mathbf{T}[\![P]\!])$, for some $P \in \text{Program}$ (where P may coincide with a compiled $\mathcal{C}(S)$ for some $S \in \text{Stm}$) and let us follow the same notation used in Section 6. Guo and Palsberg's [17] trace extraction scheme is defined as follows, where the hot path *hp* cannot be re-entered once execution leaves *hp*.

Definition 11.6 (GP trace extraction transform). The *GP trace extraction transform* of P for the hot path *hp* is:

$$\begin{aligned} &extr^{GP}_{hp}(P) \triangleq P \smallsetminus \{C_0\}\\ &\quad \cup \{L_0 : act(C_0) \to \ell_1\}\\ &\quad \cup \{\ell_i : act(C_i) \to \ell_{i+1} \mid i \in [1, n)\} \cup \{\ell_n : act(C_n) \to L_0\}\\ &\quad \cup \{\ell_i : \neg act(C_i) \to L^c_{next(i)} \mid i \in [1, n],\ cmpl(C_i) \in P\}. \end{aligned}$$

$\square$

Clearly, $extr^{GP}_{hp}(P)$ remains a well-formed program. The correctness of this GP trace extraction transform, which is stated and proved in [17, Lemma 3.6], goes as follows.

Theorem 11.7 (Correctness of GP trace extraction). *For any* $P \in \text{Program}$, $hp = \langle C_0 \cdots C_n, \rho\rangle \in \alpha^{GP}_{hot}(\mathbf{T}[\![P]\!])$, *we have that* $\alpha^\rho_{stc}(\mathbf{T}[\![extr^{GP}_{hp}(P)]\!]) = \alpha^\rho_{stc}(\mathbf{T}[\![P]\!])$.

Example 11.8. Let us consider the program P in Example 2.1 and the GP-hot path

$$hp = \langle C_1 C_2 C_3^c, \rho = [x/1]\rangle \in \alpha^{GP}_{hot}(\mathbf{T}[\![P]\!]).$$

A corresponding 2-hot path hp_1 with the same sequence of commands has been selected in Example 5.1 and extracted in Example 6.3. Here, the GP trace extraction of *hp* provides the following program transform:

$$\begin{aligned} &extr^{GP}_{hp}(P) \triangleq P \smallsetminus \{C_1\}\\ &\qquad \cup \{L_1 : x \le 20 \to \ell_1,\ \ell_1 : x := x + 1 \to \ell_2\}\\ &\qquad \cup \{\ell_2 : \neg(x\%3 = 0) \to L_1,\ \ell_2 : (x\%3 = 0) \to L_4\}. \end{aligned}$$

$\square$

12. Further Work

We have put forward a formal model of tracing compilation and correctness of hot path optimization based on program trace semantics and abstract interpretation. We see a number of interesting avenues for further work on this topic. We aim at making use of this framework to study and relate the foundational differences between traditional static vs dynamic tracing compilation. We then expect to formalize and prove the correctness of most beneficial optimizations employed by tracing compilers of practical dynamic languages like JavaScript, PHP and Python. For example, we plan to cast in our model the allocation removal optimization for Python described in [5] in order to formally prove its correctness. Finally, we plan to adapt our framework in order to provide a model of whole-method just-in-time compilation, as used, e.g., by IonMonkey [14], the current JIT compilation scheme in the Firefox JavaScript engine.

Acknowledgments

We are grateful to the anonymous referees for their helpful comments. The work of Francesco Ranzato was partially supported by Microsoft Research Software Engineering Innovation Foundation 2013 Award (SEIF 2013) and by the University of Padova under the project BECOM.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 1–12, New York, NY, USA, 2000. ACM.

[2] R. Barbuti, N. De Francesco, A. Santone, and G. Vaglini. Abstract interpretation of trace semantics for concurrent calculi. *Information Processing Letters*, 70(2):69–78, 1999.

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2010)*, pages 708–725, New York, NY, USA, 2010. ACM.

[4] I. Böhm, T.J.K. Edler von Koch, S.C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pages 74–85, New York, NY, USA, 2011. ACM.

[5] C.F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*, pages 43–52. ACM, 2011.

[6] C.F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS 2009)*, pages 18–25, New York, NY, USA, 2009. ACM.

[7] C. Colby and P. Lee. Trace-based program analysis. In *Proceedings of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1996)*, pages 195–207, New York, NY, USA, 1996. ACM.

[8] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation (extended abstract). *Electronic Notes in Theoretical Computer Science*, 6(0):77–102, 1997. Proceedings of the 13th Annual Conference on Mathematical Foundations of Progamming Semantics (MFPS XIII).

[9] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, New York, NY, USA, 1977. ACM.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1979)*, pages 269–282, New York, NY, USA, 1979. ACM.

[12] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2002)*, pages 178–190, New York, NY, USA, 2002. ACM.

[13] Mozilla Foundation. TraceMonkey. `wiki.mozilla.org`, October 2010.

[14] Mozilla Foundation. IonMonkey. `wiki.mozilla.org`, May 2013.

[15] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M.R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E.W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 465–478, New York, NY, USA, 2009. ACM.

[16] A. Gal, C.W. Probst, and M. Franz. HotPathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 2006)*, pages 144–153. ACM, 2006.

[17] S. Guo and J. Palsberg. The essence of compiling with traces. In *Proceedings of the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2011)*, pages 563–574, New York, NY, USA, 2011. ACM.

[18] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the 5th International Static Analysis Symposium (SAS 1998)*, volume 1503 of *LNCS*, pages 200–214. Springer, 1998.

[19] Facebook Inc. The HipHop Virtual Machine, Facebook Engineering, December 2011.

[20] Google Inc. A new crankshaft for V8, The Chromium Blog, December 2010.

[21] Ecma International. ECMAScript Language Specification. Standard ECMA-262, Edition 5.1, June 2011.

[22] F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems and Structures*, 35(2):100–142, 2009.

[23] R. Milner. *Communication and Concurrency*. Prentice Hall, 1995.

[24] M. Pall. The LuaJIT Project. `luajit.org`, 2005.

[25] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2004)*, New York, NY, USA, 2004. ACM.

[26] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[27] D.A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp Symb. Comput.*, 10(3):237–271, 1998.

[28] F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.