An Abstract Interpretation-based Model of Tracing Just-In-Time Compilation

STEFANO DISSEGNA*, FRANCESCO LOGOZZO‡ and FRANCESCO RANZATO*

*Dipartimento di Matematica, University of Padova, Italy
‡Microsoft Research, Redmond, USA

Abstract

Tracing just-in-time compilation is a popular compilation technique 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 program execution in order 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 with 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 practical success of tracing compilation, very little is known about its theoretical foundations. We provide a formal model of tracing compilation of programs using abstract interpretation. The monitoring phase (viz., hot path detection) corresponds to an abstraction of the trace semantics of the program 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 phase (viz., residual program generation) 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 the model of tracing compilation introduced by Guo and Palsberg [2011] which is 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.

1 Introduction

Efficient traditional static compilation of popular dynamic languages like JavaScript, Python and PHP is very hard if not impossible. In particular, 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) [Bala et al., 2000; Bebenita et al., 2010; Böhm et al., 2011; Bolz et al., 2009, 2011; Gal et al., 2006, 2009; Paal, 2005] has emerged as a valuable implementation and optimization technique for dynamic languages (and not only, e.g. Java [Häußl and Mössenböck, 2011; Häußl et al. 2014; Inoue et al. 2011]). For instance, the Facebook HipHop virtual machine for PHP and the V8 JavaScript engine of Google Chrome use some form of tracing compilation [Facebook Inc., 2013; Google Inc., 2010]. 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) [Mozilla Foundation, 2010, 2013].

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 (i.e., no loops or join points are allowed) 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 the hot path. 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 most 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 these questions.

Our formal model is based on program trace semantics [Cousot, 2002] and abstract interpretation [Cousot and Cousot, 1977, 2002]. 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 model.

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 the model of TJITC by Guo and Palsberg [2011] based on program bisimulations: we employ a less restrictive correctness criterion that enables the correctness proof of practically 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 [2002] 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 run-time 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 Cousot and Cousot [2002]. 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 [Gal et al., 2009], as they make available 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 by [Guo and Palsberg, 2011] at POPL. It is based on operational bisimulation [Milner, 1995] to describe equivalence between source and optimized programs. In Section 10 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 [Guo and Palsberg, 2011] 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 [Guo and Palsberg, 2011], while it is implemented in actual tracing compilers [Gal et al., 2009]. Furthermore, their formalization fails to model some important features of actual TJITC implementation: (i) traces are mere 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 [1997, 2002] and has been widely used as a technique for defining a range of static program analyses [Barbuti et al., 1999; Colby and Lee, 1996; Handjieva and Tzolovski, 1998; Logozzo, 2009; Rival and Mauborgne, 2007; Schmidt, 1998; Spoto and Jensen, 2003]. Abstract interpretation has been used to describe static compilation and optimizations. In particular, Rival [2004] describes various optimizations as the trace abstractions they preserve. In the Cousot and Cousot terminology [Cousot and Cousot, 2002], Rival approach corresponds to offline transformations whereas tracing compilation is an online transformation.

This is an expanded and revised version of the article [Dissegna et al., 2014] including all the proofs.

2 Language and Concrete Semantics

2.1 Notation

Given a finite set $X$ of objects, we will use the following notation concerning sequences: $\epsilon$ is the empty sequence; $X^+$ is the set of nonempty finite sequences of objects of $X$; $X^* \triangleq X^+ \cup \{\epsilon\}$; if $\sigma \in X^*$ then $|\sigma|$ denotes the length of $\sigma$; indices of objects in a sequence $\sigma \in X^*$ start from 0 and thus range in the interval $[0, |\sigma|) \triangleq [0, |\sigma| - 1]$; if $\sigma \in X^*$ and $i \in [0, |\sigma|]$ then $\sigma_i$ (or $\sigma(i)$) denotes the $i$-th object in $\sigma$.

2.2 Syntax

We consider a basic low level language with untyped global variables, a kind of elementary dynamic language, which is defined through the notation used in [Cousot and Cousot, 2002]. Program commands range in $\mathbb{C}$ and consist of a labeled action which specifies a next label ($L$ is the undefined label, where the execution becomes stuck: it is used for defining final commands).

- **Labels:** $L \in \mathbb{L}$, $L \not\in \mathbb{L}$
- **Values:** $v \in \text{Value}$
- **Variables:** $x \in \text{Var}$
- **Expressions:** $\mathbb{E} \ni E ::= v \mid x \mid E_1 + E_2$
- **Boolean Expressions:** $\mathbb{B} \ni B ::= \text{tt} \mid \text{ff} \mid E_1 \leq E_2 \mid \neg B \mid B_1 \land B_2$
- **Actions:** $\mathbb{A} \ni A ::= x ::= E \mid B \mid \text{skip}$
- **Commands:** $\mathbb{C} \ni C ::= L : A \rightarrow L'$ (with $L' \in \mathbb{L} \cup \{L\}$)
For any command $L : A \rightarrow L'$, we use the following notation:

$$\text{lbl}(L : A \rightarrow L') \triangleq L, \quad \text{act}(L : A \rightarrow L') \triangleq A, \quad \text{suc}(L : A \rightarrow L') \triangleq L'$$

Commands $L : B \rightarrow L'$ whose action is a Boolean expression are called conditionals. A program $P \in \wp(C)$ is a (possibly infinite, at least in theory) set of commands. In order to be well-formed, if a program $P$ includes a conditional $C \equiv L : B \rightarrow L'$ then $P$ must also include a unique complement conditional $L : \neg B \rightarrow L''$, which is denoted by $\text{cmpl}(C)$ or $C^\circ$, where $\neg B$ is taken to be equal to $B$, so that $\text{cmpl}(\text{cmpl}(C)) = C$. The set of well-formed programs is denoted by $\text{Program}$. In our examples, programs $P$ will be deterministic, i.e., for any $C_1, C_2 \in P$ such that $\text{lbl}(C_1) = \text{lbl}(C_2)$: (1) if $\text{act}(C_1) \neq \text{act}(C_2)$ then $C_1 = \text{cmpl}(C_2)$; (2) if $\text{act}(C_1) = \text{act}(C_2)$ then $C_1 = C_2$. We say that two programs $P_1$ and $P_2$ are equal up to label renaming, denoted by $P_1 \equiv P_2$, when there exists a suitable renaming for the labels of $P_1$ that makes $P_1$ equal to $P_2$.

### 2.3 Transition Semantics

The language semantics relies on the following types, where $\text{Char}$ is a nonempty finite set of characters and $\text{undef}$ represents a generic error:

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

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

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

while the mapping $\text{type} : \text{Value} \rightarrow \text{Types}$ provides the type of any value. Here, the type name $\text{Any}$ plays the role of top type, which is the supertype of (i.e., contains) all types, while $\varnothing$ is the bottom type, which is the subtype of all types.

Let $\text{Store} \triangleq \text{Var} \rightarrow \text{Value}$ denote the set of possible stores on variables in Var. If $P \in \text{Program}$ then $\text{vars}(P)$ denotes the set of variables in Var that occur in $P$, so that $\text{Store}_P \triangleq \text{vars}(P) \rightarrow \text{Value}$ is the set of possible stores for $P$. The semantics of expressions and program actions is standard and goes as defined in Fig. 1.
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 \( \text{undefined} \) values, i.e., \( \neg\text{undefined} = \text{undefined} \) and \( \text{undefined} \land b = \text{undefined} \land \text{undef} \).

\( p[x/v] \) denotes a store update for the variable \( x \) with the value \( v \). With a slight abuse of notation we also consider the so-called collecting versions of the semantic functions in Fig.1, which are defined as follows:

\[
\begin{align*}
E &: \text{Exp} \to \varphi(\text{Store}) \to \varphi(\text{Value}) \\
E[L]S &\triangleq \{E[L]_\rho \mid \rho \in S\} \\
B &: B\text{Exp} \to \varphi(\text{Store}) \to \varphi(\text{Store}) \\
B[B]S &\triangleq \{\rho \in S \mid B[B]_\rho = \text{true}\} \\
A &: A \to \varphi(\text{Store}) \to \varphi(\text{Store}) \\
A[A]S &\triangleq \{A[A]_\rho \mid \rho \in S \land A[A]_\rho \in \text{Store}\}
\end{align*}
\]

Generic program states are pairs of stores and commands: \( \text{State} \triangleq \text{Store} \times C \). We extend the previous functions \( \text{lbl}, \text{act} \) and \( \text{suc} \) to be defined on states, meaning that they are defined on the command component of a state. Also, \( \text{store}(s) \) and \( \text{cmd}(s) \) return, respectively, the store and the command of a state \( s \). The transition semantics \( \text{S} : \text{State} \to \varphi(\text{State}) \) is a relation between generic states defined as follows:

\[
\text{S}(\rho, C) \triangleq \{\langle \rho', C' \rangle \in \text{State} \mid \rho' \in \text{A}[\text{act}(C)]\{\rho\}, \text{suc}(C) = \text{lbl}(C')\}.
\]

If \( P \) is a program then \( \text{State}_P \triangleq \text{State}_P \times P \) is the set of possible states of \( P \). Given \( P \in \text{Program} \), the program transition relation \( \text{S}[P] : \text{State}_P \to \varphi(\text{State}_P) \) between states of \( P \) is defined as:

\[
\text{S}[P] \{\rho, C\} \triangleq \{\langle \rho', C' \rangle \in \text{State}_P \mid \rho' \in \text{A}[\text{act}(C)]\{\rho\}, C' \in P, \text{suc}(C) = \text{lbl}(C')\}.
\]

A state \( s \in \text{State}_P \) is stuck when \( \text{S}[P]s = \emptyset \). Let us remark that, according to the above definition, if \( C \triangleq \lambda L : A \to L', C_1 \equiv L' : B \to L'' \) and \( C' \equiv L'' : \text{act} \to L''' \) are all commands in \( P \) and \( \rho' \in \text{A}[\text{act}]\{\rho\} \) then we have that \( \text{S}[P] \{\rho, C\} = \{\langle \rho', C'\rangle \} \). Moreover, if the conditional command of a state \( s = \langle \rho, L : A \to L \rangle \) is such that \( \text{B}[B]_{\rho} = \text{false} \) then \( s \) is stuck. Also, if the command of a state \( s = \langle \rho, L : A \to L \rangle \) has the the undefined label \( L \) as next label then \( s \) is stuck as well.

Programs typically have an entry point, which is modeled through a distinct initial label \( L_i \in \mathcal{L} \) from which execution starts. \( \text{State}_P \triangleq \{\langle \rho, C\rangle \mid \text{lbl}(C) = L_i\} \) denotes the set of possible initial states for \( P \).

### 2.3.1 Trace Semantics

A partial trace is any nonempty finite sequence of generic program states which are related by the transition relation \( \text{S} \). Hence, the set \( \text{Trace} \) of partial traces is defined as follows:

\[
\text{Trace} \triangleq \{\sigma \in \text{State}^+ \mid \forall i \in [1, |\sigma|). \sigma_i \in \text{S}\sigma_{i-1}\}.
\]

The partial trace semantics of \( P \in \text{Program} \) is in turn defined as follows:

\[
\text{T}[P] = \text{Trace}_P \triangleq \{\sigma \in (\text{State}_P)^+ \mid \forall i \in [1, |\sigma|). \sigma_i \in \text{S}[P]\sigma_{i-1}\}.
\]

A trace \( \sigma \in \text{Trace}_P \) is complete if for any state \( s \in \text{State}_P, \sigma s \not\in \text{Trace}_P \) and \( \sigma s \not\notin \text{Trace}_P \). Observe that \( \text{Trace}_P \) contains all the possible partial traces of \( P \), complete traces included.

Let us remark that a trace \( \sigma \in \text{Trace}_P \) does not necessarily begin with an initial state, namely it may happen that \( \sigma_0 \not\not\in \text{State}_P \). Traces of \( P \) starting from initial states are denoted by

\[
\text{T}^i[P] = \text{Trace}_P^i \triangleq \{\sigma \in \text{Trace}_P \mid \sigma_0 \in \text{State}_P^i\}.
\]

Also, a complete trace \( \sigma \in \text{Trace}_P^i \) such that \( \text{suc}(\sigma_{|\sigma|-1}) = L \) corresponds to a terminating run of the program \( P \).

#### Example 2.1

Let us consider the program \( Q \) below written in some while-language:

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

5
Its translation as a program \(P\) in our language is given below, where, with a little abuse, we assume an extended syntax that allows expressions like \(x \% 3 = 0\).

\[
P = \begin{cases} 
C_0 \equiv L_0 : x := 0 \to L_1, \\
C_1 \equiv L_1 : x \leq 20 \to L_2, \\
C_2 \equiv L_2 : x := x + 1 \to L_3, \\
C_3 \equiv L_3 : (x \% 3 = 0) \to L_4, \\
C_4 \equiv L_4 : x := x + 3 \to L_1, \\
C_5 \equiv L_5 : \text{skip} \to L_1 
\end{cases}
\]

Its trace semantics from initial states includes the following complete traces, where ? stands for any integer value and stores are denoted within square brackets.

\[
\langle \{x/\?\}, C_0 \rangle, \langle \{x/\?\}, C_1 \rangle, \langle \{x/\?\}, C_2 \rangle, \langle \{x/\?\}, C_3 \rangle, \langle \{x/\?\}, C_4 \rangle \\
\vdots
\]

\[
\langle \{x/\?\}, C_0 \rangle, \langle \{x/\?\}, C_1 \rangle, \langle \{x/\?\}, C_2 \rangle, \langle \{x/\?\}, C_3 \rangle, \langle \{x/\?\}, C_4 \rangle, \langle \{x/21\}, C_1 \rangle, \langle \{x/22\}, C_1 \rangle, \langle \{x/23\}, C_1 \rangle, \langle \{x/24\}, C_1 \rangle
\]

Observe that the last trace corresponds to a terminating run of \(P\).}

3 Abstractions

3.1 Abstract Interpretation Background

In standard abstract interpretation [Cousot and Cousot 1977, 1979], abstract domains, also called abstractions, are specified by Galois connections/insertions (GCs/GIs for short) or, equivalently, adjunctions. Concrete and abstract domains, \((C, \leq_C)\) and \((A, \leq_A)\), 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 \((\alpha, C, A, \gamma)\), that is, for all \(a \in A\) and \(c \in C\), \(\alpha(c) \leq_A a \iff c \leq_C \gamma(a)\). A GC is a GI when \(\alpha \circ \gamma = \lambda x.a\). It is well known that a join-preserving \(\alpha\) uniquely determines \(\gamma\) as follows: \(\gamma(a) = \bigvee \{c \in C \mid \alpha(c) \leq_A a\}\); conversely, a meet-preserving \(\gamma\) uniquely determines \(\alpha\) as follows: \(\alpha(c) = \bigwedge \{a \in A \mid c \leq_C \gamma(a)\}\).

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 GI. 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 between functions. Moreover, the abstract function \(f^\sharp\) is called the best correct approximation of \(f\) on \(A\) because any abstract function \(f^\sharp\) is correct iff \(f^\sharp \sqsubseteq f^\sharp\). Hence, for any abstraction \(A\), \(f^\sharp\) plays the role of the best possible approximation of \(f\) on the abstract domain \(A\).

3.2 Store Abstractions

As usual in abstract interpretation [Cousot and Cousot 1977], a store property is modeled by some abstraction \(\text{Store}^\sharp\) which is encoded through a Galois connection:

\[
\langle \alpha_{\text{store}}, \langle \wp(\text{Store}), \subseteq \rangle, \langle \text{Store}^\sharp, \leq \rangle, \gamma_{\text{store}} \rangle.
\]

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

Given a program \(P\), when \(\text{Store}^\sharp\) is viewed as an abstraction of \(\langle \wp(\text{Store}_P), \subseteq \rangle\) we emphasize it by adopting the notation \(\text{Store}_P^\sharp\). A store abstraction \(\text{Store}_P^\sharp\) also induces a state abstraction \(\text{State}_P^\sharp \equiv \text{Store}_P^\sharp \times P\) and, in turn, a trace abstraction \(\text{Trace}_P^\sharp \equiv \langle \text{State}_P^\sharp \rangle^*\).

3.2.1 Nonrelational Abstractions

Nonrelational store abstractions (i.e., relationships between program variables are not taken into account) 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

\[
\langle \alpha_{\text{value}}, \langle \wp(\text{Value}), \subseteq \rangle, \langle \text{Value}^\sharp, \leq_{\text{Value}} \rangle, \gamma_{\text{value}} \rangle.
\]
The abstract domain Value\(^{\dagger}\) induces a nonrelational store abstraction
\[
\rho^\dagger \in \text{Store}_{\text{value}}^{\dagger} \triangleq \{ \text{Var} \rightarrow \text{Value}^{\dagger}, \subseteq \}
\]
where \(\subseteq\) is the pointwise ordering induced by \(\leq_{\text{Value}}^{\dagger}\): \(\rho_1^\dagger \subseteq \rho_2^\dagger\) iff for all \(x \in \text{Var}\), \(\rho_1^\dagger(x) \leq_{\text{Value}}^{\dagger} \rho_2^\dagger(x)\). Here, the bottom and top abstract stores are, respectively, \(\lambda x.\bot_{\text{value}}^{\dagger}\) and \(\lambda x.\top_{\text{value}}^{\dagger}\). The abstraction map \(\alpha_{\text{value}}^{\dagger} : \varphi(\text{Store}) \rightarrow \text{Store}_{\text{value}}^{\dagger}\) is thus defined as follows:
\[
\alpha_{\text{value}}^{\dagger}(S) \triangleq \lambda x.\alpha_{\text{value}}\{\rho(x) \in \text{Value} \mid \rho \in S\}
\]
while the corresponding concretization map \(\gamma_{\text{value}}^{\dagger} : \text{Store}_{\text{value}}^{\dagger} \rightarrow \varphi(\text{Store})\) is defined by adjunction from \(\alpha_{\text{value}}^{\dagger}\) as recalled in Section 3.1 and it is easy to check that it turns out to be defined as follows:
\[
\gamma_{\text{value}}^{\dagger}(\rho^\dagger) = \{ \rho \in \text{Store} \mid \forall x \in \text{Var} . \rho(x) \in \gamma_{\text{value}}(\rho^\dagger(x)) \}.
\]

4 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., [Gal et al., 2009]). Here, hot path selection is not operationally defined, it is instead semantically modeled as an abstraction map over program traces, i.e., program executions.

Given a program \(P\) and therefore its trace semantics \(\text{Trace}_P\), we first define a mapping \(\text{loop} : \text{Trace}_P \rightarrow \varphi(\text{Trace}_P)\) that returns all the loop paths in some execution trace of \(P\). More precisely, a loop path is a proper substring (i.e., a segment) \(\tau\) of a program trace \(\sigma\) such that:

(1) the successor command in \(\sigma\) of the last state in \(\tau\) exists and coincides with the command – or its complement, when this is the last loop iteration – of the first state in \(\tau\);

(2) there is no other such command within \(\tau\) (otherwise the sequence \(\tau\) would contain multiple iterations);

(3) the last state of \(\tau\) 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 \(P\), denoted by \(\prec\). This leads to the following formal definition:
\[
\text{loop}(\langle \rho_0, C_0 \rangle \cdots \langle \rho_n, C_n \rangle) \triangleq \{ \langle \rho_i, C_i \rangle \langle \rho_{i+1}, C_{i+1} \rangle \cdots \langle \rho_j, C_j \rangle \mid 0 \leq i \leq j < n, C_i \prec C_j, \text{suc}(C_j) = \text{lbl}(C_i), \forall k \in (i, j), C_k \notin \{ C_i, \text{cmpl}(C_i) \}\}
\]

Let us remark that a loop path
\[
\langle \rho_i, C_i \rangle \cdots \langle \rho_j, C_j \rangle \in \text{loop}(\langle \rho_0, C_0 \rangle \cdots \langle \rho_n, C_n \rangle)
\]
may contain some sub-loop path, namely it may happen that \(\text{loop}(\langle \rho_i, C_i \rangle \cdots \langle \rho_j, C_j \rangle) \neq \emptyset\) so that some commands \(C_k,\) with \(k \in [i, j]\), may occur more than once in \(\langle \rho_i, C_i \rangle \cdots \langle \rho_j, C_j \rangle\); for example, this could be the case of a while loop whose body include a nested while loop.

We abuse notation by using \(\alpha_{\text{store}}\) to denote a map \(\alpha_{\text{store}} : \text{Trace}_P \rightarrow \text{Trace}_P^{\dagger}\) which “abstracts” program traces into \(\text{Trace}_P^{\dagger}\) by disregarding commands:
\[
\alpha_{\text{store}}(\langle \rho_0, C_0 \rangle \cdots \langle \rho_n, C_n \rangle) \triangleq \langle \alpha_{\text{store}}(\langle \rho_0 \rangle), C_0 \rangle \cdots \langle \alpha_{\text{store}}(\langle \rho_n \rangle), C_n \rangle.
\]

Given a static integer parameter \(N > 0\), we define a function
\[
\text{hot}^N : \text{Trace}_P \rightarrow \varphi(\text{Trace}_P^{\dagger})
\]
which returns the set of \(\text{Store}^{\dagger}\)-abstracted loop paths appearing at least \(N\) times in some program trace. In order to count the number of times a loop path appears within a trace we need an auxiliary function \(\text{count} :\)
Hence, if count yields the number of times an abstract path \( \tau \) occurs in an abstract trace \( \sigma \):

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

Hence, hot^N can be defined as follows:

\[
\text{hot}^N(\sigma \equiv \langle \rho_0, C_0 \rangle \cdots \langle \rho_n, C_n \rangle) \triangleq \{ \langle a_i, C_i \rangle \cdots \langle a_j, C_j \rangle \mid \exists \langle \rho_i, C_i \rangle \cdots \langle \rho_j, C_j \rangle \in \text{loop}(\sigma) \text{ s.t.} \alpha_{\text{store}}(\langle \rho_i, C_i \rangle \cdots \langle \rho_j, C_j \rangle) = \langle a_i, C_i \rangle \cdots \langle a_j, C_j \rangle, \\
\text{count}(\alpha_{\text{store}}(\sigma), \langle a_i, C_i \rangle \cdots \langle a_j, C_j \rangle) \geq N \}.
\]

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

\[
\alpha^N_{\text{hot}}(T) \triangleq \bigcup_{\sigma \in T} \text{hot}^N(\sigma).
\]

A N-hot path \( hp \) in a program \( P \) is therefore any \( hp \in \alpha^N_{\text{hot}}(\text{Trace}_P) \) and is compactly denoted as \( hp = (a_0, C_0, \ldots, a_n, C_n) \). Let us observe that if the hot path corresponds to 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 \( (a_0, C_0, \ldots, a_n, C_n) \): next \( \triangleq \lambda i. i = n \ ? \ 0 : i + 1 \). For a N-hot path \( (a_0, C_0, \ldots, a_n, C_n) \in \alpha^N_{\text{hot}}(\text{Trace}_P) \), for any \( i \in \{0, n\} \), if \( C_i \) is a conditional command \( L_i : B_i \rightarrow L_{\text{next}(i)} \) then throughout the paper its complement \( C_i^c = \text{cmpl}(C_i) \) will be also denoted by \( L_i : \neg B_i \rightarrow L_{\text{next}(i)}^c \).

Example 4.1. Let us consider the program \( P \) in Example 2.1 and a trivial one-point store abstraction \( Store^I = \{ \top \} \), where all the stores are abstracted to the same abstract store \( \top \), i.e., \( \alpha_{\text{store}} = \lambda S. \top \). Here, we have two 2-hot paths in \( P \), that is, it turns out that \( \alpha^2_{\text{hot}}(\text{Trace}_P) = \{ hp_1, hp_2 \} \) where:

\[
\begin{align*}
hp_1 &= \langle \top, C_1 \equiv L_1 : x \leq 20 \rightarrow L_2, \top, C_2 \equiv L_2 : x := x + 1 \rightarrow L_3, \top, C_3 \equiv L_3 : \neg (x \% 3 = 0) \rightarrow L_4; \\
hp_2 &= \langle \top, C_1 \equiv L_1 : x \leq 20 \rightarrow L_2, \top, C_2 \equiv L_2 : x := x + 1 \rightarrow L_3, \top, C_3 \equiv L_3 : (x \% 3 = 0) \rightarrow L_4, \\
& \quad \top, C_4 \equiv x := x + 3 \rightarrow L_4 \rangle.
\end{align*}
\]

Therefore, the hot paths \( hp_1 \) and \( hp_2 \) correspond, respectively, to the cases where the Boolean test \( x \% 3 = 0 \) fails and succeeds. Observe that the maximal sequence of different values assumed by the program variable \( x \) is as follows:

\[
? \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 24
\]

Hence, if \( \sigma \) is the complete terminating trace of \( P \) in Example 2.1 then we have that count(\( \alpha_{\text{store}}(\sigma), hp_1 \)) = 8 and count(\( \alpha_{\text{store}}(\sigma), hp_2 \)) = 4. \( \square \)

5 Trace Extraction

For any abstract store \( a \in Store^I \), a corresponding Boolean expression denoted by guard \( E_a \in \text{BExp} \) 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 \( \rho \in \text{Store} \),

\[
\text{B}[\text{guard } E_a]\rho \triangleq \begin{cases} \text{true} & \text{if } \rho \in \gamma_{\text{store}}(a) \\ \text{false} & \text{if } \rho \notin \gamma_{\text{store}}(a) \end{cases}
\]

In turn, we also have program actions guard \( E_a \in \mathcal{A} \) such that:

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

8
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.
Let $P$ be a program and $hp = \langle a_0, C_0, ..., a_n, C_n \rangle \in \alpha_{hot}^N(Trace_P)$ be a hot path on some store abstraction $Store^\delta$. We define a syntactic transform of $P$ where the hot path $hp$ is explicitly extracted from $P$. This is achieved by a suitable relabeling of each command $C_i$ in $hp$ which is in turn preceded by the conditional guard $E_{ai}$, induced by the corresponding store property $a_i$. To this aim, we consider three injective relabeling functions

$$
el : [0, n] \rightarrow \mathbb{L}_1 \quad \i : [1, n] \rightarrow \mathbb{L}_2 \quad \rightarrow : \mathbb{L} \rightarrow \mathbb{L}$$

(*)

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

Definition 5.1 (Trace extraction transform). The trace extraction transform of $P$ for the hot path $hp = \langle a_0, C_0, ..., a_n, C_n \rangle$ is:

$$extr_{hp}(P) \triangleq P \setminus \left( \{ C_0 \} \cup \{ \text{compl}(C_0) \mid \text{compl}(C_0) \in P \} \right) \cup \{ L_0 : \text{act}(C_0) \rightarrow L_1 \} \cup \{ L_0 : \neg \text{act}(C_0) \rightarrow L_1' \mid \text{compl}(C_0) \in P \} \cup \text{stitch}_P(hp)$$

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

$$\text{stitch}_P(hp) \triangleq \{ L_0 : \text{guard } E_{a_0} \rightarrow \ell_0, L_0 : \neg \text{guard } E_{a_0} \rightarrow \overline{L_0} \}$$

$$\cup \{ \ell_i : \text{act}(C_i) \rightarrow L_{i+1} \mid i \in [0, n-1] \} \cup \{ \ell_n : \text{act}(C_n) \rightarrow L_0 \}$$

$$\cup \{ \ell_i : \neg \text{act}(C_i) \rightarrow L_{i\text{next}} \mid i \in [0, n], \text{compl}(C_i) \in P \}$$

$$\cup \{ \ell_1 : \text{guard } E_{a_1} \rightarrow \ell_1, \ell_1 : \neg \text{guard } E_{a_1} \rightarrow L_1 \mid i \in [1, n] \}.$$  

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

Lemma 5.2. If $P$ is well-formed then, for any hot path $hp$, $extr_{hp}(P)$ is well-formed.

Proof. Recall that a program is well-formed when for any its conditional command it also includes a unique complement conditional. It turns out that $extr_{hp}(P)$ is well-formed because: (1) $P$ is well-formed; (2) for each conditional in $P_{new} = extr_{hp}(P) \setminus P = \text{stitch}_P(hp) \cup \{ L_0 : \text{act}(C_0) \rightarrow L_1 \} \cup \{ \overline{L_0} : \neg \text{act}(C_0) \rightarrow L_1' \mid \text{compl}(C_0) \in P \}$ we also have a unique complement conditional in $P_{new}$. Moreover, observe that if $P$ is deterministic then $extr_{hp}(P)$ still is deterministic.

Let us stress that the stitch of the hot path $hp$ into $P$ is always a linear sequence of different relabellings, namely, $\text{stitch}_P(hp)$ 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 in (*) 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 $\text{stitch}_P(hp)$.

Example 5.3. Let us consider the program $P$ in Example 4.1 and the hot path $hp_1 = \langle T, C_1, T, C_2, T, C_3 \rangle$ in Example 4.1 where stores are abstracted to the trivial one-point abstraction $Store^\delta = \{ \top \}$. Here, for any $\rho \in Store$, we have that $B[guard E_T] \rho = true$. The trace extraction transform of $P$ w.r.t. $hp$ is therefore as follows:

$$extr_{hp}(P) = P \setminus \{ C_1, C_1', \} \cup \{ \overline{L_2} : x \leq 20 \rightarrow L_2, \overline{L_1} : \neg(x \leq 20) \rightarrow L_\delta \} \cup \text{stitch}_P(hp)$$

where

$$\text{stitch}_P(hp) = \{ H_0 \equiv L_1 : \text{guard } E_T \rightarrow \ell_0, H_0' \equiv L_1 : \neg \text{guard } E_T \rightarrow \overline{L_1} \}$$

$$\cup \{ H_1 \equiv \ell_0 : x \leq 20 \rightarrow L_1, H_1' \equiv \ell_0 : \neg(x \leq 20) \rightarrow L_5 \}$$

$$\cup \{ H_2 \equiv \ell_1 : \text{guard } E_T \rightarrow \ell_1, H_2' \equiv \ell_1 : \neg \text{guard } E_T \rightarrow L_2 \}$$

$$\cup \{ H_3 \equiv \ell_1 : x := x + 1 \rightarrow L_2 \}$$

$$\cup \{ H_4 \equiv \ell_2 : \text{guard } E_T \rightarrow \ell_2, H_4' \equiv \ell_2 : \neg \text{guard } E_T \rightarrow L_3 \}$$

$$\cup \{ H_5 \equiv \ell_2 : \neg(x \%3 \equiv 0) \rightarrow L_1, H_5' \equiv \ell_2 : (x \%3 \equiv 0) \rightarrow L_4 \}.$$  

Hence, $extr_{hp}(P)$ can be rewritten at a higher level representation using while-loops and gotos as follows:
\[x \colonequals 0;\]
\[
L_1 : \textbf{while guard } E \textbf{ do } \\
\quad \textbf{if } \neg(x \leq 20) \textbf{ then goto } L_5; \\
\quad \textbf{if } \neg guard E \textbf{ then goto } L_2; \\
\quad x \colonequals x + 1; \\
\quad \textbf{if } \neg guard E \textbf{ then goto } L_3; \\
\quad \textbf{if } (x \% 3 = 0) \textbf{ then goto } L_4; \\
\quad \textbf{if } \neg(x \leq 20) \textbf{ then goto } L_5; \\
L_2 : x \colonequals x + 1; \\
L_3 : \textbf{if } (x \% 3 = 0) \textbf{ then goto } L_1; \\
L_4 : x \colonequals x + 3; \\
\textbf{goto } L_1; \\
L_5 : \textbf{skip}.
\]

6 Correctness

As advocated by Cousot and Cousot [2002, par. 3.8], correctness of dynamic program transformations and optimizations should be defined with respect to some observational abstraction of program trace semantics: a dynamic 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 [2011] to tracing compilation basically relies on a notion of correctness that requires the same store changes in execution traces of a program \( P \):

\[ sc : \text{Trace}_P \rightarrow \text{Store}_P^* \]

\[ sc(\sigma) \triangleq \begin{cases} \\
\varepsilon & \text{if } \sigma = \varepsilon \\
\rho & \text{if } \sigma = \langle \rho, C \rangle \\
sc(\langle \rho, C_1 \rangle \sigma') & \text{if } \sigma = \langle \rho, C_0 \rangle \langle \rho, C_1 \rangle \sigma' \\
\rho_0 sc(\langle p_1, C_1 \rangle \sigma') & \text{if } \sigma = \langle \rho_0, C_0 \rangle \langle p_1, C_1 \rangle \sigma', \rho_0 \neq \rho_1 
\end{cases} \]

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

Since the function \( \alpha_{sc} \) obviously preserves arbitrary set unions, as recalled in Section 3.1, it admits a right adjoint \( \gamma_{sc} : \varphi(\text{Store}_P^*) \rightarrow \varphi(\text{Trace}_P) \) defined as \( \gamma_{sc}(S) \triangleq \cup \{ T \in \varphi(\text{Trace}_P) \mid \alpha_{sc}(T) \subseteq S \} \), that gives rise to a GC \( (\alpha_{sc}, \varphi(\text{Trace}_P), \subseteq), \varphi(\text{Store}_P^*), \subseteq, \gamma_{sc}) \).

However, the store changes abstraction \( \alpha_{sc} \) may be too strong in practice. This condition can be thus relaxed and generalized to an observational abstraction of execution traces that demands to have the same stores (possibly restricted to some subset of program variables) only at some specific program points. For example, these program points may depend on the language. In a language with no output primitives and functions, as that considered by Guo and Palsberg [2011], 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 command “put \( X \)” that “outputs” the value of program variables ranging in some set \( X \) then we may want to have stores with the same values for variables in \( X \) 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 \( X \in A, \text{where } X \subseteq \text{Var} \) is a set of program variables. The semantics of put \( X \) obviously does not affect program stores, i.e., \( A[\text{put } X] \rho \triangleq \rho \). Correspondingly, an observational abstraction \( \alpha_o : \varphi(\text{Trace}_P) \rightarrow \varphi(\text{Store}_P^X) \) (where Store\( _P^X \) denotes stores on variables ranging in \( X \)) of trace semantics observes program stores at output program points only:

\[ \text{out} : \text{Trace}_P \rightarrow \text{Store}_P^X \]

\[ \text{out}(\sigma) \triangleq \begin{cases} \\
\varepsilon & \text{if } \sigma = \varepsilon \\
\text{out}(\sigma') & \text{if } \sigma = s\sigma' \land \text{act}(s) \neq \text{put } X \\
\rho_{\text{out}}\text{out}(\sigma') & \text{if } \sigma = \langle \rho, L : \text{put } X \rightarrow L' \rangle \sigma' 
\end{cases} \]

\[ \alpha_o(T) \triangleq \{ \text{out}(\sigma) \mid \sigma \in T \} \]

where \( \rho_{\text{out}} \) denotes the restriction of the store \( \rho \) to variables in \( X \). Similarly to \( \alpha_{sc} \), here again we have a GC \( (\alpha_o, (\varphi(\text{Trace}_P), \subseteq), (\varphi(\text{Store}_P^X), \subseteq), \gamma_o) \). This approach is clearly more general because the above store changes
Lemma 6.3. \( \alpha_{sc} \) is more precise (i.e., less approximate) than \( \alpha_c \), i.e., for any set of traces \( T \), \( \gamma_{sc}(\alpha_{sc}(T)) \subseteq \gamma_c(\alpha_c(T)) \), or, equivalently, \( \alpha_{sc}(T_1) = \alpha_{sc}(T_2) \Rightarrow \alpha_c(T_1) = \alpha_c(T_2) \).

Example 6.1 (Dead store elimination). The above approach based on a generic observational abstraction enables to prove the correctness of program optimizations that are unsound in Guo and Palsberg [2011]'s framework, such as dead store elimination. For example, in a program fragment such as

\[
\text{while } (x \leq 0) \text{ do } \quad z := 0; \\
\text{ } x := x + 1; \\
\text{ } z := 1; \\
\]

one can extract the hot path \( hp = \langle x \leq 0, z := 0, x := x + 1, z := 1 \rangle \) (here we disregard store abstractions) and perform dead store elimination of the command \( z := 0 \) by optimizing \( hp \) to \( hp' = \langle x \leq 0, x := x + 1, z := 1 \rangle \).

As observed by Guo and Palsberg [2011, 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.

6.1 Correctness Proof

It turns out that the observational correctness of the hot path extraction transform can be proved w.r.t. the observational abstraction \( \alpha_{sc} \) of store changes.

Theorem 6.2 (Correctness of trace extraction). For any \( P \in \text{Program} \) and \( hp \in \alpha_{hot}(\text{Trace}_P) \), we have that \( \alpha_{sc}(\text{T}[\text{extr}_hp(P)]) = \alpha_{sc}(\text{T}[P]) \).

In order to prove this correctness result, we need to define suitable “dynamic” transformations of execution traces. Let us fix a hot path \( hp = (a_0, C_0, ..., a_n, C_n) \in \alpha_{hot}(\text{Trace}_P) \) (w.r.t. some store abstraction) and let \( P_{hp} \triangleq \text{extr}_hp(P) \) denote the corresponding transform of \( P \). We first define a mapping of the execution traces of the program \( P \) into execution 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 5.1 that is, this function should replace any occurrence of the hot path \( hp \) in some execution trace of \( P \) with its corresponding guarded and suitably relabeled path obtained through Definition 5.1. More precisely, we define in Fig. 3 two functions

\[
\text{tr}_{hp}^{out} : \text{Trace}_P \rightarrow \text{Trace}_{P_{hp}} \quad \text{tr}_{hp}^{in} : \text{Trace}_P \rightarrow (\text{State}_P \cup \text{State}_{P_{hp}})^* 
\]

A function application \( \text{tr}_{hp}^{out}(s) \) in \( out \)-modality on a trace \( s \) of \( P \) triggers the unfolding of the hot path \( hp \) in \( P_{hp} \) when the state \( s \) is such that: (i) \( s = (\rho, C_0) \) where \( C_0 \) is the first command of \( hp \); and (ii) the condition guard \( E_{a_0} \) is satisfied in the store \( \rho \) of \( s \). If the unfolding for the trace \( (\rho, C_0) \) is actually started by applying \( \text{tr}_{hp}^{out}(\rho, C_0) \) then: first, \( (\rho, C_0) \) is unfolded into \( (\rho, l_0 : \text{guard } E_{a_0} \rightarrow l_0)(\rho, l_0 : \text{act } C_0 \rightarrow I_1) \); then, the unfolding is carried on by applying \( \text{tr}_{hp}^{in}(\rho) \), i.e., with an \( in \)-modality.

The function application \( \text{tr}_{hp}^{in}(s) \) carries on the unfolding of \( hp \) in \( P_{hp} \) when the state \( s \) is such that: (i) \( s = (\rho, C_i) \) 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 \( E_{a_n} \) holds for the store \( \rho \) of \( s \). If this is not the case then \( \text{tr}_{hp}^{in}(\rho, C_i) \), after a suitable unfolding step for \( (\rho, C_i) \), jumps back to the \( out \)-modality by progressing with \( \text{tr}_{hp}^{out}(\rho) \).

It turns out that \( \text{tr}_{hp}^{out} \) maps traces of \( P \) into traces of \( P_{hp} \) and does not alter store change sequences.

Lemma 6.3. \( \text{tr}_{hp}^{out} \) is well-defined and for any \( \sigma \in \text{Trace}_P \) \( sc(\text{tr}_{hp}^{out}(\sigma)) = sc(\sigma) \).

Proof. We first show that: (1) \( \text{tr}_{hp}^{out} \) is well-defined, i.e., for any \( \sigma \in \text{Trace}_P \), \( \text{tr}_{hp}^{out}(\sigma) \in \text{Trace}_{P_{hp}} \), and (2) for any \( \sigma \in \text{Trace}_P \), if \( \text{cmd}(\sigma) \notin \{C_0, \text{cmdpl}(C_0)\} \) then \( \text{tr}_{hp}^{in}(\sigma) \in \text{Trace}_{P_{hp}} \). In order to prove these two points, it is enough an easy induction on the length of the execution trace \( \sigma \) and to observe that:

(i) for the first four clauses that define \( \text{tr}_{hp}^{out}(\sigma) \) in Fig. 3 we have that \( \text{tr}_{hp}^{out}(\sigma) = s' s'' \text{tr}_{hp}^{in}(\sigma) \) or \( \text{tr}_{hp}^{out}(\sigma) = s' s'' \text{tr}_{hp}^{out}(\sigma) \), where \( s' \) is a guard command of \( P_{hp} \) and \( s'' \) is a legal sub-execution trace of \( P_{hp} \);

(ii) for the last clause that defines \( \text{tr}_{hp}^{out}(\sigma) \) in Fig. 3 we have that \( \text{cmd}(s) \notin \{C_0, \text{cmdpl}(C_0)\} \), hence \( s \) is a legal state in \( P_{hp} \) and, in turn, \( \text{tr}_{hp}^{out}(\sigma) = s \cdot \text{tr}_{hp}^{out}(\sigma) \) is a trace of \( P_{hp} \);

(iii) for the clauses 1, 2 and 4 that define \( \text{tr}_{hp}^{in}(\sigma) \) in Fig. 3 we have that \( \text{tr}_{hp}^{in}(\sigma) = s' s'' \text{tr}_{hp}^{in}(\sigma) \) or \( \text{tr}_{hp}^{in}(\sigma) = s' s'' \text{tr}_{hp}^{out}(\sigma) \), where \( s' \) is a guard command and \( s'' \) is an action command such that \( s' s'' \) is a legal sub-execution trace of \( P_{hp} \);
\[
\text{tr}_{hp}^{out}(\epsilon) \triangleq \epsilon
\]
\[
\begin{cases}
(\rho, L_0 : \text{guard } E_{ao} \rightarrow \ell_0)(\rho, \ell_0 : \text{act}(C_0) \rightarrow 1_1) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, C_0), \alpha_{\text{store}}(\{\rho\}) \leq a_0 \\
(\rho, L_0 : \neg \text{guard } E_{ao} \rightarrow L_0)(\rho, \ell_0 : \text{act}(C_0) \rightarrow L_1) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, C_0), \alpha_{\text{store}}(\{\rho\}) \not\leq a_0 
\end{cases}
\]
\[
\text{tr}_{hp}^{out}(s) \triangleq \begin{cases}
(\rho, L_0 : \text{guard } E_{ao} \rightarrow \ell_0)(\rho, \ell_0 : \neg \text{act}(C_0) \rightarrow L_1) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, C_0), \alpha_{\text{store}}(\{\rho\}) \leq a_0 \\
(\rho, L_0 : \neg \text{guard } E_{ao} \rightarrow L_0)(\rho, \ell_0 : \neg \text{act}(C_0) \rightarrow L_1) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, C_0), \alpha_{\text{store}}(\{\rho\}) \not\leq a_0 \\
\end{cases}
\]
\[
s \cdot \text{tr}_{hp}^{out}(\sigma)
\]

\[
\text{tr}_{hp}^{in}(\epsilon) \triangleq \epsilon
\]
\[
\begin{cases}
(\rho, \ell_i : \text{guard } E_{a_i} \rightarrow \ell_{i})(\rho, \ell_i : \text{act}(C_i) \rightarrow 1_{i+1}) \text{tr}_{hp}^{in}(\sigma) & \text{if } s = (\rho, C_i), i \in [1, n - 1], \alpha_{\text{store}}(\{\rho\}) \leq a_i \\
(\rho, \ell_n : \text{guard } E_{a_n} \rightarrow \ell_n)(\rho, \ell_n : \text{act}(C_n) \rightarrow L_0) \text{tr}_{hp}^{in}(\sigma) & \text{if } s = (\rho, C_n), \alpha_{\text{store}}(\{\rho\}) \leq a_n 
\end{cases}
\]
\[
\text{tr}_{hp}^{in}(s) \triangleq \begin{cases}
(\rho, \ell_i : \neg \text{guard } E_{a_i} \rightarrow L_i)(\rho, C_i) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, C_i), i \in [1, n], \alpha_{\text{store}}(\{\rho\}) \not\leq a_i \\
(\rho, \ell_i : \text{guard } E_{a_i} \rightarrow \ell_i)(\rho, \ell_i : \neg \text{act}(C_i) \rightarrow L^c_{\text{new}(i)}) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, \text{cmpl}(C_i)), i \in [1, n], \alpha_{\text{store}}(\{\rho\}) \leq a_i \\
(\rho, \ell_i : \neg \text{guard } E_{a_i} \rightarrow L_i)(\rho, \text{cmpl}(C_i)) \text{tr}_{hp}^{out}(\sigma) & \text{if } s = (\rho, \text{cmpl}(C_i)), i \in [1, n], \alpha_{\text{store}}(\{\rho\}) \not\leq a_i \\
\end{cases}
\]
\[
s \cdot \text{tr}_{hp}^{out}(\sigma)
\]

Figure 3: Definitions of \(\text{tr}_{hp}^{out}\) and \(\text{tr}_{hp}^{in}\).
Hence, the above points also show that the sequence of store changes is not affected by $rtr$.

Proof.

Lemma 6.4. $\forall rtr(\sigma)$

Proof of Theorem 6.2. With a slight abuse of notation for $rtr$, let us define two functions $tr_{hp} : \wp(\text{Trace}_{P_{hp}}) \rightarrow \wp(\text{Trace}_{P_{hp}})$ and $rtr_{hp} : \wp(\text{Trace}_{P_{hp}}) \rightarrow \wp(\text{Trace}_{P_{hp}})$ which are the collecting versions of $tr_{hp}$ and $rtr_{hp}$, that is,
\[ \text{tr}_{hp}(T) \triangleq \{ \text{tr}_{hp}^\text{out}(\sigma) \mid \sigma \in T \} \] and \[ \text{rtr}_{hp}(T) \triangleq \{ \text{rtr}_{hp}(\sigma) \mid \sigma \in T \} \]. As consequences of the above lemmata, we have the following properties.

(A) \( \alpha_{sc} \circ \text{tr}_{hp} = \alpha_{sc} \); By Lemma 6.3

(B) \( \text{tr}_{hp}(T[P]) \subseteq T[P_{hp}] \); because, by Lemma 6.3, \( \text{tr}_{hp}^\text{out} \) is well-defined.

(C) \( \alpha_{sc} \circ \text{rtr}_{hp} = \alpha_{sc} \); By Lemma 6.4

(D) \( \text{rtr}_{hp}(T[P_{hp}]) \subseteq T[P] \); because, by Lemma 6.4, \( \text{rtr}_{hp} \) is well-defined.

We therefore obtain:

\[ \alpha_{sc}(T[P]) = \]\[ \text{By point (A)} \]
\[ \alpha_{sc}(\text{tr}_{hp}(T[P])) \subseteq \]\[ \text{By point (B)} \]
\[ \alpha_{sc}(T[P_{hp}]) = \]\[ \text{By point (C)} \]
\[ \alpha_{sc}(\text{rtr}_{hp}(T[P_{hp}])) \subseteq \]\[ \text{By point (D)} \]

and this closes the proof.

7 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 [Ecma International, 2011, Section 11.6]) and of the semantics of \( + \) in our language as given in Section 2.3. 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.

7.1 Type Abstraction

Let us recall that the set of type names is \( \text{Types} = \{ \text{Int}, \text{String}, \text{Undef}, \text{Any}, \emptyset \} \). Type names can therefore be viewed as the following finite lattice \( \langle \text{Types}, \sqsubseteq \rangle \):

```
\[
\begin{array}{ccc}
\text{Int} & \text{Any} & \text{String} \\
\emptyset & \text{Undef} \\
\end{array}
\]
```

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

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

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

Following the general approach described in Section 3.2.1, we consider a simple nonrelational store abstraction for types

\[ \text{Store}^t \triangleq \{ \text{Var} \to \text{Types}, \sqsubseteq \} \]

where \( \sqsubseteq \) is the standard pointwise lifting of the ordering \( \subseteq \) for \( \text{Types} \), so that \( \lambda x. \emptyset \) and \( \lambda x. \text{Any} \). Any are, respectively, the bottom and top abstract stores in \( \text{Store}^t \). The abstraction and concretization maps \( \alpha_{\text{store}} : \wp(\text{Store}) \to \text{Store}^t \) and \( \gamma_{\text{store}} : \text{Store}^t \to \wp(\text{Store}) \) are defined as a straight instantiation of the definitions in Section 3.2.1.
The abstract type semantics $E^t : \text{Exp} \to \text{Store}^t \to \text{Types}$ of expressions is defined as best correct approximation of the corresponding concrete semantics $E$ on the type abstractions $\text{Store}^t$ and $\text{Types}$, i.e.,

$$E^t[E]\rho^t \triangleq \alpha_{\text{type}}(E[E]_{\gamma_{\text{store}}}(\rho^t)).$$

This definition leads to the following equalities:

$$E^t[v]\rho^t = \text{type}(v)$$
$$E^t[x]\rho^t = \rho^t(x)$$
$$E^t[E_1 + E_2]\rho^t = \begin{cases} \emptyset & \text{if } \exists i. E^t[E_i]\rho^t = \emptyset \\
E^t[E_1]\rho^t & \text{else if } E^t[E_1]\rho^t = E^t[E_2]\rho^t \in \{\text{Int}, \text{String}\} \\
\text{Undefined} & \text{else if } \forall i. E^t[E_i]\rho^t < \text{Any} \\
\text{Any} & \text{otherwise}
\end{cases}$$

For instance, we have that:

$$E^t[x + y][x/\text{String}, y/\emptyset] = \emptyset,$$
$$E^t[x + y][x/\text{String}, y/\text{String}] = \text{String},$$
$$E^t[x + y][x/\text{Int}, y/\text{String}] = \text{Undefined},$$
$$E^t[x + y][x/\text{Int}, y/\text{Any}] = \text{Any}.$$

It turns out that the abstract type semantics $E^t$ of expressions is correct by definition.

**Corollary 7.1.** If $\rho \in \gamma_{\text{store}}(\rho^t)$ then $E[E]\rho \in E^t[E]\rho^t$.

According to Section 5, for any abstract type store $[x_i/T_i \mid x_i \in \text{Var}] \in \text{Store}^t$ we consider a corresponding Boolean action guard denoted by

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

whose corresponding action semantics is automatically induced, as defined in Section 5 by the Galois connection $(\alpha_{\text{store}}, \varphi(\text{Store}), \text{Store}^t, \gamma_{\text{store}})$: for any $\rho \in \text{Store}$,

$$A[\text{guard } x_0 : T_0, \ldots, x_n : T_n]\rho \triangleq \begin{cases} \rho & \text{if } \rho \in \gamma_{\text{store}}([x_i/T_i \mid x_i \in \text{Var}]) \\
\bot & \text{otherwise}
\end{cases}$$

For example,

$$A[\text{guard } x : \text{String}, y : \text{String}][x/\text{foo, bar}, y/\emptyset] = [x/\text{foo}, y/\text{bar}],$$
$$A[\text{guard } x : \text{String}, y : \text{Any}][x/\text{foo, bar}, y/3] = [x/\text{foo, bar}, y/3],$$
$$A[\text{guard } x : \text{String}, y : \text{Any}][x/1, y/3] = \bot.$$

### 7.2 Type Specialization of Hot Paths

Let us consider some hot path $hp = \langle \rho_0^t, C_0, \ldots, \rho_n^t, C_n \rangle \in \alpha'_{\text{hot}}(\text{Trace}_P)$ on the type abstraction $(\text{Store}^t, \subseteq)$, where each $\rho_i^t$ is therefore a type environment for $P$. Thus, in the transformed program $P_{hp} \triangleq \text{extr}_hp(P)$, the stitched hot path $\text{stitch}_p(hp)$ contains $n + 1$ typed guards, that, for any $i \in [0, n]$, we simply denote as $\text{guard } \rho_i^t$.

Typed guards allow us to perform type specialization of commands in the stitched hot path. In order to keep the notation simple, we only focus on type specialization of addition operations occurring in assignments. 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 $\text{stitch}_p(hp)$. Note that if $C \in \text{stitch}_p(hp)$ and 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 $L' \in \{1_{i+1}, L_0\}$. Let $C'$ denote the extended set of commands that permit type specific additions $+\text{Int}$ and $+\text{String}$ and, in turn, $\text{Program}'$ denote the possibly type-specialized programs over $C'$. The semantic function $E$ for expressions is then updated to type specific additions as follows:

$$E[E_1 + \text{Int} E_2]_{\rho} \triangleq \begin{cases} E[E_1]_{\rho} + E[E_2]_{\rho} & \text{if } \text{type}(E[E_1]_{\rho}) = \text{Int} \\
\text{undefined} & \text{otherwise}
\end{cases}$$

$$E[E_1 + \text{String} E_2]_{\rho} \triangleq \begin{cases} E[E_1]_{\rho} \cdot E[E_2]_{\rho} & \text{if } \text{type}(E[E_1]_{\rho}) = \text{String} \\
\text{undefined} & \text{otherwise}
\end{cases}$$
The type specialization function $ts_{hp} : P_{hp} \rightarrow C'$ for the hot path $hp$ is defined as follows:

$$ts_{hp}(\ell_i : x := E_1 + E_2 \rightarrow L') \triangleq \begin{cases} \ell_i : x := E_1 + \text{Int} \ E_2 \rightarrow L' & \text{if } \mathbf{E}[E_1 + E_2][\rho]' = \text{Int} \\ \ell_i : x := E_1 + \text{String} \ E_2 \rightarrow L' & \text{if } \mathbf{E}[E_1 + E_2][\rho]' = \text{String} \\ \ell_i : x := E_1 + E_2 \rightarrow L' & \text{otherwise} \end{cases}$$

$$ts_{hp}(C) \triangleq C \quad \text{if } C \neq \ell_i : x := E_1 + E_2 \rightarrow L'$$

Hence, if a typed guard guard $\rho_i'$ preceding a command $\ell_i : x := E_1 + E_2 \rightarrow L'$ allows us to derive abstractly on Store$^t$ that $E_1$ and $E_2$ have the same type (Int or String) then the addition $E_1 + E_2$ is accordingly type specialized.

Typed trace extraction $extr^t_{hp}(P)$, also denoted by $P^t_{hp}$, consists in extracting and simultaneously type specializing a typed hot path $hp$ in a program $P$, i.e., this is defined as the $ts_{hp}$-image of the transformed program $extr_{hp}(P)$:

$$extr^t_{hp}(P) \triangleq ts_{hp}(extr_{hp}(P)).$$

Since $ts_{hp}$ may transform only commands in $stitch_{hp}(hp)$, notice that, according to Definition $[5.1]$ we have that:

$$extr^t_{hp}(P) = extr_{hp}(P) \setminus stitch_{hp}(hp) \cup ts_{hp}(stitch_{hp}(hp)).$$

The correctness of this program transform can be stated for the store changes observational abstraction as follows.

**Theorem 7.2 (Correctness of typed trace extraction).** For any typed hot path $hp \in \alpha_{hot}^N(\text{Trace}_P)$, we have that $\alpha_{sc}(\mathbf{T}[extr^t_{hp}(P)]) = \alpha_{sc}(\mathbf{T}[P])$.

**Proof.** We define a type despecialization function $td : \text{Trace}^{\mathcal{P}_p}_{hp} \rightarrow \text{Trace}^{\mathcal{P}_p}_{hp}$ through the following inductive clauses, where $\text{Type}$ is either Int or String:

$$td(\epsilon) \triangleq \epsilon$$

$$td(s\sigma) \triangleq \begin{cases} \langle \rho, \ell_i : x := E_1 + E_2 \rightarrow L' \rangle & \text{if } s = \langle \rho, \ell_i : x := E_1 + \text{Type} \ E_2 \rightarrow L' \rangle, \\ type(\mathbf{E}[E_1 + E_2][\rho]) \neq \text{Type} & \\ \langle \rho, \ell_i : x := E_1 + E_2 \rightarrow L' \rangle \cdot td(\sigma) & \text{if } s = \langle \rho, \ell_i : x := E_1 + \text{Type} \ E_2 \rightarrow L' \rangle, \\ type(\mathbf{E}[E_1 + E_2][\rho]) = \text{Type} & \text{otherwise} \end{cases}$$

Let us explain the first defining clause of $td(s\sigma)$, namely, $s = \langle \rho, \ell_i : x := E_1 + \text{Type} \ E_2 \rightarrow L' \rangle$ and type$\mathbf{E}[E_1 + E_2][\rho] \neq \text{Type}$. These conditions can never hold in an inductive call of the function $td$: in fact, when $td(s\sigma)$ is recursively called by $td(s'\sigma')$, we necessarily have that $s' = \langle \rho, I : \text{guard} \rho_i' \rightarrow \ell_i \rangle$, so that $\rho \in \gamma\text{store}(\rho_i')$ and, in turn, by Corollary $[7.1]$ $\mathbf{E}[E_1 + E_2][\rho] \neq \mathbf{E}[E_1 + E_2][\rho]'$, which implies type$\mathbf{E}[E_1 + E_2][\rho] = \text{Type}$, that is, a contradiction. Thus, the first defining clause of $td(s\sigma)$ only applies to type specialized traces in $\text{Trace}^{\mathcal{P}_p}_{hp}$ whose first state is $s = \langle \rho, \ell_i : x := E_1 + \text{Type} \ E_2 \rightarrow L' \rangle$: in this case, we necessarily have that $\sigma = \epsilon$, because $\mathbf{A}[E_1 + \text{Type} E_2][\rho] = \text{undef}$ so that $\mathbf{S}s = \emptyset$. This clarifies the definition of $td$ in this particular case. Also, observe that in this case, $\alpha_{sc}(td(s)) = \alpha_{sc}(s)$ trivially holds. In all the remaining cases, it is clear that $td$ maps type specialized traces of $P^t_{hp}$ into legal unspecialized traces of $P_{hp}$ since labels are left unchanged. Moreover, $\alpha_{sc} \circ td = \alpha_{sc}$ holds, in particular because in the second defining clause of $td(s\sigma)$, the condition type$\mathbf{E}[E_1 + E_2][\rho] = \text{Type}$ guarantees that $\mathbf{E}[E_1 + E_2][\rho] = \mathbf{E}[E_1 + \text{Type} E_2][\rho]$.

On the other hand, we first define a type specialization function $sp : \text{Trace}^{\mathcal{P}_p}_{hp} \rightarrow (\text{State}^{\mathcal{P}_p}_{hp})^*$ as follows:

$$sp(\epsilon) \triangleq \epsilon$$

$$sp(\langle \rho, C \rangle \sigma) \triangleq \langle \rho, ts_{hp}(C) \rangle \cdot sp(\sigma)$$

Hence, $sp(\sigma)$ simply type specializes through $ts_{hp}$ all the commands occurring in the trace $\sigma$ of $P_{hp}$. In general, $sp(\langle \rho, C \rangle \sigma)$ is not guaranteed to be a legal trace of $P^t_{hp}$ because there is no check that the type abstraction of the store $\rho$ is in accordance with the type instantiation of addition operations of $ts_{hp}(C)$. In turn, $sp$ allows us to define a type specialization function $ts : \text{Trace}_P \rightarrow \text{Trace}^{\mathcal{P}_p}_{hp}$ which uses the function $tr^{out}_{hp} : \text{Trace}_P \rightarrow \text{Trace}^{\mathcal{P}_p}_{hp}$ of Lemma $[6.3]$

$$ts(\sigma) \triangleq sp(tr^{out}_{hp}(\sigma)).$$

It turns out that the function $ts$ is well defined, i.e., $sp(tr^{out}_{hp}(\sigma))$ is a legal trace of $P^t_{hp}$. In fact, by Lemma $[6.3]$ $tr^{out}_{hp}(\sigma)$ is a legal trace of $P_{hp}$ where any state $\langle \rho, \ell_i : x := E_1 + E_2 \rightarrow L' \rangle$ is always preceded by the state $\langle \rho, I_i : x := E_1 + E_2 \rightarrow L' \rangle$. In fact, by Lemma $[6.3]$}
\text{guard } ρ^t_i \to ℓ_i \text{ and } ρ \in γ_{store}(ρ^t_i) \text{ holds. Thus, by Corollary } 7.1 \ E[E_1 + E_2]|ρ \in E'[E_1 + E_2]|ρ^t = \text{Type}, \text{ so that } A[x := E_1 + \text{Type } E_2]|ρ \cong A[x := E_1 + E_2]|ρ \text{ holds. Consequently, the trace fragment }
sp(⟨ρ, ℓ_i : \text{guard } ρ^t_i \to ℓ_i⟩ | ρ, ℓ_i : x := E_1 + E_2 \to L′) = 〈ρ, ℓ_i : \text{guard } ρ^t_i \to ℓ_i⟩ (ρ, ℓ_i : x := E_1 + \text{Type } E_2 \to L′)
is legal in } P^t_i. \text{ This allows us to obtain, by an easy induction on } |σ|, \text{ that } sp(tr^\text{int}_hp(σ)) ∈ Trace^t_{hp}. \text{ Moreover, observe that } α_{sc} \circ ts = α_{sc} \text{ trivially holds. We therefore obtain: }

\begin{align*}
α_{sc}(T[extr^t_{hp}(P)]) &= [\text{since } α_{sc} \circ td = α_{sc}] \\
α_{sc}(td(T[extr^t_{hp}(P)])) \subseteq [\text{since } td \text{ is well defined}] \\
α_{sc}(T[extr^t_{hp}(P)]) &= [\text{by Theorem } 6.2] \\
α_{sc}(ts(T[extr^t_{hp}(P)])) \subseteq [\text{since } ts \text{ is well defined}] \\
α_{sc}(T[extr^t_{hp}(P)])
\end{align*}

and this closes the proof. □

\textbf{Example 7.3.} \text{Let us consider the following sieve of Eratosthenes in a Javascript-like language – this is taken from the running example in } [\text{Gal et al., 2009]} – \text{ where primes is an array initialized with 100 true values.}

\textbf{for} (\text{var } i = 2; \ i < 100; \ i = i + 1) \text{ do }
\begin{align*}
&\text{if (}!\text{primes}[i]\text{) then continue;} \\
&\text{for (}\text{var } k = i + i; \ k < 100; \ k = k + i) \text{ do primes}[k] = false;
\end{align*}

With a slight abuse, we assume that our language is extended with arrays and Boolean values ranging in the type Bool. The semantics of read and store for arrays is standard: 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 then the corresponding action command gives an \textit{undefined} value, that is, we assume that the program is aborted. The above program is encoded in our language as follows:

\begin{align*}
P = \{ \begin{array}{ll}
C_0 \equiv L_0 : i := 2 \to L_1, & C_1 \equiv L_1 : i < 100 \to L_2, \\
C_2 \equiv L_2 : \text{primes}[i] = \text{tt} \to L_3, & C_3 \equiv L_3 : ¬(\text{primes}[i] = \text{ff}) \to L_4, \\
C_4 \equiv L_4 : k := i + i \to L_5, & C_5 \equiv L_5 : ¬(k < 100) \to L_6, \\
C_6 \equiv L_6 : k := k + i \to L_7, & C_7 \equiv L_7 : i := i + 1 \to L_8, \\
C_8 \equiv L_8 : \text{skip} \to L_9. & \end{array} \}
\end{align*}

Let us consider the following type environment

\[ ρ^t \cong \{ \text{primes}[n]/\text{Bool}, i/\text{Int}, k/\text{Int} \} \in \text{Store}^t \]

where \text{primes}[n]/\text{Bool} is a shorthand for \text{primes}[0]/\text{Bool}, \ldots, \text{primes}[99]/\text{Bool}. Then the first traced 2-hot path on the type abstraction \text{Store}^t \text{ is } hp_1 \cong (ρ', C_4, ρ^t, C_5, ρ^t, C_6). \text{ As a consequence, the typed trace extraction of } hp_1 \text{ yields:

\begin{align*}
P_1 \equiv extr^t_{hp_1}(P) = P \setminus \{ C_4, C_5 \} \cup \{ \tau_4 : k < 100 \to L_5, \tau_4 : ¬(k < 100) \to L_7 \} \cup \text{ts}_{hp}(\text{stitch}_{P}(hp))
\end{align*}

where:

\[ \text{ts}_{hp}(\text{stitch}_{P}(hp)) = \{ H_0 \equiv L_4 : \text{guard } (\text{primes}[n]) \rightarrow \text{Bool}, i : \text{Int}, k : \text{Int} \rightarrow ℓ_0, \\
H_0^t \equiv L_4 : ¬\text{guard } (\text{primes}[n]) \rightarrow \text{Bool}, i : \text{Int}, k : \text{Int} \rightarrow ℓ_4, \\
H_1 \equiv ℓ_0 : k < 100 \to ℓ_1, H_1^t \equiv ℓ_0 : ¬(k < 100) \to ℓ_7, \\
H_2 \equiv ℓ_1 : \text{guard } (\text{primes}[n]) \rightarrow \text{Bool}, i : \text{Int}, k : \text{Int} \rightarrow ℓ_1, \\
H_2^t \equiv ℓ_1 : ¬\text{guard } (\text{primes}[n]) \rightarrow \text{Bool}, i : \text{Int}, k : \text{Int} \rightarrow ℓ_1, \\
H_3 \equiv ℓ_1 : \text{primes}[k] := \text{ff} \rightarrow ℓ_2, \\
H_4 \equiv ℓ_2 : \text{guard } (\text{primes}[n]) \rightarrow \text{Bool}, i : \text{Int}, k : \text{Int} \rightarrow ℓ_2, \\
H_4^t \equiv ℓ_2 : ¬\text{guard } (\text{primes}[n]) \rightarrow \text{Bool}, i : \text{Int}, k : \text{Int} \rightarrow ℓ_6, \\
H_5 \equiv ℓ_2 : k := k + \text{Int } i \rightarrow ℓ_4. \}
\]
8 A General Correctness Criterion

One major advantage of using abstract interpretation is that in our approach the type specialization procedure defined in Section[9] can be viewed as a particular correct hot path optimization and therefore it can be easily generalized.

Guarded hot paths are a key feature of our tracing compilation model, where guards are dynamically recorded by a hot path monitor and range over abstract values in some store abstraction. An abstract guard for a command \( C \) of some hot path \( hp \) thus encodes a store property which is modeled in some abstract domain \( \text{Store}^\delta \) 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 \( stitch_P(hp) \). 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 \( \alpha_o \) of trace semantics and Theorem 6.2 ensures that this correctness holds for the store changes abstraction \( \alpha_sc \) of the unoptimized trace extraction transform, we just need to prove the correctness of the optimization \( O \) on the stitched hot path \( stitch_P(hp) \), 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

\[
O : \{\text{stitch}_P(hp) | hp \in \alpha_n^N(\text{Trace}_P)\} \rightarrow \text{Program}
\]

where Program may permit new expressions and/or actions, as in the case of type-specific additions \(+\text{Type}\) in type specialization \( ts_{hp} \). The program transform \( O \) is therefore required to be correct according to the following definition.

Definition 8.1 (Correctness of hot path optimization). \( O \) is correct if for any \( P \in \text{Program} \) and for any \( hp \in \alpha_n^N(\text{Trace}_P), \alpha_o(T[O(stitch_P(hp))] = \alpha_o(T[stitch_P(hp)]). \)

As an example, it would not be hard to formalize the variable folding optimization of hot paths considered by Guo and Palsberg [2011] and to prove it correct in our framework w.r.t. the store changes abstraction \( \alpha_sc \).

9 Nested Hot Paths

Once a first hot path \( hp_1 \) has been extracted by transforming \( P \) to \( P_1 \overset{\text{extr}_{hp_1}}{=} P \), 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 [Gal et al. 2009], we attempt to nest an inner hot path inside the current trace: during trace recording, an inner hot path is called as a kind of “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[4]. Let \( P \) be the original program and let \( P’ \) be a hot path transform of \( P \) so that \( P’ \setminus P \) contains all the commands (guards included) in the hot path. We define a function \( \text{hotcut} : \text{Trace}_P \rightarrow (\text{State}_P)^* \) that cuts from an execution trace \( \sigma \) of \( P’ \) all the states whose commands appear in some previous hot path \( hp \) except for the entry and exit states of \( hp \):

\[
\text{hotcut}(\sigma) \triangleq \begin{cases} 
\epsilon & \text{if } \sigma = \epsilon \\
\text{hotcut}(\langle p_1, C_1 \rangle, \langle p_3, C_3 \rangle \sigma') & \text{if } \sigma = \langle p_1, C_1 \rangle, \langle p_2, C_2 \rangle, \langle p_3, C_3 \rangle \sigma' & C_1, C_2, C_3 \not\in P \\
\sigma_0 \text{hotcut}(\sigma_1 \cdots \sigma_{|\sigma|-1}) & \text{otherwise}
\end{cases}
\]

In turn, we define \( \text{outerhot}^N : \text{Trace}_P \rightarrow \wp((\text{State}_P)^*) \) as follows:

\[
\text{outerhot}^N(\sigma) \triangleq \{ \langle a_i, C_i \rangle \cdots \langle a_j, C_j \rangle \in (\text{State}_P)^* \mid \exists \langle p_i, C_i \rangle \cdots \langle p_j, C_j \rangle \in \text{loop(hotcut}(\sigma)) \text{ such that } i \leq j,\alpha_{\text{store}}(\langle p_i, C_i \rangle \cdots \langle p_j, C_j \rangle) = \langle a_i, C_i \rangle \cdots \langle a_j, C_j \rangle, \text{count}(\alpha_{\text{store}}(\text{hotcut}(\sigma))) \geq N \}.
\]

Clearly, when \( P’ = P \) we have that \( \text{hotcut} = \lambda \sigma.\sigma \) so that \( \text{outerhot}^N = \text{hot}^N \). Finally, we define the collecting version on \( \wp(\text{Trace}_P) \) as the abstraction map \( \text{outerhot}^N \triangleq \lambda T.\cup_{\sigma \in T} \text{outerhot}^N(\sigma) \).

Example 9.1. Let us consider again Example[5.3] where \( \text{Store}^\delta \) is the trivial one-point store abstraction \( \{\top\} \). In Example[5.3] we first extracted \( hp_1 = (\top, C_1, \top, C_2, \top, C_3) \) by transforming \( P \) to \( P_1 \overset{\text{extr}_{hp_1}}{=} P \).

We then consider the following trace in \( T[P_1] \):

\[
\sigma = (\langle x/?, C_0 \rangle, \langle x/0, H_0 \rangle, \langle x/0, H_1 \rangle, \langle x/0, H_2 \rangle, \langle x/0, H_3 \rangle, \langle x/1, H_4 \rangle, \langle x/1, H_5 \rangle, \langle x/2, H_6 \rangle, \langle x/3, H_7 \rangle, \langle x/3, H_8 \rangle, \langle x/3, C_4 \rangle, \langle x/6, H_0 \rangle, \langle x/9, H_9 \rangle, \langle x/9, H_9 \rangle, \langle x/12, H_0 \rangle, \cdots
\]
Thus, here we have that

\[
\text{hotcut}(\sigma) = \langle \langle x/3 \rangle, C_0 \rangle, \langle x/0 \rangle, H_0 \rangle, \langle x/3 \rangle, C_4 \rangle, \langle x/6 \rangle, H_0 \rangle, \langle x/9 \rangle, C_4 \rangle, \ldots
\]

so that \( hp_2 = \langle \top, H_0, \top, H_0^c, \top, C_4 \rangle \in \alpha^2_{\text{outerhat}}(T[P_1]) \). 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 \( H_0^c \).

Let \( hp = \langle a_0, C_0, \ldots, a_n, C_n \rangle \in \alpha_N^{\text{outerhat}}(T[P]) \) be a \( N \)-hot path in \( P' \), where, for all \( i \in [0, n] \), we assume that \( C_i \equiv L_i : A_i \rightarrow L_{\text{next}(i)} \). 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., \( hp \in \alpha_{\text{hot}}(T[P]) \).

- Otherwise, there exists some \( C_k \not\in P \). If \( C_i \in P \) and \( C_{i+1} \not\in P \) then \( C_{i+1} \) is the entry command of some inner hot path; on the other hand, if \( C_i \not\in P \) 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 then given as a generalization of Definition 5.1.

**Definition 9.2 (Nested trace extraction transform).** The nested trace extraction transform of \( P' \) for the hot path \( hp \) is:

\[
\text{extr}_{hp}(P') \equiv (3) \cup (4) \cup (5) \cup (6) \cup (7) \cup (9).
\]

Let us observe that:

- Clauses (1)–(6) are the same clauses of Definition 5.1 with the additional constraints that the commands \( C_i \) and \( \text{cmpl}(C_i) \) are in \( P \), that is, \( C_i \) and \( \text{cmpl}(C_i) \) are not the entry or exit commands of a nested hot path. This condition is trivially satisfied in Definition 5.1.

- Clause (7) where \( C_i \in P \) and \( C_{i+1} \not\in P \), namely \( \text{next}(C_i) \) 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 \( C_i \not\in P \) 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 \( \text{lbl}(C_{i+1}) \), performs the relabeling of \( \text{suc}(C_i) \) in \( C_i \) in order to return from \( nhp \) to \( hp \);

- \( \overline{H}_i, h_i \) and \( i_i \) are meant to be fresh labels, i.e., they have not been already used in \( P' \).

**Example 9.3.** Let us go on with Example 9.1. The second traced hot path in \( \alpha^2_{\text{outerhat}}(T[P_1]) \) is:

\[
hp_2 = \langle \top, H_0 \equiv L_1 : \text{guard } E_T \rightarrow \ell_0, \top, H_0^c \equiv \ell_2 : (x \% 3 = 0) \rightarrow L_4, \top, C_4 \equiv L_4 : x := x + 3 \rightarrow L_4 \rangle.
\]

According to Definition 9.2 trace extraction of \( hp_2 \) in \( P_1 \) yields the following transform:

\[
\text{extr}_{hp_2}(P_1) \equiv (8) \cup \{\ell_2 : (x \% 3 = 0) \rightarrow \ell_2\}
\]

where we used the additional fresh labels \( \ell_2 \) and \( \ell_2 \).
Example 9.4. Let us consider again Example 7.3. After the trace extraction of $hp_1$ that transforms $P$ to $P_1$, a second traced 2-hot path is the following:

$$hp_2 \triangleq \langle \rho', C_1, C_2, \rho', C_3, \rho', H_0, \rho', H_1, \rho', C_4 \rangle$$

where $\rho' = \{ \text{primes}[n]/\text{Bool}, i/\text{Int}, k/\text{Int} \} \in \text{Store}^\ell$. $hp_2$ contains a nested hot path which is called at $\text{suc}(C_3) = L_4$ and whose entry and exit commands are, respectively, $H_0$ and $H_1$. Here, typed trace extraction according to Definition 9.2 provides the following transform of $P_1$:

$$P_2 \triangleq \text{extr}_{hp_2}^t(P_1) = P_1 \setminus \{ C_1, C_2 \} \cup \{$$

$$\begin{align*}
\overline{H_0} & : i < 100 \rightarrow L_2, \overline{H_0} : \neg(i < 100) \rightarrow L_8, \\
H_0 & : L_1 : \text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow h_0, \\
H_6 & : L_1 : \neg\text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow \overline{H_0}, \\
H_5 & : h_0 : i < 100 \rightarrow \overline{H_1}, H_5 & : \neg(i < 100) \rightarrow L_8, \\
H_6 & : L_1 : \text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow h_1, \\
H_4 & : L_1 : \neg\text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow L_2, \\
H_8 & : h_1 : \text{primes}[i] = tt \rightarrow \overline{H_1}, H_8 & : \neg(\text{primes}[i] = tt) \rightarrow L_7, \\
H_{10} & : h_2 : \text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow h_2, \\
H_{10} & : h_2 : \neg\text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow L_3, \\
H_{10} & : h_2 : k := i + \text{Int} i \rightarrow L_4 \\
\} \setminus \{ H_1 \cup \{ (H_1')' \equiv L_0 : \neg(k < 100) \rightarrow H_3, \\
H_1 & : \text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow h_3, \\
H_3 & : \neg(\text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow L_7, \\
H_3 & : h_3 : i := i + \text{Int} 1 \rightarrow L_1 \} \}.
\end{align*}$$

Finally, a third traced 2-hot path in $P_2$ is $hp_3 \triangleq \langle \rho', H_0, \rho', H_0', \rho', C_4 \rangle$ which contains a nested hot path which is called at the beginning of $hp_3$ and whose entry and exit commands are, respectively, $H_0$ and $H_0'$. Here, typed trace extraction of $hp_3$ yields:

$$P_3 \triangleq \text{extr}_{hp_3}^t(P_2) = P_2 \setminus \{ H_0' \} \cup \{ (H_0')' \equiv h_1 : \neg(\text{primes}[i] = tt) \rightarrow j_2, \\
j_2 : \text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow j_2, \\
j_2 : \neg\text{guard} (\text{primes}[n] : \text{Bool}, i : \text{Int}, k : \text{Int}) \rightarrow L_7, \\
j_2 : i := i + \text{Int} 1 \rightarrow L_1 \}.$$

We have thus obtained the same three trace extraction steps as described by Gal et al. [2009 Section 2]. In particular, in $P_1$ we specialized the typed addition operation $\text{k + Int } i$, in $P_2$ we specialized $i + \text{Int } i$ and $i + \text{Int } 1$, while in $P_3$ we specialized once again $i + \text{Int } 1$ in a different hot path. Thus, in $P_3$ all the addition operations occurring in assignments have been type specialized. 

10 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 [2011]. Its main distinctive feature is the use of a bisimulation relation [Milner, 1995] to model the operational equivalence between source and optimized programs. In this section, we show how this model can be expressed within our framework.

10.1 Language and Semantics

Guo and Palsberg [2011] rely on a simple imperative language (without jumps and) with while loops and a so-called bail construct. Its syntax is as follows:

$$\begin{align*}
E :&= v \mid x \mid E_1 + E_2 \\
B :&= \text{tt} \mid \text{ff} \mid \text{if } E_1 \leq E_2 \mid \neg B \mid B_1 \land B_2 \\
\text{Cmd} \ni c :&= \text{skip} \mid x := E ; \mid \text{if } B \text{ then } S \mid \text{while } B \text{ do } S \mid \text{bail } B \text{ to } S \\
\text{Stm} \ni S :&= \epsilon \mid cS
\end{align*}$$
where $\epsilon$ stands for the empty string. Thus, any statement $S \in \text{Stm}$ is a (possibly empty) sequence of commands $c^n$, with $n \geq 0$. We follow [Guo and Palsberg 2011] in making an abuse in program syntax by assuming that if $S_1, S_2 \in \text{Stm}$ then $S_1S_2 \in \text{Stm}$, where $S_1S_2$ denotes simple string concatenation of $S_1$ and $S_2$. We denote by $\text{State}_{GP} \triangleq \text{Store} \times \text{Stm}$ the set of states for this language. The baseline small-step operational semantics $\rightarrow_B \subseteq \text{State}_{GP} \times \text{State}_{GP}$ is standard and is given in continuation-style (where $K \in \text{Stm}$):

$$
\langle \rho, \epsilon \rangle \not\rightarrow_B \langle \rho \rangle \\
\langle \rho, \text{skip}; K \rangle \rightarrow_B \langle \rho, K \rangle \\
\langle \rho, x := E; K \rangle \rightarrow_B \langle \rho[x/E][\rho], K \rangle \\
\langle \rho, (\text{if } B \text{ then } S)K \rangle \rightarrow_B \begin{cases} 
\langle \rho, K \rangle & \text{if } \text{Booleans}[B]\rho = \text{false} \\
\langle \rho, SK \rangle & \text{if } \text{Booleans}[B]\rho = \text{true} 
\end{cases} \\
\langle \rho, (\text{while } B \text{ do } S)K \rangle \rightarrow_B \langle \rho, (\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K \rangle \\
\langle \rho, (\text{bail } B \text{ to } S)K \rangle \rightarrow_B \begin{cases} 
\langle \rho, K \rangle & \text{if } \text{Booleans}[B]\rho = \text{false} \\
\langle \rho, S \rangle & \text{if } \text{Booleans}[B]\rho = \text{true} 
\end{cases}
$$

The relation $\rightarrow_B$ is clearly deterministic and we denote by

$$
\text{Trace}_{GP} \triangleq \{ \sigma \in \text{State}^*_{GP} \mid \forall i \in [0, |\sigma|], \sigma_i \rightarrow_B \sigma_{i+1} \}
$$

the set of generic program traces for Guo and Palsberg’s language. Then, given a program $S \in \text{Stm}$, so that $\text{Store}_S \triangleq \text{vars}(S) \rightarrow \text{Value}$ denotes the set of stores for $S$, its partial trace semantics is

$$
T_{GP}[S] = \text{Trace}_{S_{GP}} \triangleq \{ \sigma \in \text{Trace}_{GP} \mid \sigma_0 = \langle \rho, S \rangle, \rho \in \text{Store}_S \}.
$$

Notice that, differently from our trace semantics, a partial trace of the program $S$ always starts from an initial state, i.e., $\langle \rho, S \rangle$.

### 10.2 Language Compilation

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

**Definition 10.1 (Language compilation).** The “first command” compilation function $C : \text{Stm} \rightarrow \wp(\mathbb{L})$ is defined as follows:

$$
C(\epsilon) \triangleq \{ I(\epsilon) : \text{skip} \rightarrow \mathbb{L} \} \\
C(S') \equiv (\text{skip}; K) \triangleq \{ I(S') : \text{skip} \rightarrow I(K) \} \\
C(S') \equiv (x := E; K) \triangleq \{ I(S') : x := E \rightarrow I(K) \} \\
C(S') \equiv ((I B \text{ then } S)K) \triangleq \{ I(S') : B \rightarrow I(SK), I(S') : \neg B \rightarrow I(K) \} \\
C(S') \equiv ((\text{while } B \text{ do } S)K) \triangleq \{ I(S') : \text{skip} \rightarrow I((I B \text{ then } (S \text{ while } B \text{ do } S))K) \} \\
C(S') \equiv ((\text{bail } B \text{ to } S)K) \triangleq \{ I(S') : B \rightarrow I(S), I(S') : \neg B \rightarrow I(K) \}
$$

Then, the full compilation function $C : \text{Stm} \rightarrow \wp(\mathbb{L})$ is recursively defined by the following clauses:

$$
C(\epsilon) \triangleq C(\epsilon) \\
C(\text{skip}; K) \triangleq C(\text{skip}; K) \cup C(K) \\
C(x := E; K) \triangleq C(x := E; K) \cup C(K) \\
C((I B \text{ then } S)K) \triangleq C((I B \text{ then } S)K) \cup C(SK) \cup C(K) \\
C((\text{while } B \text{ do } S)K) \triangleq C((\text{while } B \text{ do } S)K) \cup C((I B \text{ then } (S \text{ while } B \text{ do } S))K) \\
C((\text{bail } B \text{ to } S)K) \triangleq C((\text{bail } B \text{ to } S)K) \cup C(S) \cup C(K)
$$

Given $S \in \text{Stm}$, $I(S)$ is the initial label of $C(S)$, while $\mathbb{L}$ is, as usual, the undefined label where the execution becomes stuck.
\[
C^*((\rho, \epsilon)) \triangleq (\rho, I(\epsilon) : \text{skip} \rightarrow L)
\]

\[
C^*((\rho, S \equiv (\text{skip}; K))) \triangleq (\rho, I(S) : \text{skip} \rightarrow I(K))
\]

\[
C^*((\rho, S \equiv (x := E; K))) \triangleq (\rho, I(S) : x := E \rightarrow I(K))
\]

\[
C^*((\rho, S \equiv (\text{if } B \text{ then } S' \text{ else } S''))K)) \triangleq \begin{cases} 
(\rho, I(S) : B \rightarrow I(S')) & \text{if } B[B]\rho = \text{true} \\
(\rho, I(S) : \neg B \rightarrow I(K)) & \text{if } B[B]\rho = \text{false}
\end{cases}
\]

\[
C^*((\rho, S \equiv (\text{while } B \text{ do } S'))K)) \triangleq \begin{cases} 
(\rho, I(S) : \text{skip} \rightarrow I((\text{if } B \text{ then } S' \text{ while } B \text{ do } S'))K)) & \text{if } B[B]\rho = \text{true} \\
(\rho, I(S) : \neg B \rightarrow I(K)) & \text{if } B[B]\rho = \text{false}
\end{cases}
\]

Figure 5: Definition of the state compile function \(C^* : \text{State}_{GP} \rightarrow \text{State}\).

It turns out that the recursive function \(C\) is well defined—the easy proof is standard and is omitted. Let us just observe that \(C((\text{while } B \text{ do } S')K)\) is a base case—so that, for any \(S \in \text{Stm}\), \(C(S)\) is a finite set of commands. Let us observe that, by Definition[10.1], if \((\rho, S) \rightarrow_B (\rho', S')\) then \(C(S') \subseteq C(S)\) (this can be proved through an easy structural induction on \(S\)). Consequently, if \((\rho, S) \rightarrow_B (\rho', S')\) then \(C(S') \subseteq C(S)\).

Example 10.2. Consider the following program \(S \in \text{Stm}\) in Guo and Palsberg’s syntax:

\[
\begin{align*}
x &:= 0; \\
\text{while } B_1 \text{ do } x &:= 1; \\
x &:= 2; \\
\text{bail } B_2 \text{ to } x &:= 3; \\
x &:= 4;
\end{align*}
\]

\(S\) is then compiled in our language by \(C\) in Definition[10.1] as follows:

\[
C(S) = \{ I(S) : x := 0 \rightarrow I_{\text{while}}, I_{\text{while}} : \text{skip} \rightarrow I_{\text{ifwhile}}, I_{\text{ifwhile}} : B_1 \rightarrow I_1, I_{\text{ifwhile}} : \neg B_1 \rightarrow I_2, I_1 : x := 1 \rightarrow I_{\text{while}}, I_2 : x := 2 \rightarrow I_{\text{bail}}, I_{\text{bail}} : B_2 \rightarrow I_3, I_{\text{bail}} : \neg B_2 \rightarrow I_4, I_3 : x := 3 \rightarrow I_e, I_4 : x := 4 \rightarrow I_e, I_e : \text{skip} \rightarrow L \}.
\]

Notice that in \(I_{\text{bail}} : B_2 \rightarrow I_3\), the label \(I_3\) stands for \(I(x := 3; \ldots)\) so that \(C(x := 3; \ldots) \equiv I_3 : x := 3 \rightarrow I_e\), i.e., after the execution of \(x := 3\) the program terminates. 

Correctness for the above compilation function \(C\) means that for any \(S \in \text{Stm}\): (i) \(C(S) \in \text{Program}\) and (ii) program traces of \(S\) and \(C(S)\) have the same store sequences. In the proof we will make use of a “state compile” function \(C^* : \text{State}_{GP} \rightarrow \text{State}\) as defined in Figure[5]. In turn, \(C^*\) allows us to define a “trace compile” function \(C^f : \text{Stm}_{GP}[S] \rightarrow \text{Stm}[C(S)]\) which applies state-by-state the function \(C^*\) to traces as follows:

\[
C^f(\epsilon) \triangleq \epsilon; \quad C^f(s\tau) \triangleq C^*(s)C^f(\tau).
\]

Lemma 10.3.

\begin{enumerate}
\item \((\rho, S) \rightarrow_B (\rho', S') \iff C^*((\rho', S')) \in S(C^*(\rho, S)))\)
\item \(C^f\) is well-defined.
\end{enumerate}

Proof. We show the equivalence (1) by structural induction on \(S \in \text{Stm}\).

\(\begin{align*}
[S \equiv \epsilon]: & \text{ Trivially true, since } (\rho, S) \not\rightarrow_B \text{ and } S(\rho, I(\epsilon) : \text{skip} \rightarrow L) = \emptyset. \\
[S \equiv \text{skip}; K]: & \iff \{ (\rho, \text{skip}; K) \rightarrow_B (\rho, K), C^*((\rho, \text{skip}; K)) = (\rho, I(S) : \text{skip} \rightarrow I(K))\} \subseteq S(\rho, I(S)) \subseteq S(\rho, I(S) : \text{skip} \rightarrow I(K)). \\
[S \equiv (x := E; K)]: & \iff \{ (\rho, x := E; K) \rightarrow_B (\rho[x/E][\rho], K), C^*((\rho, x := E; K)) = (\rho, I(S) : x := E \rightarrow I(K))\} \subseteq S(\rho, I(S) : x := E \rightarrow I(K)) \subseteq S(\rho, I(S)). \\
[S \equiv \text{if } B \text{ then } S' \text{ else } S'']: & \iff \{ (\rho, \text{if } B \text{ then } S' \text{ else } S') \rightarrow_B (\rho, K), C^*((\rho, \text{if } B \text{ then } S' \text{ else } S')) = (\rho, I(S') : \text{if } B \text{ then } S' \text{ else } S')\} \subseteq S(\rho, I(S')). \\
[S \equiv \text{while } B \text{ do } S'] : & \iff \{ (\rho, \text{while } B \text{ do } S') \rightarrow_B (\rho, K), C^*((\rho, \text{while } B \text{ do } S')) = (\rho, I(S': B) : B \rightarrow I(S'))\} \subseteq S(\rho, I(S')). \\
\end{align*}\]
if well-defined function.

Thus, C

and therefore C

and therefore S' = K. Hence, (ρ, x := E; K) →B (ρ[x/E][E][ρ], K) = (ρ', S').

[7] if (B then T)K] (⇒): Assume that B[B]ρ = false, so that (ρ, (if B then T)K) →B (ρ, K), C

(⟨ρ', S'⟩) = (ρ, l] ∈ S(ρ, l] : −B → I(K)) and C

(⟨ρ, K⟩) = (ρ, I(T) : A → I(T')) for some A and T' ∈ Stm. Hence, by definition of S, (ρ, (I(T) : A → I(T')) ∈ S(ρ, l], S) : −B → I(K)). On the other hand, if B[B]ρ = true then (ρ, (if B then T)K) →B (ρ, T), C

(⟨ρ, (if B then T)K⟩) = (ρ, I(T) : B → I(T)) and C

(⟨ρ, T⟩) = (ρ, T(K) : A → I(T')) for some A and T'. Hence, (ρ, T(K) : A → I(T')) ∈ S(ρ, l], S) : B → I(K)).

(⇐): Assume that B[B]ρ = false, so that C

(⟨ρ, (if B then T)K⟩) = (ρ, I(B : −B → I(K)), and (⟨ρ', C⟩) = C

(⟨ρ', S'⟩) ∈ S(ρ, l], S) : −B → I(K)). Hence: (1) ρ'' = ρ and therefore ρ' = ρ; (2) lbl(C) = I(K), and therefore S' = K. Hence, (ρ, (if B then T)K) : −B → I(K)).

[7] (while B do T)K] (⇒): Here, (ρ, (while B do T)K) →B (ρ, (if B then (T while B do T)K) and C

(⟨ρ, (while B do T)K⟩) = (ρ, I(S) : skip → I((if B then (T while B do T)K)). If B[B]ρ = true then C

(⟨ρ, (if B then (T while B do T)K)⟩) = (ρ, I(S) : B → I(T)). Hence, (ρ, (if B then (T while B do T)K) : −B → I(K)).

Let us now turn to point (2). By the ⇒ implication of the equivalence (1), we have that if τ ∈ T

G[S] then C(τ) ∈ Τ[C(S)]. This can be shown by an easy induction on the length of τ and by using the fact that if C(τ) = ⟨C0, C1⟩ · · · ⟨Ci, Ci+1⟩ then, for any i, Ci ∈ C(S). Moreover, since lI(S) is the initial label of the compiled program C(S) and lbl(C0) = I(S), we also notice that C(τ) ∈ Τ[C(S)]. Therefore, C(τ) is a well-defined function.

Let st : Trace

G ∪Trace → Store* be the function that returns the store sequence of any trace, that is:

st(ε) = ε and st((ρ, S)[σ]) = ρ . st(σ),

and, given a set X of traces, let αst(X) = {st(σ) | σ ∈ X}. Then, correctness of the compilation function C goes as follows:

Theorem 10.4 (Correctness of language compilation). If S ∈ Stm then C(S) ∈ Program and αst(T

G[S]) = αst(T[C(S)]).

Proof. We define a "trace de-compile" function D : T[C(S)] → T

G[S] as follows. Consider a trace σ = ⟨p0, C0⟩ · · · ⟨pi, Ci⟩ ∈ T[C(S)], so that lbl(C0) = I(S), for any i ∈ [0, n], Ci ∈ C(S) and for any i ∈ [0, n], ⟨πi+1, Ci+1⟩ ∈ S[C(S)]. Since lbl(C0) = I(S), by definition of C, we have that ⟨p0, C0⟩ =
Then, since \( \langle \rho_1, C_1 \rangle \in S[C(S)] \), there exists \( S_1 \in \text{Stm} \) such that \( \text{lbl}(C_1) = I(S_1) \), so that \( \langle \rho_1, C_1 \rangle = C^*(\langle \rho_1, S_1 \rangle) \). Hence, from \( C^*(\langle \rho_1, S_1 \rangle) \in S[C(S)] \), by the implication \( \equiv \) of Lemma 10.3 (1), we obtain that \( \langle \rho_0, S \rangle \rightarrow_B \langle \rho_1, S_1 \rangle \). Thus, an easy induction allows us to show that for any \( i \in [1, n] \) there exists \( S_i \in \text{Stm} \) such that

\[
\langle \rho_0, S \rangle \rightarrow_B \langle \rho_1, S_1 \rangle \rightarrow_B \cdots \rightarrow_B \langle \rho_n, S_n \rangle
\]

and \( C^*(\langle \rho_0, S \rangle) = \langle \rho_1, C_1 \rangle \). We therefore define \( D^i(\sigma) \triangleq \langle \rho_0, S \rangle \langle \rho_1, S_1 \rangle \cdots \langle \rho_n, S_n \rangle \in T_{GP}[S] \). Moreover, we notice that \( st(D^i(\sigma)) = st(\sigma) \). Let us also observe that \( st \circ C^i = st \), since \( C^i \) does not affect stores.

Summing up, we obtain:

\[
\begin{align*}
\alpha_{st}(T_{GP}[S]) & = \{ \text{since } st \circ C^i = st \} \\
\alpha_{st}(C^*(T_{GP}[S])) & \subseteq \{ \text{by Lemma 10.3 (2), } C^i \text{ is well-defined} \} \\
\alpha_{st}(T'[C(S)]) & = \{ \text{since } st \circ C^i = st \} \\
\alpha_{st}(D^i(T'[C(S)])) & \subseteq \{ \text{since } D^i \text{ is well-defined} \}
\end{align*}
\]

and this closes the proof. \( \square \)

### 10.3 Bisimulation

Correctness of trace extraction in \cite{Guo and Palsberg, 2011} relies on a notion of bisimulation relation, parameterized by program stores. Let us recall this definition. If \( \langle \rho, S \rangle \rightarrow_B \langle \rho', S' \rangle \) then this “silent” transition that does not change the store is also denoted by \( \langle \rho, S \rangle \rightarrow_B \langle \rho', S' \rangle \). Moreover, for the assignment transition \( \langle \rho, x := E; K \rangle \rightarrow_B \langle \rho[x/E][K], K \rangle \), if \( \delta = [x/E][\rho] \) denotes the corresponding store update of \( \rho \) then this transition is also denoted by \( \langle \rho, x := E; K \rangle \rightarrow_B \langle \rho[x/E][\rho], K \rangle \). Let \( \text{Act} \triangleq \{ \delta \mid \delta \text{ is a store update} \} \cup \{ \tau \} \).

Then, for a nonempty sequence of actions \( s = a_1 \cdots a_n \in \text{Act}^+ \), we define:

\[
\langle \rho, S \rangle \overset{\tau}{\rightarrow}_B \langle \rho', S' \rangle \text{ iff } \langle \rho, S \rangle \overset{\tau}{\rightarrow}_B \circ a_1 \circ \overset{\tau}{\rightarrow}_B \cdots \overset{\tau}{\rightarrow}_B \circ a_n \circ \overset{\tau}{\rightarrow}_B \langle \rho', S' \rangle,
\]

nameley, there may be any number of silent transitions either in front of or following any \( a_i \)-transition \( a_i \). Moreover, if \( s \in \text{Act}^+ \) is a nonempty sequence of actions then \( s \in \text{Act}^+ \) denotes the possibly empty sequence of actions where all the occurrences of \( \tau \) are removed.

**Definition 10.5 \cite{Guo and Palsberg, 2011}**. 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 \( \langle \rho, S_1 \rangle \overset{\tau}{\rightarrow}_B \langle \rho', S'_1 \rangle \) then \( \langle \rho, S_2 \rangle \overset{\tau}{\rightarrow}_B \langle \rho', S'_2 \rangle \), for some \( \langle \rho', S'_2 \rangle \) such that \( R(\rho', S'_1, S'_2) \);
2. if \( \langle \rho, S_2 \rangle \overset{\tau}{\rightarrow}_B \langle \rho', S'_2 \rangle \) then \( \langle \rho, S_1 \rangle \overset{\tau}{\rightarrow}_B \langle \rho', S'_1 \rangle \), for some \( \langle \rho', S'_1 \rangle \) such that \( R(\rho', S'_1, S'_2) \).

\( S_1 \) is bisimilar to \( S_2 \) for a given \( \rho \in \text{Store} \), denoted by \( S_1 \equiv_{\rho} S_2 \), if \( R(\rho, S_1, S_2) \) for some bisimulation \( R \). \( \square \)

Let us observe that if \( \langle \rho, S_1 \rangle \overset{\tau}{\rightarrow}_B \langle \rho', S'_1 \rangle \) then \( \hat{\tau} = \epsilon \), so that \( \{ \langle \rho, S_2 \rangle \overset{\tau}{\rightarrow}_B \langle \rho, S_2 \rangle \} \equiv \langle \rho, S_2 \rangle \) is allowed to be the matching (empty) transition sequence.

It turns out that bisimilarity can be characterized through an abstraction map of traces that observes store changes. By a negligible abuse of notation, we consider the store changes function \( sc : \text{Trace} \rightarrow \text{Store}^* \) in Section 6 also applicable to GP traces, so that \( sc : \text{Trace} \cup \text{Trace}_{GP} \rightarrow \text{Store}^* \). In turn, given \( \rho \in \text{Store} \), the function \( \alpha_{sc}^\rho : \varphi(\text{Trace}_{GP}) \rightarrow \varphi(\text{Store}^*) \) is then defined as follows:

\[
\alpha_{sc}^\rho(X) \triangleq \{ \varphi(\tau) \in \text{Store}^* \mid \tau \in X, \exists S, \tau'. \tau = (\rho, S)\tau' \}.
\]

It is worth remarking that \( \alpha_{sc}^\rho \) is a weaker abstraction than \( \alpha_{sc} \) defined in Section 6 that is, for any \( X, Y \in \varphi(\text{Trace}_{GP}), \alpha_{sc}(X) = \alpha_{sc}(Y) \Rightarrow \alpha_{sc}^\rho(X) = \alpha_{sc}^\rho(Y) \).

**Theorem 10.6**. For any \( S_1, S_2 \in \text{Stm}, \rho \in \text{Store} \), we have that \( S_1 \equiv_{\rho} S_2 \) iff \( \alpha_{sc}^\rho(T_{GP}[S_1]) = \alpha_{sc}^\rho(T_{GP}[S_2]) \).

**Proof**. (\( \Rightarrow \)): Let us prove that if \( R(\rho, S_1, S_2) \) holds for some bisimulation \( R \) then it turns out that \( \alpha_{sc}^\rho(T_{GP}[S_1]) \subseteq \alpha_{sc}^\rho(T_{GP}[S_2]) \) (the reverse containment is symmetric), that is, if \( sc(\tau) \in \text{Store}^* \) for \( \tau \in T_{GP}[S_1] \) such that \( \tau = (\rho, S_1)\tau' \) then there exists some \( \psi \in T_{GP}[S_2] \) such that \( \psi = (\rho, S_2)\psi' \) and \( sc(\tau) = sc(\psi) \). Let us
then consider \( \tau \in T_{GP}[S_1] \) such that \( \tau = \langle \rho, S_1 \rangle \tau' \). If \( \tau' = \epsilon \) then we choose \( \langle \rho, S_2 \rangle \in T_{GP}[S_2] \) so that \( sc(\langle \rho, S_1 \rangle) = sc(\langle \rho, S_2 \rangle) \). Otherwise, \( \tau = \langle \rho, S_1 \rangle \tau' \in T_{GP}[S_1] \), with \( \epsilon \neq \tau' = \tau''(\mu, S) \). We prove by induction on \( |\tau'| \geq 1 \) that there exists \( \psi = \langle \rho, S_2 \rangle \psi''(\mu, T) \in T_{GP}[S_2] \) such that \( sc(\tau) = sc(\psi) \) and \( R(\mu, S, T) \).

\((|\tau'| = 1)\): In this case, \( \tau = \langle \rho, S_1 \rangle(\mu, S) \in T_{GP}[S_1] \), so that \( \langle \rho, S_1 \rangle \xrightarrow{B} (\mu, S) \). Since, by hypothesis, \( R(\rho, S_1, S_2) \) holds, we have that \( \langle \rho, S_2 \rangle \xrightarrow{B} (\mu, T) \), for some \( T \), and \( R(\mu, S, T) \). Let \( \psi \in T_{GP}[S_2] \) be the trace corresponding to the sequence of transitions \( \langle \rho, S_2 \rangle \xrightarrow{B} (\mu, T) \). Then, by definition of \( \xrightarrow{B} \), we have that \( sc(\tau) \in sc(\psi) \), and, by definition of bisimulation, \( R(\mu, S, T) \) holds.

\((|\tau'| > 1)\): Here, \( \tau' = \tau''(\mu, S) \) and \( \tau = \langle \rho, S_1 \rangle \tau' \in T_{GP}[S_1] \), with \( |\tau'| = |\tau'| - 1 \geq 1 \). Hence, \( \tau'' = \tau''(\eta, U) \).

By inductive hypothesis there exists \( \psi = \langle \rho, S_2 \rangle \psi''(\eta, V) \in T_{GP}[S_2] \) such that \( sc(\langle \rho, S_1 \rangle \tau''(\eta, U)) = sc(\langle \rho, S_2 \rangle \psi''(\eta, V)) \) and \( R(\eta, U, V) \). Since \( (\eta, U) \xrightarrow{B} (\mu, S) \) and \( R(\eta, U, V) \) holds, we obtain that \( (\eta, V) \xrightarrow{B} (\mu, T) \), for some \( T \), and \( R(\mu, S, T) \) holds. Let \( (\eta, V) \cdots (\mu, T) \) be the sequence of states corresponding to the sequence of transitions \( (\eta, V) \xrightarrow{B} (\mu, T) \) so that we pick \( \langle \rho, S_2 \rangle \psi''(\eta, V) \cdots (\mu, T) \in T_{GP}[S_2] \). The condition \( R(\mu, S, T) \) already holds. Moreover, by definition of \( \xrightarrow{B} \) we have that \( sc((\eta, U)\langle \mu, S \rangle) = sc((\eta, V) \cdots (\mu, T)) \), and therefore we obtain \( sc(\tau) = sc((\rho, S_1)\tau''(\eta, U)\langle \mu, S \rangle) = sc((\rho, S_2)\psi''(\eta, V) \cdots (\mu, T)) \).

Given \( \rho \in \text{Store} \), we assume that \( \alpha_{sc}(T_{GP}[S_1]) = \alpha_{sc}(T_{GP}[S_2]) \) and we then define the following relation \( R \):

\[ R \triangleq \{(\rho, S_1, S_2)\} \cup \{(\mu, T_1, T_2) \mid (\rho, S_1) \cdots (\mu, T_1) \in T[S_1], (\rho, S_2) \cdots (\mu, T_1) \in T[S_2], sc((\rho, S_1) \cdots (\mu, T_1)) = sc((\rho, S_2) \cdots (\mu, T_1))\}. \]

We show that \( R \) is a bisimulation, so that \( R(\rho, S_1, S_2) \) follows.

(case A) Assume that \( (\rho, S_1) \xrightarrow{B} (\rho', S_1') \). Since \( (\rho, S_1)\langle \rho', S_1' \rangle \in T[S_1] \) and \( \alpha_{sc}(T_{GP}[S_1]) = \alpha_{sc}(T_{GP}[S_2]) \), we have that there exists \( \tau = (\rho, S_2) \cdots \in T[S_2] \) such that \( sc((\rho, S_1)\langle \rho', S_1' \rangle) = sc(\tau) \). Hence, \( \tau \) necessarily has the following shape:

\[ \tau = (\rho, S_2)(\rho, U_1) \cdots (\rho, U_n) \langle \rho', V_1 \rangle \cdots (\rho', V_m) \]

where \( n \geq 0 \) and \( n = 0 \) means that \( (\rho, U_1) \) \cdots (\rho, U_n) \) is the empty sequence and \( m \geq 1 \). This therefore means that \( (\rho, S_2) \xrightarrow{B} (\rho', V_m) \), so that, by definition of \( R \), \( R(\rho', S_1, V_m) \) holds.

(case B) Assume now that \( R(\mu, T_1, T_2) \) holds because \( \delta = (\rho, S_1) \cdots (\mu, T_1) \in T[S_1], \sigma = (\rho, S_2) \cdots (\mu, T_2) \in T[S_2] \) and \( sc(\delta) = sc(\sigma) \). Hence, let us suppose that \( (\mu, T_1) \xrightarrow{B} (\mu', T_1') \). Then, since \( \delta(\mu', T_1') \in T[S_1] \) and \( \alpha_{sc}(T_{GP}[S_1]) = \alpha_{sc}(T_{GP}[S_2]) \), we have that there exists \( \tau = (\rho, S_2) \cdots \in T[S_2] \) such that \( sc(\delta(\mu', T_1')) = sc(\tau) \).

(case B1) If \( |\tau| \leq |\sigma| \) then, by the property \((*) \) above, \( \sigma = \tau \psi, \) for some \( \psi \). Hence, \( sc(\tau) = sc(\delta(\mu', T_1')) \) is a prefix of \( sc(\sigma) \). Consequently, \( sc(\delta(\mu', T_1')) \) can be a prefix of \( sc(\sigma) \) only if \( sc(\delta(\mu', T_1')) = sc(\delta) \), so that the action \( a = \tau \) and \( \mu' = \mu \), that is, \( (\mu, T_1) \xrightarrow{B} (\mu', T_1') \). We thus consider the empty transition sequence \( (\mu, T_2) \xrightarrow{B} (\mu, T_2) \), so that \( sc(\delta(\mu', T_1')) = sc(\sigma) \), by definition of \( R \) we obtain that \( R(\mu, T_1, T_2) \) holds.

(case B2) If \( |\tau| > |\sigma| \) then, by \((*) \) above, we have that \( \tau = \sigma \psi, \) for some \( \psi, \) i.e., \( \tau = \sigma \cdots \langle \mu'', T_2' \rangle \), for some \( \mu'' \) and \( T_2' \). Since \( sc((\rho, S_2) \cdots (\mu, T_2)) = sc((\rho, S_1) \cdots (\mu, T_1)) \) and \( sc((\rho, S_2) \cdots (\mu, T_2) \cdots (\mu'', T_2')) = sc((\rho, S_1) \cdots (\mu, T_1) \langle \mu', T_1' \rangle) \), we derive that \( \mu'' = \mu' \) and \( (\mu, T_2) \xrightarrow{B} (\mu'', T_2') \). By definition of \( R \), \( R(\mu', T_1', T_2') \) holds.

This closes the proof.

\[ \square \]

### 10.4 Hot Paths

Let us recall the set of rules that define the tracing transitions in the model by [Guo and Palsberg 2011](#). Let \( tState_{GP} \triangleq \text{Store} \times \text{Stm} \times \text{Stm} \times \text{Stm} \) denote the set of states in trace recording mode, whose components are, respectively, the current store, the entry point of the recorded trace (this is always a while statement), the current trace (i.e., a sequence of commands) and the current program to be evaluated. In turn, \( State_{GP} \triangleq State_{GP} \cup tState_{GP} \) denotes the corresponding extended notion of state, which encompasses the trace recording mode. Then, the relation \( \rightarrow_T \subseteq State_{GP} \times State_{GP} \) is defined by the clauses in Figure [2](#) where \( O : \text{Stm} \times \text{Store} \rightarrow \text{Stm} \) is a “sound” optimization function that depends on a given store. Correspondingly, the
be modeled in our framework by exploiting a revised loop selection map follows:

\[(T_3) \quad \langle \rho, (\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K \rangle \rightarrow_T \langle \rho, (\text{while } B \text{ do } S)K, \epsilon, S(\text{while } B \text{ do } S)K \rangle \quad \text{if } B[B]_{\rho} = \text{true} \]

\[(T_4) \quad \langle \rho, K, w, t, \text{skip}, K \rangle \rightarrow_T \langle \rho, K, w, t(\text{skip}), K \rangle \]

\[(T_5) \quad \langle \rho, K, w, x := E; K \rangle \rightarrow_T \langle \rho[x/E][E]_{\rho}, K, w, t(x := E), K \rangle \]

\[(T_6) \quad \langle \rho, K, w, t, (\text{while } B \text{ do } S)K \rangle \rightarrow_T \left\{ \begin{array}{ll}
\langle \rho, K, w, t(\text{bail } B \text{ to } (SK)) \rangle & \text{if } B[B]_{\rho} = \text{false} \\
\langle \rho, K, w, t(\text{bail } \neg B \text{ to } K), SK \rangle & \text{if } B[B]_{\rho} = \text{true} \\
\langle \rho, K, w, t(\text{bail } B \text{ to } K), SK \rangle & \text{if } B[B]_{\rho} = \text{true} \\
\langle \rho, (\text{while } B \text{ do } S)K \rangle & \text{if } K_w \neq (\text{while } B \text{ do } S)K \\
\langle \rho, O(\text{while } B \text{ do } S, \rho)K \rangle & \text{if } K_w \equiv (\text{while } B \text{ do } S)K \\
\end{array} \right. \]

\[(T_7) \quad \langle \rho, K, w, t, S \rangle \rightarrow_T \langle \rho', S' \rangle \quad \text{if } K_w \neq S \quad \text{and } \langle \rho, S \rangle \rightarrow_B \langle \rho', S' \rangle \]

Figure 6: Definition of the tracing relation $\rightarrow_T$.

trace semantics $T_{GP}[S] \subseteq (\text{State}_{GP})^+$ of a program $S \in \text{Stm}$ is naturally extended to the relation $\rightarrow_{B,T} \triangleq \rightarrow_B \cup \rightarrow_T \subseteq \text{State}_{GP} \times \text{State}_{GP}$.

Let us notice that in Guo and Palsberg’s model of hot paths:

(i) By clause $(T_1)$, trace recording is always triggered by an unfolded while loop, and the loop itself is not included in the hot path.

(ii) By clause $(T_4)$, when we bail out of a hot path $t$ through a bail command, we cannot anymore re-enter into $t$.

(iii) By clause $(T_5)$—the second condition of this clause is called stitch rule in [Guo and Palsberg 2011]—the store used to optimize a hot path $t$ is recorded at the end of the first loop iteration. This is a concrete store which is used by $O$ to optimize the stitched hot path while $B \text{ do } t$.

(iv) Hot paths actually are 1-hot paths according to our definition, since, by clause $(T_1)$, once the first iteration of the traced while loop is terminated, trace recording necessarily discontinues.

(v) There are no clauses for trace recording bail commands. Hence, when trying to trace a loop that already contains a nested hot path, by clause $(T_6)$, trace recording is aborted when a bail command is encountered.

In other terms, in contrast to our approach described in Section[2], nested hot paths are not allowed.

(vi) Observe that when tracing a loop while $B \text{ do } S$ whose body $S$ does not contain branching commands, i.e. if or while statements, it turns out that the hot path $t$ coincides with the body $S$, so that while $B \text{ do } S$ $\equiv$ while $B \text{ do } S$, namely, in this case the hot path transform does not change the subject while loop.

In the following, we show how this hot path extraction model can be formalized within our trace-based approach. To this aim, we do not consider optimizations of hot paths, which is an orthogonal issue here, so that we assume that $O$ performs no optimization, that is, $O(\text{while } B \text{ do } t, \rho) = \text{while } B \text{ do } t$.

A sequence of commands $t \in \text{Stm}$ is defined to be a GP hot path for a program $Q \in \text{Stm}$ when we have the following transition sequence:

\[(\langle \rho, Q \rangle \rightarrow_{B,T} \langle \rho', (\text{while } B \text{ do } S)K \rangle \rightarrow^{*}_{B,T} \langle \rho'', (\text{while } B \text{ do } S)K, t, (\text{while } B \text{ do } S)K \rangle) \]

Since the operational semantics $\rightarrow_{B,T}$ is given in continuation-style, without loss of generality, we assume that the program $Q$ begins with a while statement, that is $Q \equiv (\text{while } B \text{ do } S)K$. Guo and Palsberg’s hot loops can be modeled in our framework by exploiting a revised loop selection map $\text{loop}_{GP} : \text{Trace} \rightarrow \wp(\mathbb{C})^+$ defined as follows:

\[\text{loop}_{GP}(\langle \rho_0, C_0 \rangle \cdots (\rho_n, C_n)) \triangleq \left\{ C_i C_{i+1} \cdots C_j | 0 \leq i \leq j < n, C_i \prec C_j, \right.\]

\[\left. \text{suc}(C_j) = \text{lbl}(C_i), \forall k \in (i, j), C_k \not\in \{ C_i, \text{cmpl}(C_i) \} \right\}.\]
Thus, loop\(_{GP}(\tau)\) contains sequences of commands without store. The map \(\alpha^{GP}_{hot}: \varphi(\text{Trace}) \rightarrow \varphi(C^+)\) then lifts loop\(_{GP}\) to sets of traces as usual: \(\alpha^{GP}_{hot}(T) \triangleq \cup_{\tau \in T}\) loop\(_{GP}(\tau)\). Then, let us consider a GP hot path \(t\) as recorded by a transition sequence \(\tau:\)
\[
\tau \triangleq (\rho, S_0 \equiv (\text{while } B \text{ do } S)K) \rightarrow_B \\
(\rho, S_1 \equiv (\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K) \rightarrow_T \\
(\rho, (\text{while } B \text{ do } S)K, \varepsilon, S_2 \equiv S(\text{while } B \text{ do } S)K) \rightarrow_T \\
\ldots \rightarrow_T \\
(\rho', (\text{while } B \text{ do } S)K, t', S_n) \rightarrow_T \\
(\rho'', (\text{while } B \text{ do } S)K, t, S_{n+1} \equiv (\text{while } B \text{ do } S)K)
\]
where \(B[B]_\rho = true\). Hence, the \(S_i\)’s occurring in \(\tau\) are the current statements to be evaluated. With a negligible abuse of notation, we assume that \(\tau \in T_{GP}\)\([\text{while } B \text{ do } S]K\)\), that is, the arrow symbols \(\rightarrow_B\) and \(\rightarrow_T\) are taken out of the sequence \(\tau\). By Lemma [10.3(2)], we therefore consider the corresponding execution trace \(C^*(\tau)\) of the compiled program \(\mathcal{C}(\text{while } B \text{ do } S)K\), where the state compile function \(C^*\) in Figure 5 when applied to states in trace recording mode, is assumed to act on the current store, and the program to be evaluated, that is, \(C^*(\rho, K_w, t, S) = C^*(\rho, S)\). We thus obtain:
\[
C^*(\tau) \triangleq 
(\rho, C_0 \equiv I((\text{while } B \text{ do } S)K) \text{ : skip} \rightarrow I((\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K)) \\
(\rho, C_1 \equiv I(\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K : B \rightarrow I(S \text{ while } B \text{ do } S)K)) \\
(\rho, C_2 \equiv I(S \text{ while } B \text{ do } S)K : A_2 \rightarrow I(T)) \\
\ldots \\
(\rho', C_n \equiv I(S_n) : A_n \rightarrow I(\text{while } B \text{ do } S)K)) \\
(\rho'', C_{n+1} \equiv I(\text{while } B \text{ do } S)K) \text{ : skip} \rightarrow I(\text{if } B \text{ then } (S \text{ while } B \text{ do } S)K))
\]
We obtain a hot path \(h_{p_t} = C_0C_1 \cdots C_n \in \text{loop}_{GP}(C^*(\tau))\), i.e. \(h_{p_t} \in \alpha^{GP}_{hot}(T^*[\mathcal{C}(\text{while } B \text{ do } S)K]))\), where \(\text{lbl}(C_0) = I(\text{while } B \text{ do } S)K) = \text{suc}(C_n)\). This is a consequence of the fact that for all \(k \in (0, n]\), \(C_k\) cannot be the entry command \(C_0\) or its complement command, because, by the stitch rule of clause (Tb), \(S_{n+1}\) is necessarily the first occurrence of \((\text{while } B \text{ do } S))K\) as current program to be evaluated in the trace \(\tau\), so that, for any \(k \in (0, n]\), \(\text{lbl}(C_k) \neq I((\text{while } B \text{ do } S)K)\). We have thus shown that any GP hot path arising from a trace \(\tau\) generates a corresponding hot path extracted by our selection map loop\(_{GP}\) on the compiled trace \(C^*(\tau)\):

**Lemma 10.7.** Let \(Q_w \equiv (\text{while } B \text{ do } S)K\). If \(t\) is a GP hot path for \(Q_w\), where \(t \equiv (\rho, Q_w) \rightarrow_B T\), \((\rho', Q_w, t, Q_w)\) is the transition sequence (\(\dagger\)) that records \(t\), then there exists a hot path \(h_{p_t} = C_0C_1 \cdots C_n \in \alpha^{GP}_{hot}(T^*[\mathcal{C}(Q_w)])\) such that, for any \(i \in [0, n]\), \(\text{lbl}(C_i) = I(S_i)\), and, in particular, \(\text{lbl}(C_0) = I(Q_w) = \text{suc}(C_n)\).

**Example 10.8.** Let us consider the while statement \(Q_w\) of the program in Example 2.1.
\[
Q_w \equiv \text{while } (x \leq 20) \text{ do } (x := x + 1; (\text{if } (x^3 \equiv 0) \text{ then } x := x + 3;))
\]
This program is already written in Guo and Palsberg language, so that \(Q_w\) is a well formed statement in Stm. The tracing rules in Figure 6 yield the following trace \(t\) for \(Q_w\):
\[
t \equiv x := x + 1; \text{ bail } (x^3 \equiv 0) \text{ to } (x := x + 3; Q_w).
\]
On the other hand, the compiled program \(\mathcal{C}(Q_w) \in \varphi(C)\) is as follows:
\[
\mathcal{C}(Q_w) = \{D_0 \equiv \text{lwhile : skip} \rightarrow \text{lwhile}; \}
\]
\[
D_1 \equiv \text{lwhile : } (x \leq 20) \rightarrow l_1, D_1^0 \equiv \text{lwhile : } \neg (x \leq 20) \rightarrow l_e, \\
D_2 \equiv l_1 : x := x + 1 \rightarrow l_0, \\
D_3 \equiv l_0 : (x^3 \equiv 0) \rightarrow l_2, D_3^0 \equiv l_0 : \neg (x^3 \equiv 0) \rightarrow \text{lwhile}, \\
D_4 \equiv l_2 : x := x + 3 \rightarrow \text{lwhile}, D_5 \equiv l_e : \text{skip} \rightarrow L_1, \}
\]
where labels have the following meaning:
\[
l_{\text{while}} = I(Q_w) \]
\[
l_{\text{lwhile}} = I((x \leq 20) \text{ then } (x := x + 1; (\text{if } (x^3 \equiv 0) \text{ then } x := x + 3;)) Q_w))
\]
\[
l_1 = I((x := x + 1; (\text{if } (x^3 \equiv 0) \text{ then } x := x + 3;)) Q_w))
\]
\[
l_2 = I((\text{if } (x^3 \equiv 0) \text{ then } x := x + 3;)) Q_w)\]
\[
l_3 = I(x := x + 3; Q_w).
\]
Hence, in correspondence with the trace \(t\), we obtain the hot path \(h_{p_t} = D_0D_1D_2D_3^0 \in \alpha^{GP}_{hot}(T^*[\mathcal{C}(Q_w)])\). In turn, this hot path \(h_{p_t}\) corresponds to the 2-hot path \(h_{p_1}\) with the analogous sequence of commands, which has been selected in Example 4.7. \qed
10.5 GP Trace Extraction

In the following, we conform to the notation used in Section 5 for our trace extraction transform. Let us consider a while program \( Q_w \equiv (\text{while } B \text{ do } S)K \in \text{Stm} \) and its compilation \( P_w \equiv C(Q_w) \in \nu(C) \). Observe that, by Definition 10.8 of compilation \( C \), a hot path \( C_0 \cdots C_n \in \alpha_{\text{hot}}(T[P_w]) \) for the compiled program \( P_w \) always arises in correspondence with some while loop \( \text{while } B' \text{ do } S' \) occurring in \( Q_w \) and therefore has necessarily the following shape:

\[
\begin{align*}
C_0 &\equiv l((\text{while } B' \text{ do } S') \cdot J) : \text{skip} \rightarrow l((\text{if } B' \text{ then } (S' \text{ while } B' \text{ do } S') \cdot J)) \\
C_1 &\equiv l((\text{if } B' \text{ then } (S' \text{ while } B' \text{ do } S')) \cdot J) : B' \rightarrow l(S' \text{ while } B' \text{ do } S') \cdot J) \\
C_2 &\equiv l(S' \text{ while } B' \text{ do } S') \cdot J) : A_2 \rightarrow l(T_3) \\
\cdots \\
C_n &\equiv l(T_n) : A_n \rightarrow l((\text{while } B' \text{ do } S') \cdot J)
\end{align*}
\]

The GP hot path extraction scheme for \( Q_w \) described by the rules in Figure 6 can be defined in our language by the following simple transform of \( P_w \).

**Definition 10.9 (GP trace extraction transform).** The GP trace extraction transform of \( P_w \) for the hot path \( hp = C_0 C_1 \cdots C_n \in \alpha_{\text{hot}}(T[P_w]) \) is as follows:

1. If for \( i \in [2, n] \), \( \text{cmpl}(C_i) \not\in P_w \) then \( \text{extr}_{hp}(P_w) \equiv P_w \);

2. Otherwise:

\[
\text{extr}_{hp}(P_w) \equiv P_w \cup \{\ell_i : \text{act}(C_i) \rightarrow \ell_{\text{next}(i)} \mid i \in [0, n]\} \cup \{\ell_i : \neg \text{act}(C_i) \rightarrow \ell_{\text{next}(i)} \mid i \in [0, n], \text{cmpl}(C_i) \in P_w\}. \quad \square
\]

Clearly, \( \text{extr}_{hp}(P) \) remains a well-formed program. Also observe that the case (1) of Definition 10.9 means that the traced hot path \( hp \) does not contain conditional commands (except from the entry conditional \( C_1 \)) and therefore corresponds to point (vi) in Section 5.6.4.

**Example 10.10.** Let us consider the programs \( Q_w \) and \( C(Q_w) \) of Example 10.8 and the hot path \( hp_t = D_0 D_1 D_2 D_3 \in \alpha_{\text{hot}}(T[C(Q_w)]) \) which corresponds to the trace \( t \equiv x := x + 1; \text{bail } (x \% 3 = 0) \to (x := x + 3; Q_w) \) of \( Q_w \).

Here, the GP trace extraction of \( hp_t \), according to Definition 10.9 provides the following program transform:

\[
\text{extr}_{hp_t}(C(Q_w)) \equiv \{D_0 \equiv \text{lwhile } : \text{skip} \rightarrow l_{\text{ifwhile}}, D_1 \equiv l_{\text{ifwhile}} : (x \leq 20) \rightarrow l_1, \\
D_2 \equiv l_{\text{ifwhile}} : \neg(x \leq 20) \rightarrow l_1, D_2 \equiv l_1 : x := x + 1 \rightarrow l_{\text{if}}, D_3 \equiv l_1 : (x \% 3 = 0) \rightarrow l_2, D_3 \equiv l_2 : \neg(x \% 3 = 0) \rightarrow l_{\text{ifwhile}}, \\
D_4 \equiv l_2 : x := x + 3 \rightarrow l_{\text{ifwhile}}, D_5 \equiv l_1 : \text{skip} \rightarrow l, \\
\{\ell_0 : \text{skip} \rightarrow \ell_1, \ell_1 : x \leq 20 \rightarrow \ell_2, \ell_1 : \neg(x \leq 20) \rightarrow l, \\
\ell_2 : x := x + 1 \rightarrow \ell_3, \ell_3 : \neg(x \% 3 = 0) \rightarrow l_0, \ell_3 : (x \% 3 = 0) \rightarrow l_2\}\}
\]

On the other hand, the stitch rule \( (T_5) \) transforms \( Q_w \) into the following program \( Q_t \):

\[
\text{while } (x \leq 20) \to (x := x + 1; \text{bail } (x \% 3 = 0) \to (x := x + 3; Q_w))
\]

whose compilation yields the following program:

\[
C(Q_t) = \{\text{lwhile } : \text{skip} \rightarrow l_{\text{ifwhile}}, l_{\text{ifwhile}} : (x \leq 20) \rightarrow l_{\text{if}}, l_{\text{if}} : \neg(x \leq 20) \rightarrow l, \\
l_{\text{if}} : x := x + 1 \rightarrow l_{\text{bail}}, l_{\text{bail}} : (x \% 3 = 0) \rightarrow l_{\text{bail}}, l_{\text{bail}} : \neg(x \% 3 = 0) \rightarrow l_{\text{ifwhile}}, \\
l_{\text{ifwhile}} : \text{skip} \rightarrow l_{\text{ifwhile}}, l_{\text{ifwhile}} : (x \leq 20) \rightarrow l_1, l_{\text{ifwhile}} : \neg(x \leq 20) \rightarrow l, \\
l_1 : x := x + 1 \rightarrow l_{\text{if}}, l_{\text{if}} : (x \% 3 = 0) \rightarrow l_2, l_{\text{if}} : \neg(x \% 3 = 0) \rightarrow l_{\text{ifwhile}}, \\
l_{\text{bail}} : x := x + 3 \rightarrow l_{\text{while}}, l : \text{skip} \rightarrow l\}
\]
Theorem 10.11 follows.

with the following new labels:

\[ l_{\text{while}} = l(\text{while } (x \leq 20) t) \]
\[ l'_{\text{if}} = l(\text{if } (x \leq 20) \text{ then } (t \text{ (while } (x \leq 20) t))) \]
\[ l_1 = l(t \text{ (while } (x \leq 20) t)) \]
\[ l_{\text{bail}} = l((\text{bail } (x\%3 = 0) \text{ to } (x := x + 3; Q_w))\text{(while } (x \leq 20) t)) \]

while observe that \( l_{\text{bail}} = l(x := x + 3; Q_w) &= l_2 \). It is then immediate to check that the programs \( C(Q_t) \) and \( \text{extr}_{\text{gp}}(C(Q_w)) \) are equal up to the following label renaming of \( \text{extr}_{\text{gp}}(C(Q_w)) \):

\[ \{ l_0 \mapsto l_{\text{while}}, l_1 \mapsto l'_{\text{if}}, l_2 \mapsto l_1, l_3 \mapsto l_{\text{bail}} \} \]

The equivalence of this GP trace extraction with the stitch of hot paths by [Guo and Palsberg 2011] goes as follows.

**Theorem 10.11 (Equivalence with GP trace extraction).** Let \( t \) be a GP trace such that \( \langle \rho, (\text{while } B \text{ do } S)K \rangle \to_B t \) \( \langle \rho', (\text{while } B \text{ do } S)K, t, (\text{while } B \text{ do } S)K \rangle \) and let \( h_{p_t} \in \text{extr}_{\text{gp}}(C((\text{while } B \text{ do } S)K)) \) be the corresponding GP hot path as determined by Lemma 10.7. Then, \( C((\text{while } B \text{ do } t)K) \cong \text{extr}_{\text{gp}}(C((\text{while } B \text{ do } S)K)) \).

**Proof.** Let the GP hot path \( t \) be recorded by the following transition sequence for \( \langle \rho, (\text{while } B \text{ do } S)K \rangle \):

\[ \langle \rho, S_2 \equiv (\text{while } B \text{ do } S)K \rangle \to_B \]
\[ \langle \rho, S_1 \equiv (\text{if } \text{then } \text{while } B \text{ do } S, K) \rangle \]
\[ \langle \rho_1 \equiv \rho, \text{while } B \text{ do } S, K, t_0 \equiv \epsilon, S_0 \equiv \text{while } B \text{ do } S, K \rangle \]
\[ \cdots \]
\[ \langle \rho_n, \text{while } B \text{ do } S, K, t_n \equiv t_{n-1}c_n, S_n \rangle \]
\[ \langle \rho_n+1 \equiv \rho', \text{while } B \text{ do } S, K, t \equiv t_{n}c_{n+1}, S_n+1 \equiv \text{while } B \text{ do } S, K \]

where \( n \geq 0 \), so that the body \( S \) is assumed to be nonempty, i.e., \( S \neq \epsilon \) (there is no loss of generality since for \( S = \epsilon \) the result trivially holds). Hence, \( t = c_1...c_n \), for some commands \( c_i \text{Cmd} \), and the corresponding hot path \( h_{p_t} \equiv H_0H_1...H_n \) as determined by Lemma 10.7 is as follows:

\[ H_{-2} \triangleq l(S_{-2}) : \text{skip} \to l(S_{-1}) \]
\[ H_{-1} \triangleq l(S_{-1}) : B \to l(S_0) \]
\[ H_0 \triangleq l(S_0) : A_0 \to l(S_1) \]
\[ H_1 \triangleq l(S_1) : A_1 \to l(S_2) \]
\[ \cdots \]
\[ H_n \triangleq l(S_n) : A_n \to l(S_{-2}) \]

where the action \( A_i \), with \( i \in [0, n] \), and the command \( c_{i+1} \) depend on the first command of the statement \( S_i \) as follows (this range of cases will be later referred to as (\(*\)):

\begin{enumerate}
  \item \( S_i \equiv \text{skip; J} \quad \Rightarrow \quad A_i \equiv \text{skip} \quad & \quad c_{i+1} \equiv \text{skip} \quad & \quad S_{i+1} \equiv J \)
  \item \( S_i \equiv x := E; J \quad \Rightarrow \quad A_i \equiv x := E \quad & \quad c_{i+1} \equiv x := E \quad & \quad S_{i+1} \equiv J \)
  \item \( S_i \equiv (\text{if } B' \text{ then } S')J \quad & \quad B'[\rho_i = \text{true}] \quad \Rightarrow \quad A_i \equiv B' \quad & \quad c_{i+1} \equiv \text{bail } \neg B' \text{ to } J \quad & \quad S_{i+1} \equiv S'J \)
  \item \( S_i \equiv (\text{if } B' \text{ then } S')J \quad & \quad B'[\rho_i = \text{false}] \quad \Rightarrow \quad A_i \equiv \neg B' \quad & \quad c_{i+1} \equiv \text{bail } \text{to } (S')J \quad & \quad S_{i+1} \equiv J \)
  \item \( S_i \equiv (\text{while } B' \text{ do } S')J \quad & \quad (\text{while } B' \text{ do } S')J \neq (\text{while } B \text{ do } S)K \quad \Rightarrow \quad A_i \equiv \text{skip} \quad & \quad c_{i+1} \equiv \text{skip} \quad & \quad S_{i+1} \equiv (\text{if } B' \text{ then } (S'(\text{while } B' \text{ do } S')))J \)
\end{enumerate}

If, for any \( i \in [0, n] \), \( H_i \) is not a conditional command then, by case (1) of Definition 10.9 we have that \( \text{extr}_{\text{gp}}(C((\text{while } B \text{ do } S)K)) = C((\text{while } B \text{ do } S)K) \). Also, for any \( i \in [0, n] \), \( A_i \) is either a skip or an assignment, so that \( c_{i+1} = A_i \), and, in turn, \( t = S \). Hence, \( (\text{while } B \text{ do } t)K \equiv (\text{while } B \text{ do } S)K \), so that the thesis follows trivially.
Thus, we assume that $H_k$, with $k \in [0,n]$ is the first conditional command occurring in the sequence $H_0...H_n$. Case (2) of Definition\cite{10.5555/4260303.4260698} applies, so that:

$$\text{extr}_{h_{bp}}^{GP}(C((\text{while } B \text{ do } S)K)) = C((\text{while } B \text{ do } S)K) \cup \left\{ \ell_{-2} : \text{skip} \rightarrow \ell_{-1}, \ell_{-1} : B \rightarrow \ell_0, \ell_{-1} : \neg B \rightarrow I(K), \ell_0 : A_0 \rightarrow \ell_1, ..., \ell_n : A_n \rightarrow \ell_{-2} \right\} \cup \left\{ \ell_i : \neg A_i \rightarrow I(S_{\text{next}(i)}) \mid i \in [0,n], A_i \in B\text{Exp} \right\} $$

Moreover, we have that:

$$C((\text{while } B \text{ do } S)K) = \left\{ I((\text{while } B \text{ do } S)K) : \text{skip} \rightarrow I((\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K), \right. $$

$$I((\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K) : B \rightarrow I(S(\text{while } B \text{ do } S)K), $$

$$I((\text{if } B \text{ then } (S \text{ while } B \text{ do } S))K) : \neg B \rightarrow I(K) \} $$

$$\cup C(S(\text{while } B \text{ do } S)K) \cup C(K) $$

$$C((\text{if } B \text{ then } (t \text{ while } B \text{ do } t)K) = \left\{ I((\text{if } B \text{ then } (t \text{ while } B \text{ do } t)K) : \text{skip} \rightarrow I((\text{if } B \text{ then } (t \text{ while } B \text{ do } t)K), $$

$$I((\text{if } B \text{ then } (t \text{ while } B \text{ do } t)K) : B \rightarrow I(t(\text{while } B \text{ do } t)K), $$

$$I((\text{if } B \text{ then } (t \text{ while } B \text{ do } t)K) : \neg B \rightarrow I(K) \} $$

$$\cup C(t(\text{while } B \text{ do } t)K) \cup C(K) $$

We first show that $C((\text{while } B \text{ do } t)K) \subseteq_{\text{i.e.}}^{\text{extr}_{h_{bp}}^{GP}}(C((\text{while } B \text{ do } S)K))$. We consider the following label renaming:

$$I((\text{while } B \text{ do } t)K) \mapsto \ell_{-2} $$

$$I((\text{if } B \text{ then } (t \text{ while } B \text{ do } t)K) \mapsto \ell_0 $$

so that it remains to show that $C(t(\text{while } B \text{ do } t)K) \subseteq_{\text{i.e.}}^{\text{extr}_{h_{bp}}^{GP}}(C((\text{while } B \text{ do } S)K))$. Since $t = c_1 t'$, with $t' = c_2...c_{n+1}$, let us analyze the five different cases for the first command $c_1$ of $t$.

(i) $c_1 \equiv x := E;$. Thus, $S_0 \equiv x := E; T(\text{while } B \text{ do } S)K, S_1 \equiv T(\text{while } B \text{ do } S)K, A_0 \equiv x := E$. In this case,

$$C(t(\text{while } B \text{ do } t)K) = \left\{ I(t(\text{while } B \text{ do } t)K) : x := E \rightarrow I(t'(\text{while } B \text{ do } t)K) \} \cup C(t'(\text{while } B \text{ do } t)K). $$

It is enough to consider the relabeling $I(t'(\text{while } B \text{ do } t)K) \mapsto \ell_1$ and to show that $C(t'(\text{while } B \text{ do } t)K) \subseteq_{\text{i.e.}}^{\text{extr}_{h_{bp}}^{GP}}(C((\text{while } B \text{ do } S)K))$.

(ii) $c_1 \equiv \text{skip};$ and $S_0 \equiv \text{skip}; T(\text{while } B \text{ do } S)K$. Thus, $S_1 \equiv T(\text{while } B \text{ do } S)K$, so that $A_0 \equiv \text{skip}$. This case is analogous to the previous case (i).

(iii) $c_1 \equiv \text{skip};$ and $S_0 \equiv (\text{while } B' \text{ do } S')T(\text{while } B \text{ do } S)K$. Thus,

$$S_1 \equiv (\text{if } B' \text{ then } (S'(\text{while } B' \text{ do } S')))T(\text{while } B \text{ do } S)K \text{ and } A_0 \equiv \text{skip}. $$

Here, we have that

$$C(t(\text{while } B \text{ do } t)K) = \left\{ I(t(\text{while } B \text{ do } t)K) : x := E \rightarrow I(t'(\text{while } B \text{ do } t)K) \} \cup C(t'(\text{while } B \text{ do } t)K). $$

Again, we consider the relabeling $I(t'(\text{while } B \text{ do } t)K) \mapsto \ell_1$ and we show that $C(t'(\text{while } B \text{ do } t)K) \subseteq_{\text{i.e.}}^{\text{extr}_{h_{bp}}^{GP}}(C((\text{while } B \text{ do } S)K))$.

(iv) $c_1 \equiv \text{bail } \neg B' \text{ to } T(\text{while } B \text{ do } S)K,$ with $S_0 \equiv (\text{if } B' \text{ then } S')T(\text{while } B \text{ do } S)K$ and $B[B']_{\rho_0} = \text{true}$, so that $S_1 \equiv S'T(\text{while } B \text{ do } S)K$ and $A_0 \equiv B'$. In this case:

$$C(t(\text{while } B \text{ do } t)K) = \left\{ I(t(\text{while } B \text{ do } t)K) : \neg B' \rightarrow I(T(\text{while } B \text{ do } S)K), $$

$$I(t(\text{while } B \text{ do } t)K) : B' \rightarrow I(t'(\text{while } B \text{ do } t)K) \} $$

$$\cup C(T(\text{while } B \text{ do } S)K) \cup C(t'(\text{while } B \text{ do } t)K), $$

$$C(S(\text{while } B \text{ do } S)K) = \left\{ I(S(\text{while } B \text{ do } S)K) : B' \rightarrow I(S'T(\text{while } B \text{ do } S)K), $$

$$I(S(\text{while } B \text{ do } S)K) : \neg B' \rightarrow I(T(\text{while } B \text{ do } S)K) \} $$

$$\cup C(S'T(\text{while } B \text{ do } S)K) \cup C(T(\text{while } B \text{ do } S)K). $$

31
Hence, since \( I(t(\texttt{while } B \texttt{ do } t)K) \rightarrow \ell_0 \) and \( A_0 \equiv B' \), we consider the relabeling \( I(t'(\texttt{while } B \texttt{ do } t)K) \rightarrow \ell_1 \) and we show that \( C(t'(\texttt{while } B \texttt{ do } t)K) \subseteq_{\exp_{\text{gp}}} \exp_{\text{hp}}(C((\texttt{while } B \texttt{ do } S)K)) \).

(v) \( c_1 \equiv \texttt{bail } B' \texttt{ to } (S'T(\texttt{while } B \texttt{ do } S)K) \), with \( S_0 \equiv (\texttt{if } B' \texttt{ then } S')T(\texttt{while } B \texttt{ do } S)K \) and \( B[B']_{\rho_0} = \text{false} \), so that \( S_1 \equiv T(\texttt{while } B \texttt{ do } S)K \) and \( A_0 \equiv \neg B' \). In this case:

\[
C(t(\texttt{while } B \texttt{ do } t)K) = \{I(t(\texttt{while } B \texttt{ do } t)K) : B' \rightarrow I(S'T(\texttt{while } B \texttt{ do } S)K),
I(t(\texttt{while } B \texttt{ do } t)K) : \neg B' \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K)\}
\cup C((S'T(\texttt{while } B \texttt{ do } S)K) \cup C(t'(\texttt{while } B \texttt{ do } t)K),
\]

while \( C(S(\texttt{while } B \texttt{ do } S)K) \) is the same as in the previous point (iv). Hence, since \( I(t(\texttt{while } B \texttt{ do } t)K) \rightarrow \ell_0 \) and \( A_0 \equiv \neg B' \), it is enough to consider the relabeling \( I(t'(\texttt{while } B \texttt{ do } t)K) \rightarrow \ell_1 \) and to show that \( C(t'(\texttt{while } B \texttt{ do } t)K) \subseteq_{\exp_{\text{gp}}} \exp_{\text{hp}}(C((\texttt{while } B \texttt{ do } S)K)) \).

In order to prove this containment, it is left to show that \( C(t'(\texttt{while } B \texttt{ do } t)K) \subseteq_{\exp_{\text{gp}}} \exp_{\text{hp}}(C((\texttt{while } B \texttt{ do } S)K)) \). If \( t' = e \) then the containment boils down to \( C((\texttt{while } B \texttt{ do } t)K) \subseteq_{\exp_{\text{gp}}} \exp_{\text{hp}}(C((\texttt{while } B \texttt{ do } S)K)) \) which is therefore proved. Otherwise, \( t' = c_2 t'' \), so that the containment can be inductively proved by using the same five cases (i)-(v) above.

Let us now show the reverse containment, that is, \( \exp_{\text{hp}}(C((\texttt{while } B \texttt{ do } S)K)) \subseteq_{\exp_{\text{gp}}} C((\texttt{while } B \texttt{ do } S)K) \). For the trace \( t = c_1 c_2 \ldots c_{n+1} \), we know by (\ast) that each command \( c_i \) either is in \{\texttt{skip}; \( x := E \);\} or is one of the two following bail commands (cases (3) and (4) in (\ast)):

\[
\text{bail } B' \text{ to } (T(\texttt{while } B \texttt{ do } S)K), \quad \text{bail } B' \text{ to } (S'T(\texttt{while } B \texttt{ do } S)K).
\]

Furthermore, at least a bail command occurs in \( t \) because there exists at least a conditional command \( H_k \) in \( hp_t \). Let \( c_k \), with \( k \in [1, n+1] \), be the first bail command occurring in \( t \). Thus, since the sequence \( c_1 \ldots c_k \) consists of skip and assignment commands only, we have that \( C(t(\texttt{while } B \texttt{ do } t)K) \supseteq C(c_k \ldots c_{n+1}(\texttt{while } B \texttt{ do } t)K) \). Hence, either \( C(c_k \ldots c_{n+1}(\texttt{while } B \texttt{ do } t)K) \supseteq C(T(\texttt{while } B \texttt{ do } S)K) \) or \( C(c_k \ldots c_{n+1}(\texttt{while } B \texttt{ do } t)K) \supseteq C((\texttt{while } B \texttt{ do } S)K) \). In both cases, we have that \( C(c_k \ldots c_{n+1}(\texttt{while } B \texttt{ do } t)K) \supseteq C((\texttt{while } B \texttt{ do } S)K) \), so that \( C((\texttt{while } B \texttt{ do } S)K) \subseteq C(t(\texttt{while } B \texttt{ do } t)K) \subseteq C((\texttt{while } B \texttt{ do } S)K) \). Thus, it remains to show that

\[
\{ \ell_{-2} : \texttt{skip} \rightarrow \ell_{-1}, \ell_{-1} : B \rightarrow \ell_0, \ell_{-1} : \neg B \rightarrow I(K) \} \cup
\{ \ell_i : A_i \rightarrow \ell_{\text{next}(i)} \mid i \in [0, n] \} \cup \{ \ell_i : \neg A_i \rightarrow I(S_{\text{next}(i)})^c \mid i \in [0, n], A_i \in BExp \}
\]

is contained in \( C(t(\texttt{while } B \texttt{ do } t)K) \). We consider the following label renaming:

\[
\ell_{-2} \rightarrow I((\texttt{while } B \texttt{ do } t)K)
\ell_{-1} \rightarrow I((\texttt{if } B \texttt{ then } t(\texttt{while } B \texttt{ do } t)))K)
\ell_0 \rightarrow I(t(\texttt{while } B \texttt{ do } t)K)
\]

so that it remains to check that for any \( i \in [0, n] \), the commands \( \ell_i : A_i \rightarrow \ell_{\text{next}(i)} \) and \( \ell_i : \neg A_i \rightarrow I(S_{\text{next}(i)})^c \), when \( A_i \in BExp \), are in \( C(t(\texttt{while } B \texttt{ do } t)K) \). We analyze the possible five cases listed in (*) for the action \( A_0 \):

(i) \( A_0 \equiv \texttt{skip} \) because \( S_0 \equiv \texttt{skip} ; T(\texttt{while } B \texttt{ do } S)K \). Here, \( t = \texttt{skip} ; t' \). Hence, \( I(t(\texttt{while } B \texttt{ do } t)K) : \texttt{skip} \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K) \in C(t(\texttt{while } B \texttt{ do } t)K) \) and it is enough to use the relabeling \( \ell_1 \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K) \).

(ii) \( A_0 \equiv x := e \) because \( S_0 \equiv x := E ; T(\texttt{while } B \texttt{ do } S)K \), so that \( t \equiv x := E ; t' \). Analogous to case (i).

(iii) \( A_0 \equiv \texttt{skip} \) because \( S_0 \equiv (\texttt{while } B' \texttt{ do } S')T(\texttt{while } B \texttt{ do } S)K \). Here, \( t = \texttt{skip} ; t' \). Here, again, \( I(t(\texttt{while } B \texttt{ do } t)K) : \texttt{skip} \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K) \in C(t(\texttt{while } B \texttt{ do } t)K) \), so that it is enough to use the relabeling \( \ell_1 \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K) \).

(iv) \( A_0 \equiv B' \) because \( S_0 \equiv (\texttt{if } B' \texttt{ then } S')T(\texttt{while } B \texttt{ do } S)K \) and \( B[B']_{\rho_0} = \text{true} \). Thus, we have that \( t = (\texttt{bail } B'^t \text{ to } (T(\texttt{while } B \texttt{ do } S)K))t' \) and \( S_1 \equiv T(\texttt{while } B \texttt{ do } S)K \). Note that \( I(S_1)^c = I(T(\texttt{while } B \texttt{ do } S)K) \). Hence,

\[
I(t(\texttt{while } B \texttt{ do } t)K) : \neg B' \rightarrow I(T(\texttt{while } B \texttt{ do } S)K) \in C(t(\texttt{while } B \texttt{ do } t)K),
I(t(\texttt{while } B \texttt{ do } t)K) : B' \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K) \in C(t(\texttt{while } B \texttt{ do } t)K).
\]

Once again, the relabeling \( \ell_1 \rightarrow I(t'(\texttt{while } B \texttt{ do } t)K) \) allows us to obtain that \( \ell_0 : B' \rightarrow \ell_1 \) and \( \ell_0 : \neg B' \rightarrow I(T(\texttt{while } B \texttt{ do } S)K) \) are in \( C(t(\texttt{while } B \texttt{ do } t)K) \).

32
(v) \( A_0 \equiv \neg B' \) because \( S_0 \equiv (\text{if } B' \text{ then } S'T(\text{while } B \text{ do } S)K \text{ and } B[\text{false}]p_0 = \text{false} \). Here, we have that \( t = (\text{bail } B' \text{ to } (S'T(\text{while } B \text{ do } S)K))t' \) and \( S_1 \equiv T(\text{while } B \text{ do } S)K \). Note that \( I(S_1)^c = I(S'T(\text{while } B \text{ do } S)K) \). Hence,

\[
I(t(\text{while } B \text{ do } t)K) : B' \rightarrow I(S'T(\text{while } B \text{ do } S)K) \in C(t(\text{while } B \text{ do } t)K),
I(t(\text{while } B' \text{ do } t)K) : \neg B' \rightarrow I(t'(\text{while } B' \text{ do } t)K) \in C(t(\text{while } B \text{ do } t)K).
\]

Thus, through the relabeling \( \ell_1 \mapsto I(t'(\text{while } B \text{ do } t)K) \) we obtain that \( \ell_0 : \neg B' \rightarrow \ell_1 \) and \( \ell_0 : B' \rightarrow I(S'T(\text{while } B \text{ do } S)K) \) are in \( C(t(\text{while } B \text{ do } t)K) \).

This case analysis (i)-(v) for the action \( A_0 \) can be iterated for all the other actions \( A_i \), with \( i \in [1, n] \), and this allows us to close the proof.

Finally, we can also state the correctness of the GP trace extraction transform for the store changes abstraction as follows.

**Theorem 10.12 (Correctness of GP trace extraction).** For any \( P \in \text{Program} \), \( \text{hp} = C_0 \cdots C_n \in \alpha^\text{GP}_{\text{hot}}(T[P]) \), we have that \( \alpha^\text{sc}_{\text{hp}}(T[\text{extr}_{\text{hp}}GP(P)]) = \alpha^\text{sc}_{\text{sc}}(T[P]) \).

The proof of Theorem 10.12 is omitted, since it is a conceptually straightforward adaptation of the proof technique for the analogous Theorem 6.2 of correctness of trace extraction. Let us observe that since \( \alpha^\text{sc}_{\text{sc}} \) is a stronger abstraction of \( \alpha^\text{sc}_{\text{hp}} \) and, by Theorem 10.6, we know that \( \alpha^\text{sc}_{\text{sc}} \) characterizes bisimilarity, we obtain the so-called Stitch lemma in [Guo and Palsberg 2011, Lemma 3.6] as a straight consequence of Theorem 10.12 \( \alpha^\text{sc}_{\text{sc}}(T[\text{extr}_{\text{hp}}GP(P)]) = \alpha^\text{sc}_{\text{sc}}(T[P]) \).

### 11 Further Work

This article put forward a formal model of tracing JIT compilation and correctness of hot path optimization based on trace semantics and abstract interpretation. We see a number of interesting avenues for further work on this topic. First, we expect that most beneficial optimizations employed by tracing compilers of practical dynamic languages like JavaScript, PHP and Python can be formalized and proved correct within our framework. For example, we plan to cast in our model the allocation removal optimization for Python described by [Bolz et al. 2011] in order to formally prove its correctness. Then, 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 [Mozilla Foundation 2013], the current JIT compilation scheme in the Firefox JavaScript engine. Finally, we aim at exploiting the main ideas of our model to study and relate the foundational differences between traditional static vs dynamic tracing compilation.

**Acknowledgements.** 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

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

R. Barbutti, N. De Francesco, A. Santone, and G. Vaglini. 1999. Abstract interpretation of trace semantics for concurrent calculi. *Inform. Process. Lett.* 70, 2 (1999), 69–78.

M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. 2010. 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)*. ACM, New York, NY, USA, 708–725.

I. Böhm, T.J.K. Edler von Koch, S.C. Kyle, B. Franke, and N. Topham. 2011. 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)*. ACM, New York, NY, USA, 74–85.
C.F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. 2011. 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). ACM, ACM, New York, NY, USA, 43–52.

C.F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. 2009. 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). ACM, New York, NY, USA, 18–25.

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

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

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

P. Cousot and R. Cousot. 1977. 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). ACM, New York, NY, USA, 238–252.

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

P. Cousot and R. Cousot. 2002. 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). ACM, New York, NY, USA, 178–190.

S. Dissegna, F. Logozzo, and F. Ranzato. 2014. Tracing compilation by abstract interpretation. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014). ACM, New York, NY, USA, 47–59.

Ecma International. 2011. Standard ECMA-262, ECMAScript Language Specification, Edition 5.1. http://www.ecma-international.org (2011).

Facebook Inc. 2013. The HipHop Virtual Machine. (Oct. 2013). https://www.facebook.com/hphp.

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. 2009. 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). ACM, New York, NY, USA, 465–478.

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

Google Inc. 2010. A new crankshaft for V8. (Dec. 2010). The Chromium Blog.

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

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

C. Häubl and H. Mössenböck. 2011. Trace-based compilation for the Java HotSpot virtual machine. In Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ 2011). ACM, New York, NY, USA, 129–138.

C. Häubl, C. Wimmer, and H. Mössenböck. 2014. Trace transitioning and exception handling in a trace-based JIT compiler for Java. ACM Trans. Archit. Code Optim. 11, 1, Article 6 (Feb. 2014), 26 pages.
H. Inoue, H. Hayashizaki, Peng Wu, and T. Nakatani. 2011. A trace-based Java JIT compiler retrofitted from a method-based compiler. In 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011). IEEE Computer Society, Washington, DC, USA, 246–256.

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

R. Milner. 1995. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ.

Mozilla Foundation. 2010. TraceMonkey. (Oct. 2010). MozillaWiki

Mozilla Foundation. 2013. IonMonkey. (May 2013). MozillaWiki

M. Pall. 2005. The LuaJIT Project. [http://luajit.org](http://luajit.org) (2005).

X. Rival. 2004. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’04)*. ACM, New York, NY, USA, 1–13.

X. Rival and L. Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 26 (2007), 51 pages.

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

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