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

STEFANO DISSEGNA, University of Padova
FRANCESCO LOGOZZO, Facebook Inc.
FRANCESCO RANZATO, University of Padova

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 program paths. Then, hot paths are optimized by exploiting some information on program stores which is available and therefore gathered at runtime. The result is a residual program where the optimized hot paths are guarded by sufficient conditions ensuring some form of equivalence with the original program. The residual program is persistently mutated during its execution, e.g., to add new optimized hot paths or to merge existing paths. Tracing compilation is thus fundamentally different from 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 along hot paths and to prove them correct. We instantiate it to prove the correctness of dynamic type specialization and constant variable folding. 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.

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

Additional Key Words and Phrases: Tracing JIT compilation, abstract interpretation, trace semantics

1. INTRODUCTION

Efficient traditional static compilation of popular dynamic languages like JavaScript, Python and PHP is very hard if not impossible. In particular, these 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

The work of Francesco Logozzo was carried out while being affiliated with Microsoft Research, Redmond, WA, USA. 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 PRAT projects BECOM and ANCORE.

Author's addresses: S. Dissegna and F. Ranzato, Dipartimento di Matematica, University of Padova, Padova, Italy; F. Logozzo, Facebook Inc., Seattle, WA, USA.

DOI: http://dx.doi.org/10.1145/0000000.0000000

order to produce an optimized version of the program. Tracing just-in-time (JIT) compilation (TJITC) [Bala et al. 2000; Bauman et al. 2015; Bebenita et al. 2010; Böhm et al. 2011; Bolz et al. 2009; Bolz et al. 2011; Gal et al. 2006; Gal et al. 2009; Pall 2005; Schilling 2013] has emerged as a valuable implementation and optimization technique for dynamic languages (and not only, e.g. Java [Häubl and Mössenböck 2011; Häubl 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 [Adams et al. 2014; Facebook Inc. 2013; Google Inc. 2010]. The Mozilla Firefox JavaScript engine used to have a tracing engine, called TraceMonkey, which has been later substituted by whole-method just-in-time compilation engines (initially JägerMonkey and then IonMonkey) [Mozilla Foundation 2010; Mozilla Foundation 2013].

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 and constant variables. 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 JIT 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; Cousot and Cousot 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 an abstraction of its trace semantics: the residual program is correct if it is indistinguishable, up to some abstraction of its 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.

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 as abstract interpretations. 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. This choice is made at runtime based on program executions, so it can be seen once again as an abstraction of trace semantics. Here, a simple trace abstraction selects cyclic instruction sequences, i.e. loop paths, that appear at least N times within a single execution trace. These instruction sequences are recorded together with some property of the values assumed by program variables at that point, which is represented as an abstract store belonging to a suitable store abstraction, which in general depends on the successive optimizations to perform.

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 correctness, 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 then be 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 runtime values of their operands. Moreover, as a further application of our model, we consider the constant variable folding optimization along hot paths, which relies on the standard constant propagation abstract domain [Wegman and Zadeck 1991].

Related Work. A formal model for tracing JIT compilation has been put forward by Guo and Palsberg [2011] at POPL symposium. It is based on operational bisimulation

[Milner 1995] to describe the equivalence between source and optimized programs. 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 this 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, Section 5.1]. 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, since 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]. Also, Rival [2004] describes various program optimizations as the trace abstractions they preserve. In the Cousot and Cousot terminology [Cousot and Cousot 2002], Rival's approach corresponds to offline transformations whereas tracing compilation is an online transformation.

Structure. The rest of the paper is organized as follows. Sections 2 and 3 contain some necessary background: the language considered in the paper and its operational trace semantics are defined in Section 2, while Section 3 recalls some basic notions of abstract interpretation, in particular for defining abstract domains of program stores. Hot paths are formally defined in Section 4 as a suitable abstract interpretation of program traces, while Section 5 defines the program transform for extracting a given hot path. The correctness of the hot path extraction transform is defined and proved correct in Section 6, which also introduces in Subsection 6.2 program optimizations along hot paths together with a methodology for proving their correctness. Section 7 applies our model of hot path optimization to type specialization of untyped program commands, while Section 8 describes an application to constant variable folding along hot paths. Nested hot paths and the corresponding program transform for their extraction are the subject of Section 9. Section 10 provides a thorough formal comparison of our model with Guo and Palsberg [2011]'s framework for tracing compilation. Finally, Section 11 concludes, also discussing some directions for future work.

This is an expanded and revised version of the POPL symposium 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: ϵ 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 σ; indices of objects in a nonempty 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 $\sigma_i \in X$ (or $\sigma(i)$) denotes the i-th object in σ; if $\sigma \in X^*$ and $i, j \in [0, |\sigma|)$ then $\sigma_{[i,j]} \in X^*$ denotes the subsequence $\sigma_i \sigma_{i+1} \ldots \sigma_j$, which is therefore the empty sequence if $j < i$, while if $k \in \mathbb{N}$ then $\sigma_{k^{\to}} \in X^*$ denotes the suffix $\sigma_k \sigma_{k+1} \ldots \sigma_{|\sigma|-1}$, which is the empty sequence when $k \geq |\sigma|$.

If $f : X \to Y$ is any function then its collecting version $f^c : \wp(X) \to \wp(Y)$ is defined pointwise by $f^c(S) \triangleq \{f(x) \in Y \mid x \in S\}$, and when clear from the context, by a slight abuse of notation, it is sometimes denoted by f itself.

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 (Ł is the undefined label, where the execution becomes stuck: it is used for defining final commands).

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

For any command $L : A \to L'$, we use the following notation:

$$\mathit{lbl}(L : A \to L') \triangleq L, \quad \mathit{act}(L : A \to L') \triangleq A, \quad \mathit{suc}(L : A \to L') \triangleq L'.$$

Commands $L : B \to L'$ whose action is a Boolean expression are called conditionals. A program $P \in \wp(\mathbb{C})$ is a (possibly infinite, at least in theory) set of commands. In order to be well-formed, if a program P includes a conditional $C \equiv L : B \to L'$ then P must also include a unique complement conditional $L : \neg B \to L''$, which is denoted by $\mathit{cmpl}(C)$ or C^c, where $\neg\neg B$ is taken to be equal to B, so that $\mathit{cmpl}(\mathit{cmpl}(C)) = C$. The set of well-formed programs is denoted by $\mathrm{Program}$. In our examples, programs P will be deterministic, i.e., for any $C_1, C_2 \in P$ such that $\mathit{lbl}(C_1) = \mathit{lbl}(C_2)$: (1) if $\mathit{act}(C_1) \neq \mathit{act}(C_2)$ then $C_1 = \mathit{cmpl}(C_2)$; (2) if $\mathit{act}(C_1) = \mathit{act}(C_2)$ then $C_1 = C_2$. We say that two programs P_1 and P_2 are equal up to label renaming, denoted by $P_1 \cong P_2$, 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 values ranging in Value, possibly undefined values ranging in $\mathrm{Value}_{\mathrm{u}}$, truth values in Bool, possibly undefined truth values ranging in $\mathrm{Bool}_{\mathrm{u}}$ and type names ranging in Types, which are defined as follows:

$$\mathrm{Value} \triangleq \mathbb{Z} \cup \mathrm{Char}^* \qquad \mathrm{Value}_{\mathrm{u}} \triangleq \mathbb{Z} \cup \mathrm{Char}^* \cup \{\mathit{undef}\}$$

$$\mathrm{Bool} \triangleq \{\mathit{true}, \mathit{false}\} \qquad \mathrm{Bool}_{\mathrm{u}} \triangleq \{\mathit{true}, \mathit{false}, \mathit{undef}\}$$

$$\mathrm{Types} \triangleq \{\mathrm{Int}, \mathrm{String}, \mathrm{Undef}, \top_{\mathrm{T}}, \bot_{\mathrm{T}}\}$$

$$\mathbf{E} : \mathrm{Exp} \to \mathrm{Store} \to \mathrm{Value}_{\mathrm{u}}$$

$$\mathbf{E}[\![v]\!]\rho \triangleq v \quad \mathbf{E}[\![x]\!]\rho \triangleq \rho(x)$$

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

$$\mathbf{B} : \mathrm{BExp} \to \mathrm{Store} \to \mathrm{Bool}_{\mathrm{u}}$$

$$\mathbf{B}[\![\mathrm{tt}]\!]\rho \triangleq \mathit{true} \quad \mathbf{B}[\![\mathrm{ff}]\!]\rho \triangleq \mathit{false}$$

$$\mathbf{B}[\![E_1 \leq E_2]\!]\rho \triangleq \begin{cases} \mathbf{E}[\![E_1]\!]\rho \leq_{\mathbb{Z}} \mathbf{E}[\![E_2]\!]\rho & \text{if } \mathit{type}(\mathbf{E}[\![E_i]\!]\rho) = \mathrm{Int} \\ \exists \sigma \in \mathrm{String} \,.\, \mathbf{E}[\![E_2]\!]\rho = (\mathbf{E}[\![E_1]\!]\rho) \cdot \sigma & \text{if } \mathit{type}(\mathbf{E}[\![E_i]\!]\rho) = \mathrm{String} \\ \mathit{undef} & \text{otherwise} \end{cases}$$

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

$$\mathbf{A} : \mathbb{A} \to \mathrm{Store} \to \mathrm{Store} \cup \{\bot\}$$

$$\mathbf{A}[\![\mathbf{skip}]\!]\rho \triangleq \rho$$

$$\mathbf{A}[\![x := E]\!]\rho \triangleq \begin{cases} \rho[x/\mathbf{E}[\![E]\!]\rho] & \text{if } \mathbf{E}[\![E]\!]\rho \neq \mathit{undef} \\ \bot & \text{if } \mathbf{E}[\![E]\!]\rho = \mathit{undef} \end{cases}$$

$$\mathbf{A}[\![B]\!]\rho \triangleq \begin{cases} \rho & \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{true} \\ \bot & \text{if } \mathbf{B}[\![B]\!]\rho \in \{\mathit{false}, \mathit{undef}\} \end{cases}$$

Fig. 1. Semantics of program expressions and actions.

where Char is a nonempty finite set of characters and *undef* is a distinct symbol. The mapping $\mathit{type} : \mathrm{Value}_{\mathrm{u}} \to \mathrm{Types}$ provides the type of any possibly undefined value:

$$\mathit{type}(v) \triangleq \begin{cases} \mathrm{Int} & \text{if } v \in \mathbb{Z} \\ \mathrm{String} & \text{if } v \in \mathrm{Char}^* \\ \mathrm{Undef} & \text{if } v = \mathit{undef} \end{cases}$$

The type names $\bot_{\mathrm{T}}$ and $\top_{\mathrm{T}}$ will be used in Section 7 as, respectively, top and bottom type, that is, subtype and supertype of all types.

Let $\mathrm{Store} \triangleq \mathrm{Var} \to \mathrm{Value}_{\mathrm{u}}$ denote the set of possible stores on variables in Var, where $\rho(x) = \mathit{undef}$ means that the store ρ is not defined on a program variable $x \in \mathrm{Var}$. Hence, let us point out that the symbol *undef* will be used to represent both store undefinedness and a generic error when evaluating an expression (e.g., additions and comparisons between integers and strings), two situations which are not distinguished in our semantics. A store $\rho \in \mathrm{Store}$ will be denoted by $[x/\rho(x)]_{\rho(x) \neq \mathit{undef}}$, thus omitting undefined variables, while $[\,]$ will denote the totally undefined store. If $P \in \mathrm{Program}$ then $\mathit{vars}(P)$ denotes the set of variables in Var that occur in P, so that $\mathrm{Store}_P \triangleq \mathit{vars}(P) \to \mathrm{Value}_{\mathrm{u}}$ is the set of possible stores for P.

The semantics of expressions $\mathbf{E}$, Boolean expressions $\mathbf{B}$ and program actions $\mathbf{A}$ is standard and goes as defined in Fig. 1. Let us remark that:

(i) the binary function $+_{\mathbb{Z}}$ denotes integer addition, $\leq_{\mathbb{Z}}$ denotes integer comparison, while $\cdot$ is string concatenation;

(ii) logical negation and conjunction in $\mathrm{Bool_u}$ are extended in order to handle *undef* as follows: $\neg$*undef* = *undef* and *undef* $\wedge\, b =$ *undef* $= b \wedge$ *undef*;
(iii) $\rho[x/v]$ denotes a store update for the variable x with $v \in \mathrm{Value}$;
(iv) the distinct symbol $\bot \notin \mathrm{Value_u}$ is used to denote the result of: $\mathbf{A}[\![x := E]\!]\rho$ when the evaluation of the expression E for ρ generates an error; $\mathbf{A}[\![B]\!]\rho$ when the evaluation of the Boolean expression B for ρ is either *false* or generates an error.

With a slight abuse of notation we also consider the collecting versions of the semantic functions in Fig. 1, which are defined as follows:

$$\begin{aligned}&\mathbf{E} : \mathrm{Exp} \to \wp(\mathrm{Store}) \to \wp(\mathrm{Value_u})\\ &\mathbf{E}[\![E]\!]S \triangleq \{\mathbf{E}[\![E]\!]\rho \in \mathrm{Value_u} \mid \rho \in S\}\end{aligned}$$

$$\begin{aligned}&\mathbf{B} : \mathrm{BExp} \to \wp(\mathrm{Store}) \to \wp(\mathrm{Store})\\ &\mathbf{B}[\![B]\!]S \triangleq \{\rho \in S \mid \mathbf{B}[\![B]\!]\rho = \textit{true}\}\end{aligned}$$

$$\begin{aligned}&\mathbf{A} : \mathbb{A} \to \wp(\mathrm{Store}) \to \wp(\mathrm{Store})\\ &\mathbf{A}[\![A]\!]S \triangleq \{\mathbf{A}[\![A]\!]\rho \mid \rho \in S,\ \mathbf{A}[\![A]\!]\rho \in \mathrm{Store}\}\end{aligned}$$

Let us point out that, in the above collecting versions, if $\mathbf{E}[\![E]\!]\rho = \textit{undef}$ then $\mathbf{E}[\![E]\!]\{\rho\} = \{\textit{undef}\}$ and $\mathbf{A}[\![x := E]\!]\{\rho\} = \varnothing$, while if $\mathbf{B}[\![B]\!]\rho \in \{\textit{false}, \textit{undef}\}$ then $\mathbf{B}[\![B]\!]\{\rho\} = \varnothing$ and $\mathbf{A}[\![B]\!]\{\rho\} = \varnothing$.

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

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

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

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

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

A state $s \in \mathrm{State}_P$ is stuck for P when $\mathbf{S}[\![P]\!]s = \varnothing$. Let us point that:

(i) If the conditional command of a state $s = \langle\rho, L : B \to L'\rangle \in \mathrm{State}_P$ is such that $\mathbf{B}[\![B]\!]\rho = \textit{false}$ then s is stuck for P because there exists no store $\rho' \in \mathbf{A}[\![B]\!]\{\rho\} = \varnothing$.
(ii) If the command of a state $s = \langle\rho, L : A \to \text{Ł}\rangle \in \mathrm{State}_P$ has the undefined label Ł as next label then s is stuck for P.
(iii) We have a stuck state s when an error happens. E.g., this is the case for an undefined evaluation of an addition as in $s = \langle[y/3, z/\texttt{foo}], L : x := y + z \to L'\rangle$ and for an undefined evaluation of a Boolean expression as in $s = \langle[y/3, z/\texttt{foo}], L : y \le x \to L'\rangle$.

Programs typically have an entry point, and this is modeled through a distinct initial label $L_\iota \in \mathbb{L}$ from which execution starts. $\mathrm{State}_P^\iota \triangleq \{\langle\rho, C\rangle \mid \mathit{lbl}(C) = L_\iota\}$ 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 $\mathbf{S}$. Hence, the set Trace of partial traces is defined as follows:

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

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

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

A trace $\sigma \in \text{Trace}_P$ is complete if for any state $s \in \text{State}_P$, $\sigma s \notin \text{Trace}_P$ and $s\sigma \notin \text{Trace}_P$. Observe that 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 $\sigma_0 \notin \text{State}^\iota_P$. Traces of P starting from initial states are denoted by

$$\mathbf{T}^\iota[\![P]\!] = \text{Trace}^\iota_P \triangleq \{\sigma \in \text{Trace}_P \mid \sigma_0 \in \text{State}^\iota_P\}.$$

Also, a complete trace $\sigma \in \text{Trace}^\iota_P$ such that $suc(\sigma_{|\sigma|-1}) = \text{Ł}$ 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** $x := x + 3;$
```

Its translation as a program P in our language is given below, with $L_\iota = L_0$, where, with a little abuse, we assume an extended syntax that allows expressions like $x\%3 = 0$.

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

Its trace semantics from initial states Trace^ι_P includes the following complete traces, where $[\,]$ is the initial totally undefined store.

$\langle[\,], C_0\rangle\langle[x/0], C_1^c\rangle$
$\langle[\,], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3\rangle$
$\langle[\,], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle\langle[x/1], C_1^c\rangle$
$\ldots$
$\ldots$
$\langle[\,], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle\langle[x/1], C_1\rangle \cdots \langle[x/21], C_4\rangle\langle[x/24], C_1\rangle$
$\langle[\,], C_0\rangle\langle[x/0], C_1\rangle\langle[x/0], C_2\rangle\langle[x/1], C_3^c\rangle\langle[x/1], C_1\rangle \cdots \langle[x/21], C_4\rangle\langle[x/24], C_1^c\rangle\langle[x/24], C_5\rangle$

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; Cousot and Cousot 1979], abstract domains, also called abstractions, are specified by Galois connections/insertions (GCs/GIs for short) or, equivalently, adjunctions. Concrete and abstract domains, $\langle C, \leq_C\rangle$ and $\langle A, \leq_A\rangle$, are assumed to be complete lattices which are related by abstraction and concretization maps $\alpha : C \to A$ and

$\gamma : A \to C$ such that, for all a and c, $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. A GC is a GI when $\alpha \circ \gamma = \lambda x.x$. It is well known that a join-preserving α uniquely determines, by adjunction, γ as follows: $\gamma(a) = \vee\{c \in C \mid \alpha(c) \leq_A a\}$; conversely, a meet-preserving γ uniquely determines, by adjunction, α as follows: $\alpha(c) = \wedge\{a \in A \mid c \leq_C \gamma(a)\}$.

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 GC. Then, $f^\sharp$ is a correct abstract interpretation of f on A when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ holds, where $\sqsubseteq$ denotes the pointwise ordering between functions. Moreover, the abstract function $f^A \triangleq \alpha \circ f \circ \gamma : A \to A$ is called the best correct approximation of f on A because any abstract function $f^\sharp$ is correct iff $f^A \sqsubseteq f^\sharp$. Hence, for any A, f^A plays the role of the best possible approximation of f on the abstraction A.

3.2. Store Abstractions

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

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

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

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 $\mathrm{Value}^\sharp$ be an abstraction of sets of possibly undefined values in $\wp(\mathrm{Value_u})$ as formalized by a Galois connection

$$(\alpha_{value}, \langle\wp(\mathrm{Value_u}), \subseteq\rangle, \langle\mathrm{Value}^\sharp, \leq_{\mathrm{Value}^\sharp}\rangle, \gamma_{value}).$$

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

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

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

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

The corresponding concretization map $\gamma^{\sqsubseteq}_{value} : \mathrm{Store}^\sharp_{value} \to \wp(\mathrm{Store})$ is defined, as recalled in Section 3.1, by adjunction from the abstraction map $\alpha^{\sqsubseteq}_{value}$ and it is easy to check that it can be given as follows:

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

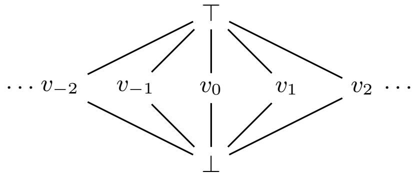
Let us observe that:

(i) $\alpha^{\sqsubseteq}_{value}(\varnothing) = \lambda x.\alpha_{value}(\varnothing) = \lambda x.\bot_{\mathrm{Value}^\sharp}$ because $\alpha_{value}(\varnothing) = \bot_{\mathrm{Value}^\sharp}$ always holds in a GC;

(ii) $\alpha^{\sqsubseteq}_{value}(\{[\,]\}) = \lambda x.\alpha_{value}(\{\mathit{\mathbf{undef}}\})$;

(iii) if $\gamma_{value}(\bot_{\text{Value}^\sharp}) = \varnothing$, $\rho^\sharp \in \text{Store}^\sharp_{value}$ and $\rho^\sharp(x) = \bot_{\text{Value}^\sharp}$ then $\gamma^{\sqsubseteq}_{value}(\rho^\sharp) = \varnothing$;
(iv) if $\gamma_{value}(\bot_{\text{Value}^\sharp}) = \{\textit{undef}\}$ then $\gamma^{\sqsubseteq}_{value}(\lambda x.\bot_{\text{Value}^\sharp}) = \{[\,]\}$.

Example 3.1 (***The constant propagation abstraction***). The constant propagation (see [Wegman and Zadeck 1991]) lattice $\langle \text{CP}, \preceq \rangle$ is depicted below.



where $\{v_i\}_{i\in\mathbb{Z}}$ is any enumeration of Value_u, thus *undef* is included. Abstraction $\alpha_{cp} : \wp(\text{Value}_\text{u}) \to \text{CP}$ and concretization $\gamma_{cp} : \text{CP} \to \wp(\text{Value}_\text{u})$ functions are defined as follows:

$$\alpha_{cp}(S) \triangleq \begin{cases} \bot & \text{if } S = \varnothing \\ v_i & \text{if } S = \{v_i\} \\ \top & \text{otherwise} \end{cases} \qquad \gamma_{cp}(a) \triangleq \begin{cases} \varnothing & \text{if } a = \bot \\ \{v_i\} & \text{if } a = v_i \\ \text{Value}_\text{u} & \text{if } a = \top \end{cases}$$

and give rise to a GI $(\alpha_{cp}, \langle \wp(\text{Value}_\text{u}), \subseteq \rangle, \langle \text{CP}, \preceq \rangle, \gamma_{cp})$. The corresponding nonrelational store abstraction is denoted by $\text{CP}_{st} \triangleq \langle \text{Var} \to \text{CP}, \dot{\preceq} \rangle$, where $\alpha_{\text{CP}} : \wp(\text{Store}) \to \text{CP}_{st}$ and $\gamma_{\text{CP}} : \text{CP}_{st} \to \wp(\text{Store})$ denote the abstraction and concretization maps. For example, for $\text{Var} = \{x, y, z, w\}$ and omitting the bindings $v/\textit{undef}$ also in abstract stores, we have that:

$$\alpha_{\text{CP}}(\{[x/2, y/\texttt{foo}, z/1], [x/2, y/\texttt{bar}]\}) = [x/2, y/\top, z/\top],$$
$$\gamma_{\text{CP}}([x/2, y/\top, w/\texttt{foo}]) = \{\rho \in \text{Store} \mid \rho(x) = 2, \rho(y) \in \text{Value}_\text{u}, \rho(z) = \textit{undef}, \rho(w) = \texttt{foo}\},$$
$$\gamma_{\text{CP}}([x/2, y/\top, w/\bot]) = \varnothing. \quad \square$$

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 Trace_P, we first define a mapping $\textit{loop} : \text{Trace}_P \to \wp(\text{Trace}_P)$ 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) τ of a program trace σ such that:

(1) the successor command in σ of the last state in τ exists and coincides with the command – or its complement, when this is the last loop iteration – of the first state in τ;
(2) there is no other such command within τ (otherwise the sequence τ would contain multiple iterations);
(3) the last state of τ performs a backward jump in the program P.

To recognize backward jumps, we consider a topological order on the control flow graph of commands in P, denoted by $\lessdot$. This leads to the following formal definition:

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

Let us remark that a loop path

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

may contain some sub-loop path, namely it may happen that $\mathit{loop}(\langle\rho_i, C_i\rangle \cdots \langle\rho_j, C_j\rangle) \neq \varnothing$ so that some commands C_k, with $k \in [i, j]$, 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 includes a nested while loop.

We abuse notation by using α_{store} to denote a map $\alpha_{store} : \mathrm{Trace}_P \to \mathrm{Trace}^\sharp_P$ which "abstracts" a program trace τ into $\mathrm{Trace}^\sharp_P$ by abstracting the sequence of stores occurring in τ:

$$\alpha_{store}(\langle\rho_0, C_0\rangle \cdots \langle\rho_n, C_n\rangle) \triangleq \langle\alpha_{store}(\{\rho_0\}), C_0\rangle \cdots \langle\alpha_{store}(\{\rho_n\}), C_n\rangle.$$

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

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

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

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

Hence, hot^N can be defined as follows:

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

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

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

A N-hot path hp in a program P is therefore any $hp \in \alpha^N_{hot}(\mathrm{Trace}_P)$ and is compactly denoted as $hp = \langle a_0, C_0, ..., a_n, C_n\rangle$. 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 successor function *next* for indices in a hot path $\langle a_0, C_0, ..., a_n, C_n\rangle$ as follows: $\mathit{next} \triangleq \lambda i.\ i = n\, ?\, 0 : i + 1$. For a N-hot path $\langle a_0, C_0, ..., a_n, C_n\rangle \in \alpha^N_{hot}(\mathrm{Trace}_P)$, for any $i \in [0, n]$, if C_i is a conditional command $L_i : B_i \to L_{\mathit{next}(i)}$ then throughout the paper its complement $C^c_i = \mathit{cmpl}(C_i)$ will be also denoted by $L_i : \neg B_i \to L^c_{\mathit{next}(i)}$.

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

$$\begin{aligned} hp_1 = \langle &\top, C_1 \equiv L_1 : x \leq 20 \to L_2, \top, C_2 \equiv L_2 : x := x+1 \to L_3, \\ &\top, C_3^c \equiv L_3 : \neg(x\%3 = 0) \to L_1\rangle; \end{aligned}$$

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

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:

$$? \mapsto 0 \mapsto 1 \mapsto 2 \mapsto 3 \mapsto 6 \mapsto 7 \mapsto 8 \mapsto 9 \mapsto 12 \mapsto 13 \mapsto 14 \mapsto 15 \mapsto 18 \mapsto 19 \mapsto 20 \mapsto 21 \mapsto 24$$

Hence, if σ is the complete terminating trace of P in Example 2.1 then it turns out that $count(\alpha_{store}(\sigma), hp_1) = 8$ and $count(\alpha_{store}(\sigma), hp_2) = 4$. □

5. TRACE EXTRACTION

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

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

In turn, we also have program actions $guard\ E_a \in \mathbb{A}$ such that:

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

Let P be a program and $hp = \langle a_0, C_0, ..., a_n, C_n\rangle \in \alpha^N_{hot}(\mathrm{Trace}_P)$ be a hot path on some store abstraction $\mathrm{Store}^\sharp$. 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 $guard\ E_{a_i}$ induced by the corresponding store property a_i. To this aim, we consider three *injective* relabeling functions

$$\ell : [0,n] \to \mathbb{L}_1 \qquad \mathbb{l} : [1,n] \to \mathbb{L}_2 \qquad \overline{(\cdot)} : \mathbb{L} \to \overline{\mathbb{L}} \qquad (*)$$

where $\mathbb{L}_1$, $\mathbb{L}_2$ and $\overline{\mathbb{L}}$ are pairwise disjoint sets of fresh labels, so that $labels(P) \cap (\mathbb{L}_1 \cup \mathbb{L}_2 \cup \overline{\mathbb{L}}) = \varnothing$. 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:

$$\begin{aligned} extr_{hp}(P) \triangleq P \smallsetminus \big(&\{C_0\} \cup \{cmpl(C_0) \mid cmpl(C_0) \in P\}\big) \\ &\cup \{\overline{L_0} : act(C_0) \to L_1\} \cup \{\overline{L_0} : \neg act(C_0) \to L_1^c \mid cmpl(C_0) \in P\} \cup stitch_P(hp) \end{aligned}$$

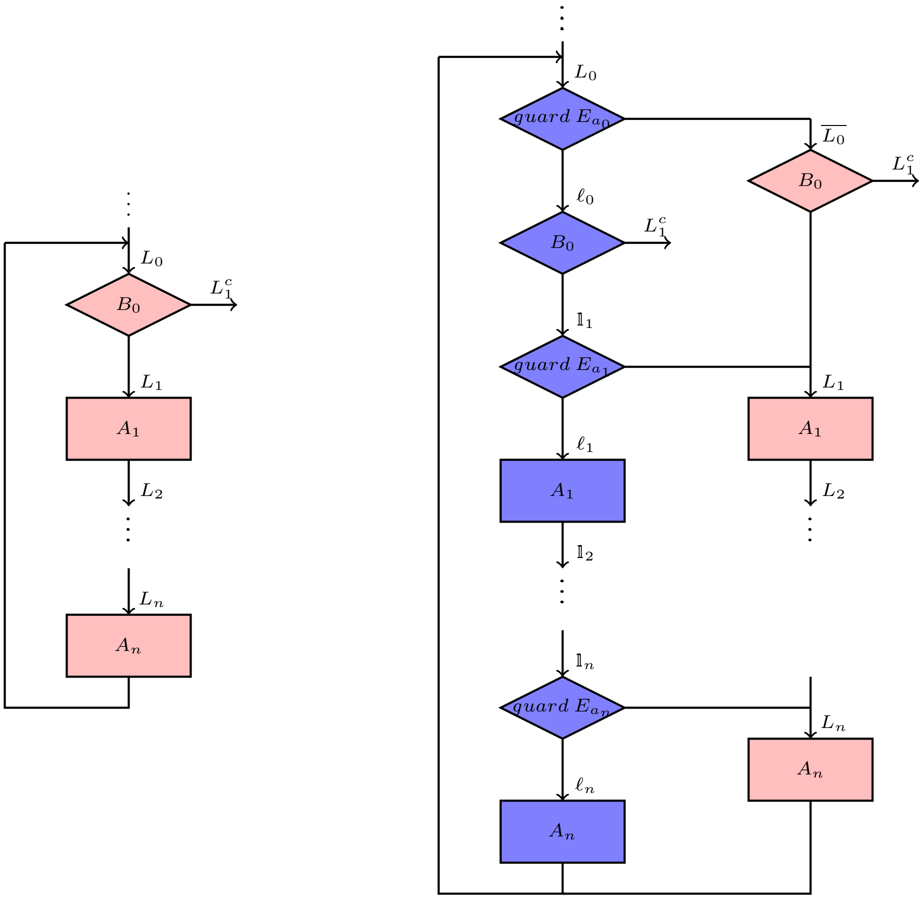


Fig. 2. An example of trace extraction transform: on the left, a hot path hp with commands in pink (in black/white: loosely dotted) shapes; on the right, the corresponding trace transform $extr_{hp}(P)$ with new commands in blue (in black/white: densely dotted) shapes.

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

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

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

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 $P_{new} = extr_{hp}(P) \smallsetminus P =$

$\mathit{stitch}_P(hp) \cup \{\overline{L_0} : \mathit{act}(C_0) \to L_1\} \cup \{\overline{L_0} : \neg \mathit{act}(C_0) \to L_1^c \mid \mathit{cmpl}(C_0) \in P\}$ we also have a unique complement conditional in P_{new}. Moreover, observe that if P is deterministic then $\mathit{extr}_{hp}(P)$ still is deterministic. □

Let us remark that the stitch of the hot path hp into P is always a linear sequence of different commands, namely, $\mathit{stitch}_P(hp)$ does not contain loops nor join points. Furthermore, this happens even if the hot path hp does contain some inner sub-loop. Technically, this is achieved as a consequence of the fact that the above relabeling functions ℓ and $\mathbb{l}$ are required to be injective. Hence, even if some command C occurs more than once inside hp, e.g., $C_i = C = C_j$ for some $i, j \in [0, n-1]$ with $i \neq j$, then these multiple occurrences of C in hp are transformed into differently labeled commands in $\mathit{stitch}_P(hp)$, e.g., because $\ell_i \neq \ell_j$ and $\mathbb{l}_{i+1} \neq \mathbb{l}_{j+1}$.

Let us now illustrate the trace extraction transform on a first simple example.

Example 5.3. Let us consider the program P in Example 2.1 and the hot path $hp = \langle \top, C_1, \top, C_2, \top, C_3^c \rangle$ in Example 4.1 (denoted there by hp_1), where stores are abstracted to the trivial one-point abstraction $\mathrm{Store}^\sharp = \{\top\}$. Here, for any $\rho \in \mathrm{Store}$, we have that $\mathbf{B}[\![\mathit{guard}\ E_\top]\!]\rho = \textbf{\textit{true}}$. The trace extraction transform of P w.r.t. hp is therefore as follows:

$$\mathit{extr}_{hp}(P) = P \smallsetminus \{C_1, C_1^c\} \cup \{\overline{L_1} : x \le 20 \to L_2,\ \overline{L_1} : \neg(x \le 20) \to L_5\} \cup \mathit{stitch}_P(hp)$$

where

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

The flow graph of $\mathit{extr}_{hp}(P)$ is depicted in Figure 3, while a higher level representation using while-loops and gotos is as follows:

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

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

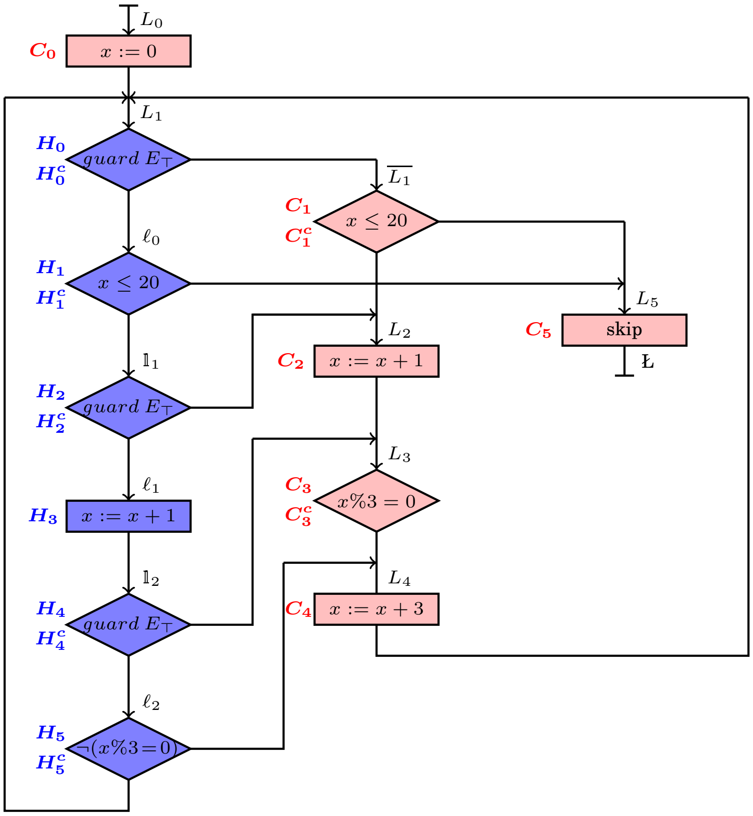


Fig. 3. The flow graph of the trace extraction transform $extr_{hp}(P)$ in Example 5.3, where commands of $stitch_P(hp)$ are in blue (in black/white: densely dotted) shapes, while commands of the source program P are in pink (in black/white: loosely dotted) shapes.

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/optimized program.

Store Changes Abstraction. The approach by Guo and Palsberg [2011] to tracing compilation basically relies on a notion of correctness that requires the same *store changes* to happen in both the transformed/optimized program and the original program. This can be easily encoded by an observational abstraction $\alpha_{sc} : \wp(\mathrm{Trace}) \to \wp(\mathrm{Store}^*)$ of partial traces that observes store changes in execution traces:

$$sc : \mathrm{Trace} \to \mathrm{Store}^*$$

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

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

Since the function α_{sc} obviously preserves arbitrary set unions, as recalled in Section 3.1, it admits a right adjoint $\gamma_{sc} : \wp(\mathrm{Store}^*) \to \wp(\mathrm{Trace})$ defined as $\gamma_{sc}(S) \triangleq \cup\{T \in$

$\wp(\mathrm{Trace}) \mid \alpha_{sc}(T) \subseteq S\}$, that gives rise to a GC $(\alpha_{sc}, \langle\wp(\mathrm{Trace}), \subseteq\rangle, \langle\wp(\mathrm{Store}^*), \subseteq\rangle, \gamma_{sc})$. By a slight abuse of notation, α_{sc} is also used as an abstraction of the partial trace semantics of a given program P, that is, $\alpha_{sc} : \wp(\mathrm{Trace}_P) \to \wp(\mathrm{Store}^*_P)$, which, clearly, gives rise to a corresponding GC $(\alpha_{sc}, \langle\wp(\mathrm{Trace}_P), \subseteq\rangle, \langle\wp(\mathrm{Store}^*_P), \subseteq\rangle, \gamma_{sc})$.

Output Abstraction. The store changes abstraction α_{sc} may be too strong in practice. This can be generalized to any observational abstraction of execution traces $\alpha_o : \langle\wp(\mathrm{Trace}), \subseteq\rangle \to \langle A, \leq_A\rangle$ (which gives rise to a GC). As a significant example, one may consider an output abstraction that demands to have the same stores (possibly restricted to some subset of program variables) only at some specific output points. For example, in a language with no explicit output primitives, as that considered by Guo and Palsberg [2011], one 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 we consider a language including a distinct primitive command "$put\ \mathcal{X}$" that "outputs" the value of program variables ranging in some set $\mathcal{X}$ then we may want to have the same stores for variables in $\mathcal{X}$ at each output point $put\ \mathcal{X}$. In this case, optimizations should preserve the same sequence of outputs, i.e. optimizations should not modify the order of output commands. More formally, this can be achieved by adding a further sort of actions: $put\ \mathcal{X} \in \mathbb{A}$, where $\mathcal{X} \subseteq \mathrm{Var}$ is a set of program variables. The semantics of $put\ \mathcal{X}$ obviously does not affect program stores, i.e., $\mathbf{A}[\![put\ \mathcal{X}]\!]\rho \triangleq \rho$. Correspondingly, if $\mathrm{Store}_{\mathcal{X}}$ denotes stores on variables ranging in $\mathcal{X}$ then the following output abstraction $\alpha_{out} : \wp(\mathrm{Trace}) \to \wp(\mathrm{Store}^*_{\mathcal{X}})$ of partial traces observes program stores at output program points only:

$$out : \mathrm{Trace} \to \mathrm{Store}^*_{\mathcal{X}}$$

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

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

where $\rho_{|\mathcal{X}}$ denotes the restriction of the store ρ to variables in $\mathcal{X}$. Similarly to α_{sc}, here again we have a GC $(\alpha_o, \langle\wp(\mathrm{Trace}), \subseteq\rangle, \langle\wp(\mathrm{Store}^*_{\mathcal{X}}), \subseteq\rangle, \gamma_o)$.

Example 6.1 (***Dead store elimination***). This 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 based on the store changes abstraction, such as dead store elimination. For example, in a program fragment such as

```
while (x ≤ 0) do
  z := 0;
  x := x + 1;
  z := 1;
```

one can extract the hot path $hp = \langle x \leq 0, z := 0, x := x + 1, z := 1\rangle$ (here we ignore 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 by choosing and then formalizing an observational abstraction of program traces which requires to have the same stores at the beginning and at the exit of loops containing an extracted hot path, while outside of hot paths one could still consider the store changes abstraction. □

Observational Abstraction. One can generalize the store changes abstraction α_{sc} by considering any observational abstraction $\alpha_o : \langle \wp(\text{Trace}), \subseteq \rangle \rightarrow \langle A, \leq_A \rangle$ which is less precise (i.e., more approximate) than α_{sc}: this means that for any $T_1, T_2 \in \wp(\text{Trace})$, if $\alpha_{sc}(T_1) = \alpha_{sc}(T_2)$ then $\alpha_o(T_1) = \alpha_o(T_2)$, or, equivalently, for any $T \in \wp(\text{Trace})$, $\gamma_{sc}(\alpha_{sc}(T)) \subseteq \gamma_o(\alpha_o(T))$. Informally, this means that α_o abstracts more information than α_{sc}. As an example, when considering programs with output actions, the following abstraction $\alpha_{osc} : \wp(\text{Trace}) \rightarrow \wp(\text{Store}^*_{\mathcal{X}})$ observes store changes at output program points only:

$$osc : \text{Trace} \rightarrow \text{Store}^*_{\mathcal{X}}$$

$$osc(\sigma) \triangleq \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \text{ or } \sigma = \langle \rho, C \rangle,\ act(C) \neq put\ \mathcal{X} \\ \rho_{|\mathcal{X}} & \text{if } \sigma = \langle \rho, C \rangle,\ act(C) = put\ \mathcal{X}, \\ osc(\langle \rho, L_1 : put\ \mathcal{X} \rightarrow L'_1 \rangle \sigma') & \text{if } \sigma = \langle \rho, C_0 \rangle \langle \rho, L_1 : A_1 \rightarrow L'_1 \rangle \sigma',\ act(C_0) = put\ \mathcal{X} \\ osc(\langle \rho, L_1 : A_1 \rightarrow L'_1 \rangle \sigma') & \text{if } \sigma = \langle \rho, C_0 \rangle \langle \rho, L_1 : A_1 \rightarrow L'_1 \rangle \sigma',\ act(C_0) \neq put\ \mathcal{X} \\ \rho_{0|\mathcal{X}}\ osc(\langle \rho_1, C_1 \rangle \sigma') & \text{if } \sigma = \langle \rho_0, C_0 \rangle \langle \rho_1, C_1 \rangle \sigma',\ \rho_0 \neq \rho_1,\ act(C_0) = put\ \mathcal{X} \\ osc(\langle \rho_1, C_1 \rangle \sigma') & \text{if } \sigma = \langle \rho_0, C_0 \rangle \langle \rho_1, C_1 \rangle \sigma',\ \rho_0 \neq \rho_1,\ act(C_0) \neq put\ \mathcal{X} \end{cases}$$

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

Clearly, it turns out that α_{osc} is more approximate than α_{sc} since $osc(\sigma)$ records a store change $\rho_0\rho_1$ only when the two contiguous subsequences of commands whose common stores are ρ_0 and ρ_1 contain among them at least a put command.

6.1. Correctness of Trace Extraction

It turns out that the observational correctness of the hot path extraction transform in Definition 5.1 can be proved w.r.t. the observational abstraction α_{sc} of store changes.

THEOREM 6.2 (**CORRECTNESS OF TRACE EXTRACTION**). *For any* $P \in \text{Program}$ *and* $hp \in \alpha^N_{hot}(\text{Trace}_P)$*, we have that* $\alpha_{sc}(\mathbf{T}[\![extr_{hp}(P)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$.

This is the crucial result concerning the correctness of our hot path extraction transform. We will show in Section 10.5 (see Theorem 10.12) that the correctness of the hot path extraction strategy defined in [Guo and Palsberg 2011] can be proved by a simple adaptation of the proof technique that we will use here.

In order to prove Theorem 6.2, we need to define some suitable "dynamic" transformations of execution traces. Let us fix a hot path $hp = \langle a_0, C_0, ..., a_n, C_n \rangle \in \alpha^N_{hot}(\text{Trace}_P)$ (w.r.t. some store abstraction) and let $P_{hp} \triangleq extr_{hp}(P)$ denote the corresponding transform of P given by Definition 5.1. We first define a mapping tr^{out}_{hp} of the execution traces of the program P into execution traces of the transformed program P_{hp} that unfolds the hot path hp (or any prefix of it) according to the hot path extraction strategy given by Definition 5.1: a function application $\text{tr}^{out}_{hp}(\tau)$ should replace any occurrence of the hot path hp in the execution trace $\tau \in \text{Trace}_P$ with its corresponding guarded and suitably relabeled path obtained through Definition 5.1. More precisely, Fig. 4 provides the definitions for the following two functions:

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

Let us first describe how the trace transform tr^{out}_{hp} works. A function application $\text{tr}^{out}_{hp}(s\sigma)$ on a trace $s\sigma$ of P—the superscript out hints that the first state s of the trace $s\sigma$ is still *outside* of the hot path hp so that $\text{tr}^{out}_{hp}(s\sigma)$ could either enter into the transform of hp or remain outside of hp—triggers the unfolding of the hot path hp in P_{hp} when the first state s is such that:

(i) $s = \langle \rho, C_0 \rangle$, where C_0 is the first command of hp;
(ii) the entry conditional $guard\ E_{a_0}$ of $stitch_P(hp)$ is satisfied in the store ρ of the state $s = \langle \rho, C_0 \rangle$, that is, $\alpha_{store}(\{\rho\}) \leq a_0$.

If the unfolding for the trace $\langle \rho, C_0 \rangle \sigma$ is actually started by applying $\mathbf{tr}^{out}_{hp}(\langle \rho, C_0 \rangle \sigma)$ then:

(iii) the first state $\langle \rho, C_0 \rangle$ is unfolded into the following sequence of two states of P_{hp}: $\langle \rho, L_0 : guard\ E_{a_0} \to \ell_0 \rangle \langle \rho, \ell_0 : act(C_0) \to \mathbb{L}_1 \rangle$;
(iv) in turn, the unfolding of the residual trace σ is carried on by applying $\mathbf{tr}^{in}_{hp}(\sigma)$.

Let us now focus on the function $\mathbf{tr}^{in}_{hp}$. A function application $\mathbf{tr}^{in}_{hp}(s\sigma)$—here the superscript in suggests that we are currently *inside* the hot path hp so that $\mathbf{tr}^{in}_{hp}(s\sigma)$ could either exit from the unfolding of hp or advance with the unfolding of hp—carries on the unfolding of hp as a trace in P_{hp} when the current state s is such that:

(i) $s = \langle \rho, C_i \rangle$, where $i \in [1, n-1]$, meaning that the command C_i is strictly inside hp, i.e., C_i is different from the first command C_0 and the last command C_n of hp;
(ii) the guarded conditional $guard\ E_{a_i}$ is satisfied in the store ρ of the state $s = \langle \rho, C_i \rangle$, that is, $\alpha_{store}(\{\rho\}) \leq a_i$.

If one of these two conditions does not hold then the trace transformation $\mathbf{tr}^{in}_{hp}(\langle \rho, C_i \rangle \sigma)$, after a suitable unfolding step for $\langle \rho, C_i \rangle$, jumps back to the "outside of hp" modality by progressing with $\mathbf{tr}^{out}_{hp}(\sigma)$.

Example 6.3. Consider the transform P_{hp} of Example 5.3 for the program P in Example 2.1 w.r.t. the hot path $hp = \langle \top, C_1, \top, C_2, \top, C_3^c \rangle$. In particular, we refer to the notation H_i, H_i^c used to denote the commands in the stitch of hp into P. Consider the following trace fragment $\tau \in \mathrm{Trace}_P$:

$$\tau = \langle [x/3], C_0 \rangle \langle [x/0], C_1 \rangle \langle [x/0], C_2 \rangle \langle [x/1], C_3^c \rangle \langle [x/1], C_1 \rangle \langle [x/1], C_2 \rangle \langle [x/2], C_3^c \rangle$$
$$\langle [x/2], C_1 \rangle \langle [x/2], C_2 \rangle \langle [x/3], C_3 \rangle \langle [x/3], C_4 \rangle$$

Then, we have that the dynamic transformation $\mathbf{tr}^{out}_{hp}(\tau)$ acts as follows:

$$\begin{aligned}
\mathbf{tr}^{out}_{hp}(\tau) &= \langle [x/3], C_0 \rangle \mathbf{tr}^{out}_{hp}(\tau_{1^{\rightarrow}}) = \langle [x/3], C_0 \rangle \langle [x/0], H_0 \rangle \langle [x/0], H_1 \rangle \mathbf{tr}^{in}_{hp}(\tau_{2^{\rightarrow}}) \\
\mathbf{tr}^{in}_{hp}(\tau_{2^{\rightarrow}}) &= \langle [x/0], H_2 \rangle \langle [x/0], H_3 \rangle \mathbf{tr}^{in}_{hp}(\tau_{3^{\rightarrow}}) \\
\mathbf{tr}^{in}_{hp}(\tau_{3^{\rightarrow}}) &= \langle [x/1], H_4 \rangle \langle [x/1], H_5 \rangle \mathbf{tr}^{in}_{hp}(\tau_{4^{\rightarrow}}) \\
&\ \ \dots \\
\mathbf{tr}^{in}_{hp}(\tau_{9^{\rightarrow}}) &= \mathbf{tr}^{in}_{hp}(\langle [x/3], C_3 \rangle \langle [x/3], C_4 \rangle) = \langle [x/3], H_4 \rangle \langle [x/3], H_5^c \rangle \mathbf{tr}^{out}_{hp}(\langle [x/3], C_4 \rangle) \\
&= \langle [x/3], H_4 \rangle \langle [x/3], H_5^c \rangle \langle [x/3], C_4 \rangle \mathbf{tr}^{out}_{hp}(\epsilon) \\
&= \langle [x/3], H_4 \rangle \langle [x/3], H_5^c \rangle \langle [x/3], C_4 \rangle
\end{aligned}$$

Summing up, using the colors in the flow graph of P_{hp} in Fig. 3 and representing traces as sequences of commands only, we have that:

$$\tau \equiv \boxed{C_0} \to \boxed{C_1} \to \boxed{C_2} \to \boxed{C_3^c} \to \boxed{C_1} \to \boxed{C_2} \to \boxed{C_3^c} \to \boxed{C_1} \to \boxed{C_2} \to \boxed{C_3} \to \boxed{C_4}$$

$$\mathbf{tr}^{out}_{hp}(\tau) \equiv \boxed{C_0} \to \boxed{H_0} \to \boxed{H_1} \to \boxed{H_2} \to \boxed{H_3} \to \boxed{H_4} \to \boxed{H_5} \to \boxed{H_0} \to \boxed{H_1} \to \boxed{H_2} \to$$
$$\to \boxed{H_3} \to \boxed{H_4} \to \boxed{H_5} \to \boxed{H_0} \to \boxed{H_1} \to \boxed{H_2} \to \boxed{H_3} \to \boxed{H_4} \to \boxed{H_5^c} \to \boxed{C_4}$$

$hp = \langle a_0, C_0, ..., a_n, C_n\rangle$ is a given hot path

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

$$\mathbf{tr}^{out}_{hp}(s\sigma) \triangleq \begin{cases} \langle \rho, L_0 : guard\ E_{a_0} \to \ell_0\rangle\langle \rho, \ell_0 : act(C_0) \to \mathbb{1}_1\rangle\, \mathbf{tr}^{in}_{hp}(\sigma) & \\ \qquad\qquad \text{if } s = \langle \rho, C_0\rangle,\ \alpha_{store}(\{\rho\}) \leq a_0 & \\ \langle \rho, L_0 : \neg guard\ E_{a_0} \to \overline{L_0}\rangle\langle \rho, \overline{L_0} : act(C_0) \to L_1\rangle\, \mathbf{tr}^{out}_{hp}(\sigma) & \\ \qquad\qquad \text{if } s = \langle \rho, C_0\rangle,\ \alpha_{store}(\{\rho\}) \not\leq a_0 & \\ \langle \rho, L_0 : guard\ E_{a_0} \to \ell_0\rangle\langle \rho, \ell_0 : \neg act(C_0) \to L^c_1\rangle\, \mathbf{tr}^{out}_{hp}(\sigma) & \\ \qquad\qquad \text{if } s = \langle \rho, cmpl(C_0)\rangle,\ \alpha_{store}(\{\rho\}) \leq a_0 & \\ \langle \rho, L_0 : \neg guard\ E_{a_0} \to \overline{L_0}\rangle\langle \rho, \overline{L_0} : \neg act(C_0) \to L^c_1\rangle\, \mathbf{tr}^{out}_{hp}(\sigma) & \\ \qquad\qquad \text{if } s = \langle \rho, cmpl(C_0)\rangle,\ \alpha_{store}(\{\rho\}) \not\leq a_0 & \\ s \cdot \mathbf{tr}^{out}_{hp}(\sigma) & \text{otherwise} \end{cases}$$

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

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

Fig. 4. Definitions of $\mathbf{tr}^{out}_{hp}$ and $\mathbf{tr}^{in}_{hp}$.

where red boxes denote commands of τ and $\mathbf{tr}^{out}_{hp}(\tau)$ outside of the hot path hp, black boxes with red commands denote commands of τ inside hp, while black boxes with blue commands denote commands of $\mathbf{tr}^{out}_{hp}(\tau)$ in $stitch_P(hp)$. Hence, $\mathbf{tr}^{out}_{hp}(\tau)$ carries out the unfolding of the hot path hp for the execution trace τ of P, and therefore provides an execution trace of the transformed program P_{hp}. □

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

LEMMA 6.4. $\mathbf{tr}^{out}_{hp}$ *is well-defined and for any* $\sigma \in \mathrm{Trace}_P$, $sc(\mathbf{tr}^{out}_{hp}(\sigma)) = sc(\sigma)$.

PROOF. We first show that: (1) $\mathbf{tr}^{out}_{hp}$ is well-defined, i.e., for any $\sigma \in \mathrm{Trace}_P$, $\mathbf{tr}^{out}_{hp}(\sigma) \in \mathrm{Trace}_{P_{hp}}$, and (2) for any $\sigma \in \mathrm{Trace}_P$, if $cmd(\sigma_0) \notin \{C_0, cmpl(C_0)\}$ then

$hp = \langle a_0, C_0, ..., a_n, C_n\rangle$ is a given hot path

$$\mathbf{rtr}_{hp}(\epsilon) \triangleq \epsilon$$

$$\mathbf{rtr}_{hp}(s\sigma) \triangleq \begin{cases} \langle \mathit{store}(s), C_i\rangle & \text{if } \sigma = \epsilon,\ \mathit{act}(s) \in \{\mathit{guard}\ E_{a_i}, \neg\mathit{guard}\ E_{a_i}\},\ i \in [1, n] \\ \mathbf{rtr}_{hp}(\sigma) & \text{if } \sigma \neq \epsilon,\ \mathit{act}(s) \in \{\mathit{guard}\ E_{a_i}, \neg\mathit{guard}\ E_{a_i}\},\ i \in [1, n] \\ \langle \rho, C_0\rangle\mathbf{rtr}_{hp}(\sigma) & \text{if } s = \langle \rho, \overline{L_0} : \mathit{act}(C_0) \to L_1\rangle \\ \langle \rho, C_0^c\rangle\mathbf{rtr}_{hp}(\sigma) & \text{if } s = \langle \rho, \overline{L_0} : \neg\mathit{act}(C_0) \to L_1^c\rangle \\ \langle \rho, C_i\rangle\mathbf{rtr}_{hp}(\sigma) & \text{if } s = \langle \rho, \ell_i : \mathit{act}(C_i) \to \mathbb{l}_{i+1}\rangle,\ i \in [1, n-1] \\ \langle \rho, C_i^c\rangle\mathbf{rtr}_{hp}(\sigma) & \text{if } s = \langle \rho, \ell_i : \neg\mathit{act}(C_i) \to L^c_{\mathit{next}(i)}\rangle,\ i \in [1, n] \\ \langle \rho, C_n\rangle\mathbf{rtr}_{hp}(\sigma) & \text{if } s = \langle \rho, \ell_n : \mathit{act}(C_n) \to L_0\rangle \\ s \cdot \mathbf{rtr}_{hp}(\sigma) & \text{otherwise} \end{cases}$$

Fig. 5. Definition of rtr_{hp}.

$\mathbf{tr}^{in}_{hp}(\sigma) \in \mathrm{Trace}_{P_{hp}}$. In order to prove these two points, it is enough an easy induction on the length of the execution trace σ and to observe that:

(i) for the first four clauses that define $\mathbf{tr}^{out}_{hp}(s\sigma)$ in Fig. 4 we have that $\mathbf{tr}^{out}_{hp}(s\sigma) = s's''\mathbf{tr}^{out}_{hp}(\sigma)$ or $\mathbf{tr}^{out}_{hp}(s\sigma) = s's''\mathbf{tr}^{in}_{hp}(\sigma)$, where s' is a guard command of P_{hp} and $s's''$ is in turn a legal sub-execution trace of P_{hp};
(ii) for the last clause that defines $\mathbf{tr}^{out}_{hp}(s\sigma)$ in Fig. 4 we have that $\mathit{cmd}(s) \notin \{C_0, \mathit{cmpl}(C_0)\}$, hence s is a legal state in P_{hp} and, in turn, $\mathbf{tr}^{out}_{hp}(s\sigma) = s \cdot \mathbf{tr}^{out}_{hp}(\sigma)$ is a trace of P_{hp};
(iii) for the clauses 1, 2 and 4 that define $\mathbf{tr}^{in}_{hp}(s\sigma)$ in Fig. 4 we have that $\mathbf{tr}^{in}_{hp}(s\sigma) = s's''\mathbf{tr}^{in}_{hp}(\sigma)$ or $\mathbf{tr}^{in}_{hp}(s\sigma) = s's''\mathbf{tr}^{out}_{hp}(\sigma)$, 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};
(iv) for the clauses 3 and 5 that define $\mathbf{tr}^{in}_{hp}(s\sigma)$ in Fig. 4 we have that $\mathbf{tr}^{in}_{hp}(s\sigma) = s's\mathbf{tr}^{in}_{hp}(\sigma)$ where s' is a guard command and $s's$ turns out to be a legal sub-execution trace of P_{hp};
(v) for the last clause that defines $\mathbf{tr}^{in}_{hp}(s\sigma)$ in Fig. 4 we have that $\mathit{cmd}(s) \notin \{C_i, \mathit{cmpl}(C_i) \mid i \in [1, n]\}$; by hypothesis, $\mathit{cmd}(s) \notin \{C_0, \mathit{cmpl}(C_0)\}$, so that $\mathit{cmd}(s) \notin \{C_i, \mathit{cmpl}(C_i) \mid i \in [0, n]\}$, hence s is a legal state in P_{hp} and in turn $\mathbf{tr}^{in}_{hp}(s\sigma) = s \cdot \mathbf{tr}^{out}_{hp}(\sigma)$ is a trace of P_{hp};
(vi) $\mathbf{tr}^{in}_{hp}(s\sigma)$ is never recursively called by a function application $\mathbf{tr}^{out}_{hp}(s_0 s\sigma)$ when $\mathit{cmd}(s) \in \{C_0, \mathit{cmpl}(C_0)\}$.

Then, it is immediate to check from the definitions in Fig. 4 that if $\mathbf{tr}^{out}_{hp}(s\sigma) = s's''\tau$ then $\mathit{store}(s) = \mathit{store}(s') = \mathit{store}(s'')$. Therefore, for any $\sigma \in \mathrm{Trace}_P$, we obtain that $\mathit{sc}(\mathbf{tr}^{out}_{hp}(\sigma)) = \mathit{sc}(\sigma)$. □

Vice versa, it is a simpler task to define a reverse transformation function $\mathbf{rtr}_{hp}$ that "decompiles" an execution trace σ of P_{hp} into an execution trace of P by removing guarded commands in σ, as generated by the hot path hp, and by mapping the relabeled commands of hp in σ back to their corresponding source commands of hp. This function $\mathbf{rtr}_{hp} : \mathrm{Trace}_{P_{hp}} \to \mathrm{Trace}_{P}$ is correctly defined by the clauses in Fig. 5 and it preserves store change sequences.

LEMMA 6.5. $\mathbf{rtr}_{hp}$ *is well-defined and for any* $\sigma \in \mathrm{Trace}_{P_{hp}}$*,* $sc(\mathbf{rtr}_{hp}(\sigma)) = \mathbf{rtr}_{hp}(\sigma)$*.*

PROOF. We show that $\mathbf{rtr}_{hp}$ is well-defined, i.e., for any $\sigma \in \mathrm{Trace}_{P_{hp}}$, $\mathbf{rtr}_{hp}(\sigma) \in \mathrm{Trace}_{P}$. This follows by an easy induction on the length of the execution trace σ by observing that:

(i) the first clause that defines $\mathbf{rtr}_{hp}(s\sigma)$ in Fig. 5 is an extremal base case where $s\sigma = s$ and the command action of s is a guard command $guard\ E_{a_i}$ (or its complement); in this case, we simply retain the store of s and pick the command C_i of P.
(ii) the clause 2 of $\mathbf{rtr}_{hp}(s\sigma)$ in Fig. 5 simply removes the states whose commands are some $guard\ E_{a_i}$; since $guard\ E_{a_i}$ does not alter stores, this removal preserves the sequence of store changes.
(iii) the clauses 3-7 of $\mathbf{rtr}_{hp}(s\sigma)$ in Fig. 5 map a state s of P_{hp} whose command H_i is a relabeled action $act(C_i)$ or $\neg act(C_i)$ of the hot path hp to a corresponding state of P that has the same $store(s)$ and whose command is: C_i for $act(C_i)$ and C_i^c for $\neg act(C_i)$; here, we observe that since guards in σ are removed, by induction, these definitions allow us to obtain that $s\sigma$ is mapped to a legal trace of P that does not alter the sequence of store changes.
(iv) the clause 8 of $\mathbf{rtr}_{hp}(s\sigma)$ in Fig. 5 states that if s is already a state of P then it is left unchanged.

Hence, the above points also show that the sequence of store changes is not affected by $\mathbf{rtr}_{hp}$, i.e., for any $\sigma \in \mathrm{Trace}_{P_{hp}}$, $sc(\mathbf{rtr}_{hp}(\sigma)) = sc(\sigma)$. □

Example 6.6. We carry on Example 6.3 by considering the following trace fragment $\sigma \in \mathrm{Trace}_{P_{hp}}$, where the transformed program P_{hp} is in Example 5.3:

$$\begin{aligned}\sigma = \langle [x/2], H_4\rangle\langle [x/2], H_5\rangle\langle [x/2], H_0\rangle\langle [x/2], H_1\rangle\langle [x/2], H_2\rangle\langle [x/2], H_3\rangle\langle [x/3], H_4\rangle \\ \langle [x/3], H_5^c\rangle\langle [x/3], C_4\rangle\langle [x/6], C_1\rangle\end{aligned}$$

Here, the decompilation of σ back into an execution trace of P through $\mathbf{rtr}_{hp}$ yields:

$$\begin{aligned}\mathbf{rtr}_{hp}(\sigma) &= \mathbf{rtr}_{hp}(\sigma_{1^{\to}}) = \langle [x/2], C_3^c\rangle \mathbf{rtr}_{hp}(\sigma_{2^{\to}}) = \langle [x/2], C_3^c\rangle \mathbf{rtr}_{hp}(\sigma_{3^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle \mathbf{rtr}_{hp}(\sigma_{4^{\to}}) = \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle \mathbf{rtr}_{hp}(\sigma_{5^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle \mathbf{rtr}_{hp}(\sigma_{6^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle \mathbf{rtr}_{hp}(\sigma_{7^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle\langle [x/3], C_3\rangle \mathbf{rtr}_{hp}(\sigma_{8^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle\langle [x/3], C_3\rangle\langle [x/3], C_4\rangle \mathbf{rtr}_{hp}(\sigma_{9^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle\langle [x/3], C_3\rangle \mathbf{rtr}_{hp}(\sigma_{8^{\to}}) \\ &= \langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle\langle [x/3], C_3\rangle\langle [x/3], C_4\rangle\langle [x/6], C_1\rangle\end{aligned}$$

Indeed, $\langle [x/2], C_3^c\rangle\langle [x/2], C_1\rangle\langle [x/2], C_2\rangle\langle [x/3], C_3\rangle\langle [x/3], C_4\rangle\langle [x/6], C_1\rangle$ is a well-defined execution trace of P. □

We are now in the position to prove Theorem 6.2.

PROOF OF THEOREM 6.2. With an abuse of notation for $\mathbf{rtr}_{hp}$, let us define two functions $\mathbf{tr}_{hp} : \wp(\mathrm{Trace}_{P}) \to \wp(\mathrm{Trace}_{P_{hp}})$ and $\mathbf{rtr}_{hp} : \wp(\mathrm{Trace}_{P_{hp}}) \to \wp(\mathrm{Trace}_{P})$ which are the collecting versions of $\mathbf{tr}_{hp}^{out}$ and $\mathbf{rtr}_{hp}$, that is, $\mathbf{tr}_{hp}(T) \triangleq \{\mathbf{tr}_{hp}^{out}(\sigma) \mid \sigma \in T\}$ and

$\mathbf{rtr}_{hp}(T) \triangleq \{\mathbf{rtr}_{hp}(\sigma) \mid \sigma \in T\}$. As consequences of the above lemmata, we have the following properties.

(A) $\alpha_{sc} \circ \mathbf{tr}_{hp} = \alpha_{sc}$: by Lemma 6.4.
(B) $\mathbf{tr}_{hp}(\mathbf{T}[\![P]\!]) \subseteq \mathbf{T}[\![P_{hp}]\!]$: because, by Lemma 6.4, $\mathbf{tr}_{hp}^{out}$ is well-defined.
(C) $\alpha_{sc} \circ \mathbf{rtr}_{hp} = \alpha_{sc}$: by Lemma 6.5.
(D) $\mathbf{rtr}_{hp}(\mathbf{T}[\![P_{hp}]\!]) \subseteq \mathbf{T}[\![P]\!]$: because, by Lemma 6.5, $\mathbf{rtr}_{hp}$ is well-defined.

We therefore obtain:

$$\begin{aligned}
\alpha_{sc}(\mathbf{T}[\![P]\!]) = &\quad \text{[By point (A)]}\\
\alpha_{sc}(\mathbf{tr}_{hp}(\mathbf{T}[\![P]\!])) \subseteq &\quad \text{[By point (B)]}\\
\alpha_{sc}(\mathbf{T}[\![P_{hp}]\!]) = &\quad \text{[By point (C)]}\\
\alpha_{sc}(\mathbf{rtr}_{hp}(\mathbf{T}[\![P_{hp}]\!])) \subseteq &\quad \text{[By point (D)]}\\
\alpha_{sc}(\mathbf{T}[\![P]\!]) &
\end{aligned}$$

and this closes the proof. □

6.2. Correctness of Hot Path Optimizations

Guarded hot paths are a key feature of our tracing compilation model and are meant to be dynamically recorded by a hot path monitor. An abstract guard for a command C of some stitched hot path $stitch_P(hp)$ encodes a property of program stores which is represented as an element of an abstract domain $\mathrm{Store}^\sharp$ and is guaranteed to hold at the entry of C. This information on program stores, as encapsulated by the abstract guards in $stitch_P(hp)$, can then be used in hot path optimizations, namely, to optimize the commands in hp.

We follow a modular approach for proving the correctness of hot path optimizations. A hot path optimization O should optimize P along some hot path hp of P, by relying on the abstract store information recorded in hp, while leaving unchanged the commands outside of hp. Hence, in our framework, fixed $P \in \mathrm{Program}$, an optimization O is defined to be a program transform of the commands in $stitch_P(hp)$, that is,

$$O : \{stitch_P(hp) \mid hp \in \alpha_{hot}^N(\mathrm{Trace}_P)\} \to \mathrm{Program}$$

where $\mathrm{Program}$ may allow new optimized expressions and/or actions introduced by O, as it will be the case of type-specific additions $+_{\mathrm{Type}}$ in the type specialization optimization described in Section 7. Let $P_{\neg hp} \triangleq extr_{hp}(P) \smallsetminus stitch_P(hp)$ denote the commands outside of the stitched hot path. Then, the corresponding full optimization O_{full} of the whole program P w.r.t. the hot path hp should extract and simultaneously optimize hp, namely, this is defined by

$$O_{full}(P, hp) \triangleq P_{\neg hp} \cup O(stitch_P(hp))$$

where $O_{full}(P, hp)$ is required to be a well-formed program, i.e., $O_{full}(P, hp) \in \mathrm{Program}$. This full optimization $O_{full}(P, hp)$ has to be proved correct w.r.t. some observational abstraction $\alpha_o : \wp(\mathrm{Trace}_P) \to A$ of program traces, which is assumed to be more abstract than the store changes abstraction α_{sc} (cf. Section 6). Then, this full optimization is correct for α_o when:

$$\alpha_o(\mathbf{T}[\![O_{full}(P, hp)]\!]) = \alpha_o(\mathbf{T}[\![P]\!]).$$

Since Theorem 6.2 ensures that the unoptimized trace extraction transform is already correct for the store changes abstraction α_{sc}, which is more precise than α_o, the intuition is that in order to prove the correctness of O_{full} w.r.t. α_o, it is enough to focus on the correctness of the optimization O along the stitched hot path $stitch_P(hp)$. This therefore leads to the following definition of correctness for a hot path optimization.

Definition 6.7 (***Correctness of hot path optimization***). O is *correct* for the observational abstraction α_o if for any $P \in \mathrm{Program}$ and for any $hp \in \alpha^N_{hot}(\mathrm{Trace}_P)$, $\alpha_o(\mathbf{T}[\![O(stitch_P(hp))]\!]) = \alpha_o(\mathbf{T}[\![stitch_P(hp)]\!])$. □

In order to prove that this correctness of a hot path optimization implies the correctness of the corresponding full optimization, we define two functions

$$to : \mathrm{Trace}_{stitch_P(hp)} \to \mathrm{Trace}_{O(stitch_P(hp))} \qquad tdo : \mathrm{Trace}_{O(stitch_P(hp))} \to \mathrm{Trace}_{stitch_P(hp)}$$

which must be well-defined, i.e. they have to map well-formed traces into well-formed traces, and, intuitively, encode the effect of optimizing (function to) and de-optimizing (function tdo) execution traces along a stitched hot path. Since $\mathrm{Trace}_{stitch_P(hp)} \subseteq \mathrm{Trace}_{extr_{hp}(P)}$ and $\mathrm{Trace}_{O(stitch_P(hp))} \subseteq \mathrm{Trace}_{O_{full}(P,hp)}$, we then extend to and tdo to two functions

$$to_{full} : \mathrm{Trace}_{extr_{hp}(P)} \to \mathrm{Trace}_{O_{full}(P,hp)} \qquad tdo_{full} : \mathrm{Trace}_{O_{full}(P,hp)} \to \mathrm{Trace}_{extr_{hp}(P)}$$

which simply apply to and tdo to maximal subtraces, respectively, in $\mathrm{Trace}_{stitch_P(hp)}$ and $\mathrm{Trace}_{O(stitch_P(hp))}$, while leaving unchanged the remaining states. Let us formalize this idea. If $\sigma \in \mathrm{Trace}_{extr_{hp}(P)}$ is nonempty and, for some $k \in [0, |\sigma|)$, $cmd(\sigma_k) \in stitch_P(hp)$ then $\sigma_{[k,n_{st}]}$ denotes the maximal subtrace of σ beginning at σ_k which belongs to $\mathrm{Trace}_{stitch_P(hp)}$, that is, the index $n_{st} \geq k$ is such that: (1) $cmd(\sigma_{n_{st}}) \in stitch_P(hp)$, (2) if $n_{st} < |\sigma| - 1$ then $cmd(\sigma_{n_{st}+1}) \notin stitch_P(hp)$, (3) for any $j \in [k, n_{st}]$, $cmd(\sigma_j) \in stitch_P(hp)$. Analogously, if $\tau \in \mathrm{Trace}_{O_{full}(P,hp)}$ is nonempty and $cmd(\tau_k) \in O(stitch_P(hp))$ then $\tau_{[k,n_{st}]}$ denotes the maximal subtrace of τ beginning at τ_k which belongs to $\mathrm{Trace}_{O(stitch_P(hp))}$. Then, the formal definition of to_{full} goes as follows:

$$to_{full}(\sigma) \triangleq \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \sigma_0 \; to_{full}(\sigma_{1^\rightarrow}) & \text{if } \sigma \neq \epsilon,\; cmd(\sigma_0) \notin stitch_P(hp) \\ to(\sigma_{[0,n_{st}]}) \; to_{full}(\sigma_{(n_{st}+1)^\rightarrow}) & \text{if } \sigma \neq \epsilon,\; cmd(\sigma_0) \in stitch_P(hp) \end{cases}$$

and analogously for tdo_{full}. Since to and tdo are supposed to be well-defined, it turns out that to_{full} and tdo_{full} are well-defined once we make the weak and reasonable assumption that to and tdo do not modify the entry (which is always L_0) and exit labels of the stitched hot path. This assumption, e.g., for to can be formalized as follows: if $\sigma \in \mathrm{Trace}_{stitch_P(hp)}$ and $to(\sigma) = \tau$ then (i) if $lbl(\sigma_0) = L_0$ then $lbl(\tau_0) = L_0$; (ii) if $suc(\sigma_{|\sigma|-1}) = L' \notin labels(P)$ then $suc(\tau_{|\tau|-1}) = L'$. In the following, to_{full} and tdo_{full} are also used to denote their corresponding collecting functions defined on sets of traces.

LEMMA 6.8. *Assume that $\alpha_o \circ to_{full} = \alpha_o = \alpha_o \circ tdo_{full}$. If O is correct for α_o then O_{full} is correct for α_o.*

PROOF. We have that:

$$\begin{aligned} \alpha_o(\mathbf{T}[\![O_{full}(P,hp)]\!]) = & \quad [\text{By } \alpha_o \circ tdo_{full} = \alpha_o] \\ \alpha_o(tdo_{full}(\mathbf{T}[\![O_{full}(P,hp)]\!])) \subseteq & \quad [\text{Since } tdo_{full} \text{ is well-defined}] \\ \alpha_o(\mathbf{T}[\![extr_{hp}(P)]\!]) = & \quad [\text{By } \alpha_o \circ to_{full} = \alpha_o] \\ \alpha_o(to_{full}(\mathbf{T}[\![extr_{hp}(P)]\!])) \subseteq & \quad [\text{Since } to_{full} \text{ is well-defined}] \\ \alpha_o(\mathbf{T}[\![O_{full}(P,hp)]\!]) & \end{aligned}$$

Thus, $\alpha_o(\mathbf{T}[\![O_{full}(P, hp)]\!]) = \alpha_o(\mathbf{T}[\![extr_{hp}(P)]\!])$. By Theorem 6.2, $\alpha_{sc}(\mathbf{T}[\![extr_{hp}(P)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$, so that, since α_{sc} is more precise than α_o, $\alpha_o(\mathbf{T}[\![extr_{hp}(P)]\!]) = \alpha_o(\mathbf{T}[\![P]\!])$, and, in turn, $\alpha_o(\mathbf{T}[\![O_{full}(P, hp)]\!]) = \alpha_o(\mathbf{T}[\![P]\!])$. $\square$

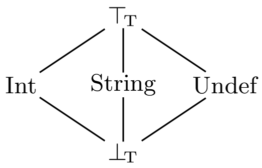
We will see in Sections 7 and 8 two significant examples of hot path optimizations, namely, type specialization and constant folding.

7. TYPE SPECIALIZATION

One key optimization for dynamic languages like JavaScript and PHP is type specialization, that is, the use of type-specific primitives in place of generic untyped operations whose runtime execution can be costly. As a paradigmatic example, a generic addition operation could be defined on more than one type, so that the runtime 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 runtime semantics in the ECMA-262 standard [Ecma International 2015, Section 12.7.3.1]) 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. In this section, we show that type specialization can be viewed as a particular hot path optimization which can be proved correct according to our definition in Section 6.2.

7.1. Type Abstraction

Let us recall that the set of type names is $\mathrm{Types} = \{\top_{\mathrm{T}}, \mathrm{Int}, \mathrm{String}, \mathrm{Undef}, \bot_{\mathrm{T}}\}$, which can be viewed as the following finite lattice $\langle \mathrm{Types}, \leq_{\mathrm{t}} \rangle$:



The abstraction $\alpha_{type} : \wp(\mathrm{Value_u}) \to \mathrm{Types}$ and concretization $\gamma_{type} : \mathrm{Types} \to \wp(\mathrm{Value_u})$ functions are defined as follows:

$$\alpha_{type}(S) \triangleq \begin{cases} \bot_{\mathrm{T}} & \text{if } S = \varnothing \\ \mathrm{Int} & \text{if } \varnothing \neq S \subseteq \mathbb{Z} \\ \mathrm{String} & \text{if } \varnothing \neq S \subseteq \mathrm{Char}^* \\ \mathrm{Undef} & \text{if } \varnothing \neq S = \{\mathit{undef}\} \\ \top_{\mathrm{T}} & \text{otherwise} \end{cases} \qquad \gamma_{type}(T) \triangleq \begin{cases} \varnothing & \text{if } T = \bot_{\mathrm{T}} \\ \mathbb{Z} & \text{if } T = \mathrm{Int} \\ \mathrm{Char}^* & \text{if } T = \mathrm{String} \\ \{\mathit{undef}\} & \text{if } T = \mathrm{Undef} \\ \mathrm{Value_u} & \text{if } T = \top_{\mathrm{T}} \end{cases}$$

Thus, $\alpha_{type}(S)$ provides the smallest type in $\langle \mathrm{Types}, \leq_{\mathrm{t}} \rangle$ for a set S of values. In particular, given $v \in \mathrm{Value_u}$, $\alpha_{type}(\{v\})$ coincides with $\mathit{type}(v)$. Following the approach described in Section 3.2.1, we then consider a simple nonrelational store abstraction for types

$$\mathrm{Store^t} \triangleq \langle \mathrm{Var} \to \mathrm{Types}, \dot{\leq}_{\mathrm{t}} \rangle$$

where $\dot{\leq}_{\mathrm{t}}$ is the standard pointwise lifting of $\leq_{\mathrm{t}}$, so that $\lambda x.\bot_{\mathrm{T}}$ and $\lambda x.\top_{\mathrm{T}}$ are, respectively, the bottom and top abstract stores in $\mathrm{Store^t}$. The abstraction and concretization maps $\alpha_{store} : \wp(\mathrm{Store}) \to \mathrm{Store^t}$ and $\gamma_{store} : \mathrm{Store^t} \to \wp(\mathrm{Store})$ are defined as a straight instantiation of the definitions in Section 3.2.1.

The abstract type semantics $\mathbf{E}^t : \mathrm{Exp} \to \mathrm{Store}^{\mathrm{t}} \to \mathrm{Types}$ of expressions is defined as the best correct approximation of the concrete collecting semantics $\mathbf{E} : \mathrm{Exp} \to \wp(\mathrm{Store}) \to \wp(\mathrm{Value})$ on the type abstractions $\mathrm{Store}^{\mathrm{t}}$ and Types, i.e.,

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

Hence, this definition leads to the following equalities:

$$\mathbf{E}^t[\![v]\!]\rho^t = \mathit{type}(v)$$
$$\mathbf{E}^t[\![x]\!]\rho^t = \rho^t(x)$$
$$\mathbf{E}^t[\![E_1 + E_2]\!]\rho^t = \begin{cases} \bot_{\mathrm{T}} & \text{if } \exists i.\, \mathbf{E}^t[\![E_i]\!]\rho^t = \bot_{\mathrm{T}} \\ \mathbf{E}^t[\![E_1]\!]\rho^t & \text{else if } \mathbf{E}^t[\![E_1]\!]\rho^t = \mathbf{E}^t[\![E_2]\!]\rho^t \in \{\mathrm{Int}, \mathrm{String}\} \\ \mathrm{Undef} & \text{else if } \forall i.\, \mathbf{E}^t[\![E_i]\!]\rho^t < \top_{\mathrm{T}} \\ \top_{\mathrm{T}} & \text{otherwise} \end{cases}$$

For instance, we have that:

$$\mathbf{E}^t[\![x + y]\!][x/\,\mathrm{String}, y/\bot_{\mathrm{T}}] = \alpha_{type}(\mathbf{E}[\![x + y]\!]\varnothing) = \alpha_{type}(\varnothing) = \bot_{\mathrm{T}}$$

$$\mathbf{E}^t[\![x + y]\!][x/\,\mathrm{String}, y/\,\mathrm{String}] = \alpha_{type}(\mathbf{E}[\![x + y]\!]\{\rho \mid \rho(x), \rho(y) \in \mathrm{Char}^*\}) = \\ = \alpha_{type}(\mathrm{Char}^*) = \mathrm{String},$$

$$\mathbf{E}^t[\![x + y]\!][x/\,\mathrm{Int}, y/\,\mathrm{String}] = \alpha_{type}(\mathbf{E}[\![x + y]\!]\{\rho \mid \rho(x) \in \mathbb{Z}, \rho(y) \in \mathrm{Char}^*\}) = \\ \alpha_{type}(\{\textit{\textbf{undef}}\}) = \mathrm{Undef},$$

$$\mathbf{E}^t[\![x + y]\!][x/\,\mathrm{Int}, y/\top_{\mathrm{T}}] = \alpha_{type}(\mathbf{E}[\![x + y]\!]\{\rho \mid \rho(x) \in \mathbb{Z}, \rho(y) \in \mathrm{Value}_{\mathrm{u}}\}) = \\ \alpha_{type}(\mathbb{Z} \cup \{\textit{\textbf{undef}}\}) = \top_{\mathrm{T}}$$

Being defined as best correct approximation, it turns out that the abstract type semantics $\mathbf{E}^t$ of expressions is correct by definition.

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

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

$$\textbf{guard}\; x_0 : T_0, \ldots, x_n : T_n \in \mathrm{BExp}$$

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

$$\mathbf{A}[\![\textbf{guard}\; x_0 : T_0, ..., x_n : T_n]\!]\rho \triangleq \begin{cases} \rho & \text{if } \rho \in \gamma_{store}([x_i/T_i \mid x_i \in \mathrm{Var}]) \\ \bot & \text{otherwise} \end{cases}$$
$$= \begin{cases} \rho & \text{if } \forall i.\, \rho(x_i) \in \gamma_{type}(T_i) \\ \bot & \exists i.\, \rho(x_i) \not\in \gamma_{type}(T_i) \end{cases}$$

For example, we have that:

$$\mathbf{A}[\![\textbf{guard}\; x : \mathrm{String}, y : \mathrm{String}]\!][x/\texttt{foo}, y/\texttt{bar}] = [x/\texttt{foo}, y/\texttt{bar}],$$
$$\mathbf{A}[\![\textbf{guard}\; x : \mathrm{String}, y : \top_{\mathrm{T}}]\!][x/\texttt{foo}, y/3] = [x/\texttt{foo}, y/3],$$
$$\mathbf{A}[\![\textbf{guard}\; x : \mathrm{String}, y : \top_{\mathrm{T}}]\!][x/1, y/3] = \bot,$$
$$\mathbf{A}[\![\textbf{guard}\; x : \mathrm{String}, y : \mathrm{Undef}]\!][x/\texttt{foo}] = [x/\texttt{foo}].$$

7.2. Type Specialization of Hot Paths

Let us consider some hot path $hp = \langle \rho_0^t, C_0, \ldots, \rho_n^t, C_n \rangle \in \alpha_{hot}^N(\mathrm{Trace}_P)$ on the type abstraction $\langle \mathrm{Store}_P^t, \dot{\leq}_{\mathrm{t}} \rangle$, where each ρ_i^t is therefore a type environment for P. Thus, in the transformed program $extr_{hp}(P)$, the stitched hot path $stitch_P(hp)$ contains $n+1$ typed guards, that, for any $i \in [0,n]$, we simply denote as $guard\ \rho_i^t$. 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, while one could also consider an analogous type specialization of Boolean comparisons in conditional commands. 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 $stitch_P(hp)$. Note that if $C \in stitch_P(hp)$ 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 $L' \in \{\mathbb{l}_{i+1}, L_0\}$. Let $\mathbb{C}^t$ denote the extended set of commands which includes type specific additions $+_{\mathrm{Int}}$ and $+_{\mathrm{String}}$ and, in turn, let $\mathrm{Program}^t$ denote the possibly type-specialized programs with commands ranging in $\mathbb{C}^t$. The semantic function $\mathbf{E}$ for expressions is then updated to type specific additions as follows:

$$\mathbf{E}[\![E_1 +_{\mathrm{Int}} E_2]\!]\rho \triangleq \begin{cases} \mathbf{E}[\![E_1]\!]\rho +_{\mathbb{Z}} \mathbf{E}[\![E_2]\!]\rho & \text{if } \mathit{type}(\mathbf{E}[\![E_i]\!]\rho) = \mathrm{Int} \\ \mathit{undef} & \text{otherwise} \end{cases}$$

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

Given a hot path $hp = \langle \rho_0^t, C_0, \ldots, \rho_n^t, C_n \rangle$, the type specialization function $\mathbf{ts}_{hp} : stitch_P(hp) \to \mathbb{C}^t$ is defined as follows:

$$\begin{aligned} \mathbf{ts}_{hp}(\ell_i : x := E_1 + E_2 \to L') &\triangleq \begin{cases} \ell_i : x := E_1 +_{\mathrm{Int}} E_2 \to L' & \text{if } \mathbf{E}^t[\![E_1 + E_2]\!]\rho_i^t = \mathrm{Int} \\ \ell_i : x := E_1 +_{\mathrm{String}} E_2 \to L' & \text{if } \mathbf{E}^t[\![E_1 + E_2]\!]\rho_i^t = \mathrm{String} \\ \ell_i : x := E_1 + E_2 \to L' & \text{otherwise} \end{cases} \\ \mathbf{ts}_{hp}(C) &\triangleq C \qquad\qquad\qquad \text{if } C \not\equiv \ell_i : x := E_1 + E_2 \to L' \end{aligned}$$

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

$$O^{\mathrm{ts}} : \{ stitch_P(hp) \mid hp \in \alpha_{hot}^N(\mathrm{Trace}_P) \} \to \mathrm{Program}^t$$

simply by

$$O^{\mathrm{ts}}(stitch_P(hp)) \triangleq \{ \mathbf{ts}_{hp}(C) \mid C \in stitch_P(hp) \}.$$

In turn, as described in Section 6.2, this induces the full type specialization optimization

$$O_{\mathit{full}}^{\mathrm{ts}}(P, hp) \triangleq extr_{hp}(P) \smallsetminus stitch_P(hp) \cup O^{\mathrm{ts}}(stitch_P(hp)).$$

$O_{\mathit{full}}^{\mathrm{ts}}(P, hp)$ is also called *typed trace extraction* since it extracts and simultaneously type specializes a typed hot path hp in a program P. The correctness of this program optimization can be proved for the store changes observational abstraction by relying on Lemma 6.8.

THEOREM 7.2 (**CORRECTNESS OF TYPED TRACE EXTRACTION**). *For any typed hot path* $hp \in \alpha_{hot}^N(\mathrm{Trace}_P)$*, we have that* $\alpha_{sc}(\mathbf{T}[\![O_{\mathit{full}}^{\mathrm{ts}}(P, hp)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$.

PROOF. Let $td : \mathrm{Trace}_{O^{\mathrm{ts}}(\mathit{stitch}_P(hp))} \to \mathrm{Trace}_{\mathit{stitch}_P(hp)}$ be the following type despecialization function, where Type is either Int or String:

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

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

Let us explain the first defining clause of $td(s\sigma)$, i.e., $s = \langle \rho, \ell_i : x := E_1 +_{\mathrm{Type}} E_2 \to L' \rangle$ and $\mathit{type}(\mathbf{E}[\![E_1 + E_2]\!]\rho) \neq \mathrm{Type}$. These conditions can never hold in an inductive call of the function td: in fact, when $td(s\sigma)$ is recursively called by $td(s's\sigma)$, we necessarily have that $s' = \langle \rho, \mathbb{1}_i : \mathit{guard}\ \rho_i^t \to \ell_i \rangle$, so that $\rho \in \gamma_{\mathit{store}}(\rho_i^t)$, and, in turn, by Corollary 7.1, $\mathbf{E}[\![E_1 + E_2]\!]\rho \in \mathbf{E}^t[\![E_1 + E_2]\!]\rho^t$, which implies $\mathit{type}(\mathbf{E}[\![E_1 + E_2]\!]\rho) = \mathrm{Type}$, which is a contradiction. Thus, the first defining clause of $td(s\sigma)$ only applies to type specialized traces in $\mathrm{Trace}_{O^{\mathrm{ts}}(\mathit{stitch}_P(hp))}$ whose first state is $s = \langle \rho, \ell_i : x := E_1 +_{\mathrm{Type}} E_2 \to L' \rangle$: in this case, we necessarily have that $\sigma = \epsilon$, because $\mathbf{A}[\![E_1 +_{\mathrm{Type}} E_2]\!]\rho = \mathit{undef}$ so that $\mathbf{S}s = \varnothing$. This clarifies the definition of td in this particular case. Also, observe that in this case, $sc(td(s)) = sc(s)$ trivially holds. In all the remaining cases, it is clear that td maps type specialized traces into legal unspecialized traces of $\mathit{stitch}_P(hp)$ since labels are left unchanged. Moreover, $sc \circ td = sc$ holds, in particular because in the second defining clause of $td(s\sigma)$, the condition $\mathit{type}(\mathbf{E}[\![E_1 + E_2]\!]\rho) = \mathrm{Type}$ guarantees that $\mathbf{E}[\![E_1 + E_2]\!]\rho = \mathbf{E}[\![E_1 +_{\mathrm{Type}} E_2]\!]\rho$.

On the other hand, we define a trace specialization function $sp : \mathrm{Trace}_{\mathit{stitch}_P(hp)} \to \mathrm{Trace}_{O^{\mathrm{ts}}(\mathit{stitch}_P(hp))}$ as follows:

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

$$sp(\langle \mu_0, H_0 \rangle \cdots \langle \mu_k, H_k \rangle) \triangleq \begin{cases} \langle \mu_0, \mathrm{ts}_{hp}(H_0) \rangle & \text{if } \mathrm{ts}_{hp}(H_0) \equiv \ell_i : x := E_1 +_{\mathrm{Type}} E_2 \to L', \\ & \mu_0 \notin \gamma_{\mathit{store}}(\rho_i^t) \\ \langle \mu_0, \mathrm{ts}_{hp}(H_0) \rangle \cdots \langle \mu_k, \mathrm{ts}_{hp}(H_k) \rangle & \text{otherwise} \end{cases}$$

Let us comment on this definition. If $\sigma \in \mathrm{Trace}_{\mathit{stitch}_P(hp)}$ and $\sigma \neq \epsilon$ then it may happen that the first state $\langle \mu_0, H_0 \rangle$ of σ is such that the command H_0 is $\ell_i : x := E_1 + E_2 \to L'$ and, since $\mathbf{E}^t[\![E_1 + E_2]\!]\rho_i^t = \mathrm{Type}$ (Int or String), H_0 is type specialized to $\mathrm{ts}_{hp}(H_0) \equiv \ell_i : x := E_1 +_{\mathrm{Type}} E_2 \to L'$, while the store μ_0 is not approximated by the abstract store ρ_i^t, i.e., $\mu_0 \notin \gamma_{\mathit{store}}(\rho_i^t)$. Thus, in this case, the trace in $O^{\mathrm{ts}}(\mathit{stitch}_P(hp))$ beginning at $\langle \mu_0, \mathrm{ts}_{hp}(H_0) \rangle$ is stuck, because the concrete semantics of addition is $\mathbf{E}[\![E_1 +_{\mathrm{Type}} E_2]\!]\mu_0 = \mathit{undef}$, and in turn $\mathbf{A}[\![x := E_1 +_{\mathrm{Type}} E_2]\!]\mu_0 = \bot$, so that we necessarily have to define $sp(\sigma) = \langle \mu_0, \mathrm{ts}_{hp}(H_0) \rangle$. Otherwise, $sp(\sigma)$ simply type specializes through ts_{hp} all the commands (actually, addition expressions) occurring in σ. Here, it turns out that sp is well-defined, i.e. $sp(\sigma)$ is a legal trace of $O^{\mathrm{ts}}(\mathit{stitch}_P(hp))$, because any state $\langle \rho, \ell_i : x := E_1 + E_2 \to L' \rangle$ of σ is always preceded by the state $\langle \rho, \mathbb{1}_i : \mathit{guard}\ \rho_i^t \to \ell_i \rangle$ and $\rho \in \gamma_{\mathit{store}}(\rho_i^t)$ must hold. Thus, by Corollary 7.1, $\mathbf{E}[\![E_1 + E_2]\!]\rho \in \mathbf{E}^t[\![E_1 + E_2]\!]\rho^t = \mathrm{Type}$, so that $\mathbf{A}[\![x := E_1 +_{\mathrm{Type}} E_2]\!]\rho = \mathbf{A}[\![x := E_1 + E_2]\!]\rho$ holds. Consequently, the trace fragment

$$sp(\langle \rho, \mathbb{1}_i : \mathit{guard}\ \rho_i^t \to \ell_i \rangle \langle \rho, \ell_i : x := E_1 + E_2 \to L' \rangle) = \\ \langle \rho, \mathbb{1}_i : \mathit{guard}\ \rho_i^t \to \ell_i \rangle \langle \rho, \ell_i : x := E_1 +_{\mathrm{Type}} E_2 \to L' \rangle$$

is legal in $O^{\mathrm{ts}}(\mathit{stitch}_P(hp))$. Furthermore, let us also observe that $sc \circ ts = sc$ trivially holds.

Thus, following the scheme in Section 6.2, these two functions td and ts allow us to define $td_{\mathit{full}} : \mathrm{Trace}_{O^{\mathrm{ts}}_{\mathit{full}}(P,hp)} \to \mathrm{Trace}_{\mathit{extr}_{hp}(P)}$ and $ts_{\mathit{full}} : \mathrm{Trace}_{\mathit{extr}_{hp}(P)} \to \mathrm{Trace}_{O^{\mathrm{ts}}_{\mathit{full}}(P,hp)}$ such that $\alpha_{sc} \circ td_{\mathit{full}} = \alpha_{sc} = \alpha_{sc} \circ ts_{\mathit{full}}$, so that the thesis follows by Lemma 6.8. □

Example 7.3. Let us consider the following sieve of Eratosthenes in a Javascript-like language—this is taken from the running example in [Gal et al. 2009]—where primes is an array initialized with 100 *true* values:

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

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 $\bot$, that is, we assume that the program generates an error (e.g., it is aborted). The above program is encoded in our language as follows:

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

Let us consider the following type environment

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

where $\mathit{primes}[n]/\,\mathrm{Bool}$ is a shorthand for $\mathit{primes}[0]/\,\mathrm{Bool}, \ldots, \mathit{primes}[99]/\,\mathrm{Bool}$. Then the first traced 2-hot path on the type abstraction $\mathrm{Store}^{\mathrm{t}}$ is $hp_1 \triangleq \langle \rho^t, C_4, \rho^t, C_5, \rho^t, C_6 \rangle$. As a consequence, the typed trace extraction of hp_1 yields:

$$\begin{aligned}
P_1 &\triangleq O^{\mathrm{ts}}_{\mathit{full}}(P, hp_1)\\
&= P \smallsetminus \{C_4, C_4^c\} \cup \{\overline{L_4} : k < 100 \to L_5,\ \overline{L_4} : \neg(k < 100) \to L_7\} \cup O^{\mathrm{ts}}(\mathit{stitch}_P(hp_1))
\end{aligned}$$

where:

$$\begin{aligned}
O^{\mathrm{ts}}(\mathit{stitch}_P(hp_1)) = \big\{ & H_0 \equiv L_4 : \mathit{guard}\,(\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \ell_0,\\
& H_0^c \equiv L_4 : \neg\mathit{guard}\,(\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \overline{L_4},\\
& H_1 \equiv \ell_0 : k < 100 \to \mathbb{l}_1,\ H_1^c \equiv \ell_0 : \neg(k < 100) \to L_7,\\
& H_2 \equiv \mathbb{l}_1 : \mathit{guard}\,(\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \ell_1,\\
& H_2^c \equiv \mathbb{l}_1 : \neg\mathit{guard}\,(\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to L_5,\\
& H_3 \equiv \ell_1 : \mathit{primes}[k] := \mathsf{ff} \to \mathbb{l}_2,\\
& H_4 \equiv \mathbb{l}_2 : \mathit{guard}\,(\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \ell_2,\\
& H_4^c \equiv \mathbb{l}_2 : \neg\mathit{guard}\,(\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to L_6,\\
& H_5 \equiv \ell_2 : k := k +_{\mathrm{Int}} i \to L_4\big\}. \quad \square
\end{aligned}$$

8. CONSTANT VARIABLE FOLDING

Constant variable folding, a.k.a. constant propagation [Wegman and Zadeck 1991], is a standard and well-known program optimization, whose goal is to detect which program variables at some program point are constant on all possible executions and then to propagate these constant values as far forward through the program as possible. Guo and Palsberg [2011] show how to define this optimization along hot paths and then prove its correctness. As a significant example, we show here how to specify and prove the correctness w.r.t. the store changes abstraction α_{sc} of this simple hot path optimization according to the approach defined in Section 6.2.

The constant propagation store abstraction CP_{st} and its corresponding GI $(\alpha_{\rm CP}, \wp(\mathrm{Store}), \mathrm{CP}_{st}, \gamma_{\rm CP})$ have been defined in Example 3.1. Following Section 5, any abstract store $[x_i/a_i \mid x_i \in \mathrm{Var}] \in \mathrm{CP}_{st}$, where, as usual, the bindings x_i/undef are omitted, defines a corresponding guard $x_0 : a_0, \dots, x_n : a_n \in \mathrm{BExp}$ whose semantics is induced by the GI $(\alpha_{\rm CP}, \wp(\mathrm{Store}), \mathrm{CP}_{st}, \gamma_{\rm CP})$, as defined in Section 5: for any $\rho \in \mathrm{Store}$,

$$\mathbf{A}[\![\textbf{guard}\, x_0 : a_0, ..., x_n : a_n]\!]\rho \triangleq \begin{cases} \rho & \text{if } \rho \in \gamma_{\rm CP}([x_i/a_i \mid x_i \in \mathrm{Var}]) \\ \bot & \text{otherwise} \end{cases}$$
$$= \begin{cases} \rho & \text{if } \forall i.\, \rho(x_i) \in \gamma_{cp}(a_i) \\ \bot & \exists i.\, \rho(x_i) \notin \gamma_{cp}(a_i) \end{cases}$$

Therefore, we have that:

$$\begin{aligned}
&\mathbf{A}[\![\textbf{guard}\, x : 2, y : \mathtt{foo}]\!][x/2, y/3] = \bot,\\
&\mathbf{A}[\![\textbf{guard}\, x : 2, y : \mathtt{foo}]\!][x/2, y/\mathtt{foo}, z/4] = \bot,\\
&\mathbf{A}[\![\textbf{guard}\, x : 2, y : \mathtt{foo}]\!][x/2] = \bot,\\
&\mathbf{A}[\![\textbf{guard}\, x : 2, y : \mathtt{foo}]\!][x/2, y/\mathtt{foo}] = [x/2, y/\mathtt{foo}],\\
&\mathbf{A}[\![\textbf{guard}\, x : 2, y : \top]\!][x/2, y/\mathtt{foo}] = [x/2, y/\mathtt{foo}],\\
&\mathbf{A}[\![\textbf{guard}\, x : 2, y : \top]\!][x/2] = [x/2].
\end{aligned}$$

Let us consider some hot path $hp = \langle \rho_0^c, C_0, \dots, \rho_n^c, C_n \rangle \in \alpha_{hot}^N(\mathrm{Trace}_P)$ on the constant propagation abstraction CP_{st}, where each ρ_i^c is therefore an abstract store in CP_{st}, whose corresponding guard in $\mathit{stitch}_P(hp)$ will be denoted by $guard\ \rho_i^c$. The constant value information encoded in these guards is used to define the variable folding in the stitched hot path. Following Guo and Palsberg [2011, Section 2.4], let $\mathrm{FV} : \wp(\mathbb{C}) \to \wp(\mathrm{Var})$ denote the function that returns the "free" variables occurring in some set of commands (in particular, a well-defined program), i.e., $\mathrm{FV}(P)$ is the set of variables occurring in P which are never-assigned-to in some command of P. As in Guo and Palsberg [2011], constant variable folding is restricted to expressions E of some assignment $x := E$ and is defined as a program transform which exploits the constant information recorded by abstract guards in $\mathit{stitch}_P(hp)$. The constant folding function $\mathbf{cf}_{hp} : \mathit{stitch}_P(hp) \to \mathbb{C}$ is defined as follows:

$$\mathbf{cf}_{hp}(\ell_i : x := E \to L') \triangleq$$
$$\begin{cases} \ell_i : x := E[y_1/v_{y_1}, ..., y_k/v_{y_k}] \to L' & \text{if } \{y_1, ..., y_k\} = \{y \in \mathit{vars}(E) \cap \mathrm{FV}(\mathit{stitch}_P(hp)) \mid \\ & \qquad \rho_i^c(y) = v_y \in \mathrm{Value}\} \neq \varnothing \\ \ell_i : x := E \to L' & \text{otherwise} \end{cases}$$
$$\mathbf{cf}_{hp}(C) \triangleq C \qquad \text{if } C \not\equiv \ell_i : x := E \to L'$$

where $E[y_1/v_{y_1}, ..., y_k/v_{y_k}]$ denotes the standard synctactic substitution of variables $y_j \in \mathit{vars}(E)$ with constant values $\rho_i^c(y_j) = v_{y_j} \in \mathrm{Value}$. Hence, when the abstract guard $\mathit{guard}\ \rho_i^c$ which precedes an assignment $\ell_i : x := E \to L'$ tells us that a free variable y occuring in the expression E is definitely a constant value $v_y \in \mathrm{Value}$ then $\mathbf{cf}_{hp}$ performs the corresponding variable folding in E. Thus, the hot path constant folding optimization is defined by

$$O^{\mathrm{cf}}(\mathit{stitch}_P(hp)) \triangleq \{\mathbf{cf}_{hp}(C) \mid C \in \mathit{stitch}_P(hp)\}$$

and, in turn, this induces the full constant folding optimization $O^{\mathrm{cf}}_{\mathit{full}}(P, hp)$. The correctness of this constant folding optimization can be proved for the store changes observational abstraction

THEOREM 8.1 (**CORRECTNESS OF CONSTANT FOLDING OPTIMIZATION**). *For any hot path* $hp \in \alpha^N_{hot}(\mathrm{Trace}_P)$ *w.r.t. the constant propagation store abstraction* CP_{st}*,* $\alpha_{sc}(\mathbf{T}[\![O^{\mathrm{cf}}_{\mathit{full}}(P, hp)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$.

This proof is omitted, since it follows the same pattern of Theorem 7.2 for the correctness of typed trace extraction, in particular it relies on Lemma 6.8.

Example 8.2. Let us consider the following program written in a while-language:

```
$x := 0$; $a := 2$;
**while** $(x \le 15)$ **do**
    **if** $(x \le 5)$ **then** $x := x + a$
    **else** $\{a := a + 1;\ x := x + a;\}$
```

whose translation as $P \in \mathrm{Program}$ goes as follows:

$$\begin{aligned}
P = \big\{ & C_0 \equiv L_0 : x := 0 \to L_1,\ C_1 \equiv L_1 : a := 2 \to L_2\\
& C_2 \equiv L_2 : x \le 15 \to L_3,\ C_2^c \equiv L_2 : \neg(x \le 15) \to L_7,\\
& C_3 \equiv L_3 : x \le 5 \to L_4,\ C_3^c \equiv L_3 : \neg(x \le 5) \to L_5,\\
& C_4 \equiv L_4 : x := x + a \to L_2,\ C_5 \equiv L_5 : a := a + 1 \to L_6\\
& C_6 \equiv L_6 : x := x + a \to L_2,\ C_7 \equiv L_7 : \mathsf{skip} \to \text{Ł}\big\}
\end{aligned}$$

The first traced 2-hot path for the abstraction CP_{st} is:

$$hp = \langle [x/\top, a/2], C_2, [x/\top, a/2], C_3, [x/\top, a/2], C_4\rangle.$$

In fact, the initial prefix of the complete trace of P which corresponds to the terminating run of P is as follows:

$$\begin{aligned}
&\langle [\,], C_0\rangle\langle [x/0], C_1\rangle\langle [x/0, a/2], C_2\rangle\langle [x/0, a/2], C_3\rangle\langle [x/0, a/2], C_4\rangle\langle [x/2, a/2], C_2\rangle\\
&\qquad\langle [x/2, a/2], C_3\rangle\langle [x/2, a/2], C_4\rangle\langle [x/4, a/2], C_2\rangle\langle [x/4, a/2], C_3\rangle\langle [x/4, a/2], C_4\rangle
\end{aligned}$$

so that $hp \in \alpha^2_{hot}(\mathrm{Trace}_P)$. Hence, the constant folding optimization O^{cf} along hp provides:

$$O^{\mathrm{cf}}_{\mathit{full}}(P, hp) = P \smallsetminus \{C_2, C_2^c\} \cup \{\overline{L_2} : x \le 15 \to L_3,\ \overline{L_2} : \neg(x \le 15) \to L_7\} \cup O^{\mathrm{cf}}(\mathit{stitch}_P(hp))$$

where:

$$\begin{aligned}
O^{\mathrm{cf}}(\mathit{stitch}_P(hp)) = \big\{ & H_0 \equiv L_2 : \mathit{guard}\,[x\!:\!\top, a\!:\!2] \to \ell_0,\ H_0^c \equiv L_2 : \neg\mathit{guard}\,[x\!:\!\top, a\!:\!2] \to \overline{L_2},\\
& H_1 \equiv \ell_0 : x \le 15 \to \mathbb{l}_1,\ H_1^c \equiv \ell_0 : \neg(x \le 15) \to L_7,\\
& H_2 \equiv \mathbb{l}_1 : \mathit{guard}\,[x\!:\!\top, a\!:\!2] \to \ell_1,\ H_2^c \equiv \mathbb{l}_1 : \neg\mathit{guard}\,[x\!:\!\top, a\!:\!2] \to L_3,\\
& H_3 \equiv \ell_1 : x \le 5 \to \mathbb{l}_2,\ H_3^c \equiv \ell_1 : \neg(x \le 5) \to L_5,\\
& H_4 \equiv \mathbb{l}_2 : \mathit{guard}\,[x\!:\!\top, a\!:\!2] \to \ell_2,\ H_4^c \equiv \mathbb{l}_2 : \neg\mathit{guard}\,[x\!:\!\top, a\!:\!2] \to L_4,\\
& H_5 \equiv \ell_2 : x := x + 2 \to L_2\big\}.
\end{aligned}$$

Therefore, this hot path optimization allows us to fold the constant value 2 for the variable a, in the hot path command $H_5 \equiv \ell_2 : x := x + 2 \to L_2$. □

9. NESTED HOT PATHS

Once a first hot path hp_1 has been extracted by transforming P to $P_1 \triangleq \mathit{extr}_{hp_1}(P)$, it may well happen that a new hot path hp_2 in P_1 contains hp_1 as a nested sub-path. Following TraceMonkey's trace recording strategy [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.

In order to handle nested hot paths, we need a more general definition of hot path which takes into account previously extracted hot paths and a corresponding program transform for extracting nested hot paths. Let P be the original program and let P' be a hot path transform of P so that $P' \smallsetminus P$ contains all the commands (guards included) in the hot path. We define a function $\mathit{hotcut} : \mathrm{Trace}_{P'} \to (\mathrm{State}_{P'})^*$ that cuts from an execution trace σ of P' all the states whose commands appear in some previous hot path hp except for the entry and exit states of hp:

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

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

$$\begin{aligned} \mathit{outerhot}^N(\sigma) \triangleq \{\langle a_i, C_i\rangle \cdots \langle a_j, C_j\rangle \in (\mathrm{State}^\sharp_{P'})^* \mid \exists \langle \rho_i, C_i\rangle \cdots \langle \rho_j, C_j\rangle \in \mathit{loop}(\mathit{hotcut}(\sigma)) \\ \text{such that } i \leq j,\ \alpha_{\mathit{store}}(\langle \rho_i, C_i\rangle \cdots \langle \rho_j, C_j\rangle) = \langle a_i, C_i\rangle \cdots \langle a_j, C_j\rangle, \\ \mathit{count}(\alpha_{\mathit{store}}(\mathit{hotcut}(\sigma)), \langle a_i, C_i\rangle \cdots \langle a_j, C_j\rangle) \geq N\}. \end{aligned}$$

Clearly, when $P' = P$ it turns out that $\mathit{hotcut} = \lambda\sigma.\sigma$ so that $\mathit{outerhot}^N = \mathit{hot}^N$. We define the usual collecting version of $\mathit{outerhot}^N$ on $\wp(\mathrm{Trace}_{P'})$ as the abstraction map $\alpha^N_{\mathit{outerhot}} \triangleq \lambda T. \cup_{\sigma \in T} \mathit{outerhot}^N(\sigma)$. Then, $\alpha^N_{\mathit{outerhot}}(\mathbf{T}[\![P']\!])$ provides the set of N-hot paths in P'.

Example 9.1. Let us consider again Example 5.3, where $\mathrm{Store}^\sharp$ is the trivial one-point store abstraction $\{\top\}$. In Example 5.3, we first extracted $hp_1 = \langle \top, C_1, \top, C_2, \top, C_3^c\rangle$ by transforming P to $P_1 \triangleq \mathit{extr}_{hp}(P)$. We then consider the following trace in $\mathbf{T}[\![P_1]\!]$:

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

Thus, here we have that

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

so that $hp_2 = \langle \top, H_0, \top, H_5^c, \top, C_4\rangle \in \alpha^2_{\mathit{outerhot}}(\mathbf{T}[\![P_1]\!])$. Hence, hp_2 contains a nested hot path, which is called at the beginning of hp_2 and whose entry and exit commands are, respectively, H_0 and H_5^c. □

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

- If for all $i \in [0, n]$, $C_i \in P$ then hp actually is a hot path in P, i.e., $hp \in \alpha^N_{\mathit{hot}}(\mathbf{T}[\![P]\!])$.

– Otherwise, there exists some $C_k \notin P$. If $C_i \in P$ and $C_{i+1} \notin P$ then C_{i+1} is the entry command of some inner hot path; on the other hand, if $C_i \notin P$ and $C_{i+1} \in P$ then C_i is the exit command of some inner hot path.

The transform of P' for extracting hp is then given as the following generalization of Definition 5.1.

Definition 9.2 (***Nested trace extraction transform***). The *nested trace extraction transform* of P' for the hot path $hp = \langle a_0, C_0, \ldots, a_n, C_n \rangle$ is:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{extr}_{hp}(P') \triangleq P} \\
(1) & \smallsetminus (\{C_0 \mid C_0 \in P\} \cup \{\mathit{cmpl}(C_0) \mid \mathit{cmpl}(C_0) \in P\}) \\
(2) & \cup \, \{\overline{H_0} : \mathit{act}(C_0) \to L_1 \mid C_0 \in P\} \cup \{\overline{H_0} : \neg \mathit{act}(C_0) \to L_1^c \mid \mathit{cmpl}(C_0) \in P\} \\
(3) & \cup \, \{L_0 : \mathit{guard}\ E_{a_0} \to \hbar_0,\ L_0 : \neg \mathit{guard}\ E_a \to \overline{H_0} \mid C_0 \in P\} \\
(4) & \cup \, \{\hbar_i : \mathit{act}(C_i) \to \mathbb{h}_{i+1} \mid i \in [0, n-1], C_i, C_{i+1} \in P\} \cup \{\hbar_n : \mathit{act}(C_n) \to L_0 \mid C_n \in P\} \\
(5) & \cup \, \{\hbar_i : \neg \mathit{act}(C_i) \to L^c_{\mathit{next}(i)} \mid i \in [0, n], C_i, \mathit{cmpl}(C_i) \in P\} \\
(6) & \cup \, \{\mathbb{h}_i : \mathit{guard}\ E_{a_i} \to \hbar_i, \mathbb{h}_i : \neg \mathit{guard}\ E_{a_i} \to L_i \mid i \in [1, n], C_i \in P\} \\
(7) & \cup \, \{\hbar_i : \mathit{act}(C_i) \to L_{i+1} \mid i \in [0, n-1], C_i \in P, C_{i+1} \notin P\} \\
(8) & \smallsetminus \{C_i \mid i \in [0, n-1], C_i \notin P, C_{i+1} \in P\} \\
(9) & \cup \, \{L_i : \mathit{act}(C_i) \to \mathbb{h}_{i+1} \mid i \in [0, n-1], C_i \notin P, C_{i+1} \in P\}
\end{array}
$$

where we define $\mathit{stitch}_{P'}(hp) \triangleq (3) \cup (4) \cup (5) \cup (6) \cup (7) \cup (9)$. □

Let us observe that:

– Clauses (1)–(6) are the same clauses of the trace extraction transform of Definition 5.1, with the additional constraint that all the commands C_i of hp are required to belong to the original program P. This is equivalent to ask that any C_i is not the entry or exit command of a nested hot path inside hp, i.e., $C_i \notin P' \smallsetminus P$. In Definition 5.1, where no previous hot path extraction is assumed, any command C_i of hp belongs to P, so that this constraint is trivially satisfied.

– Clause (7) where $C_i \in P$ and $C_{i+1} \notin P$, namely $\mathit{next}(C_i)$ 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 neatly nest nhp in hp.

– Clauses (8)–(9) where $C_i \notin P$ and $C_{i+1} \in P$, i.e., C_i is the exit command of a nested hot path nhp that returns to the program point $\mathit{lbl}(C_{i+1})$, performs the relabeling of $\mathit{suc}(C_i)$ in C_i in order to return from nhp to hp;

– $\overline{H_0}$, $\hbar_i$ and $\mathbb{h}_i$ 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 $\alpha^2_{\mathit{outerhot}}(\mathbf{T}[\![P_1]\!])$ is:

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

According to Definition 9.2, trace extraction of hp_2 in P_1 yields the following transform:

$$\begin{array}{ll} & \mathit{extr}_{hp_2}(P_1) \triangleq \\ \text{[by clause (8)]} & P_1 \smallsetminus \{H_5^c\} \\ \text{[by clause (9)]} & \cup \{\ell_2 : (x\%3 = 0) \to \mathbb{h}_2\} \\ \text{[by clause (6)]} & \cup \{\mathbb{h}_2 : \mathit{guard}\ E_\top \to \hbar_2, \mathbb{h}_2 : \neg \mathit{guard}\ E_\top \to L_4\} \\ \text{[by clause (4)]} & \cup \{\hbar_2 : x := x + 3 \to L_1\} \end{array}$$

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

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:

$$hp_2 \triangleq \langle \rho^t, C_1, \rho^t, C_2, \rho^t, C_3, \rho^t, H_0, \rho^t, H_1^c, \rho^t, C_7 \rangle$$

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

$$\begin{array}{rl} P_2 \triangleq O^{\mathrm{ts}}_{\mathit{full}}(P_1, hp_2) = P_1 \smallsetminus \{C_1, C_1^c\} \cup \{ & \\ & \overline{H_0} : i < 100 \to L_2,\ \overline{H_0} : \neg(i < 100) \to L_8, \\ & H_6 \equiv L_1 : \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \hbar_0, \\ & H_6^c \equiv L_1 : \neg \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \overline{H_0}, \\ & H_7 \equiv \hbar_0 : i < 100 \to \mathbb{h}_1,\ H_7^c \equiv \hbar_0 : \neg(i < 100) \to L_8, \\ & H_8 \equiv \mathbb{h}_1 : \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \hbar_1, \\ & H_8^c \equiv \mathbb{h}_1 : \neg \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to L_2, \\ & H_9 \equiv \hbar_1 : \mathit{primes}[i] = \mathit{tt} \to \mathbb{h}_1,\ H_9^c \equiv \hbar_1 : \neg(\mathit{primes}[i] = \mathit{tt}) \to L_7, \\ & H_{10} \equiv \mathbb{h}_2 : \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \hbar_2, \\ & H_{10}^c \equiv \mathbb{h}_2 : \neg \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to L_3, \\ & H_{11} \equiv \hbar_2 : k := i +_{\mathrm{Int}} i \to L_4 \big\} \\ & \smallsetminus \{H_1^c\} \cup \big\{(H_1^c)' \equiv \ell_0 : \neg(k < 100) \to \mathbb{h}_3, \\ & H_{12} \equiv \mathbb{h}_3 : \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \hbar_3, \\ & H_{12}^c \equiv \mathbb{h}_3 : \neg \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to L_7, \\ & H_{13} \equiv \hbar_3 : i := i +_{\mathrm{Int}} 1 \to L_1 \big\}. \end{array}$$

Finally, a third traced 2-hot path in P_2 is $hp_3 \triangleq \langle \rho^t, H_6, \rho^t, H_9^c, \rho^t, C_7 \rangle$ which contains a nested hot path which is called at the beginning of hp_3 and whose entry and exit commands are, respectively, H_6 and H_9^c. Here, typed trace extraction of hp_3 yields:

$$\begin{array}{rl} P_3 \triangleq O^{\mathrm{ts}}_{\mathit{full}}(P_2, hp_3) = P_2 \smallsetminus \{H_9^c\} \cup \big\{ & (H_9^c)' \equiv \hbar_1 : \neg(\mathit{primes}[i] = \mathit{tt}) \to \mathbb{j}_2, \\ & \mathbb{j}_2 : \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to \jmath_2, \\ & \mathbb{j}_2 : \neg \mathit{guard}\ (\mathit{primes}[n] : \mathrm{Bool}, i : \mathrm{Int}, k : \mathrm{Int}) \to L_7, \\ & \jmath_2 : i := i +_{\mathrm{Int}} 1 \to L_1 \big\}. \end{array}$$

We have thus obtained the same three trace extraction steps described by Gal et al. [2009, Section 2]. In particular, in P_1 we specialized the typed addition operation $k +_{\mathrm{Int}} i$, in P_2 we specialized $i +_{\mathrm{Int}} i$ and $i +_{\mathrm{Int}} 1$, while in P_3 we specialized once again $i +_{\mathrm{Int}} 1$ 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 at POPL 2011 symposium 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:

$$\begin{aligned}
&E ::= v \mid x \mid E_1 + E_2\\
&B ::= \mathsf{tt} \mid \mathsf{ff} \mid E_1 \leq E_2 \mid \neg B \mid B_1 \wedge B_2\\
&\mathrm{Cmd} \ni c ::= \mathbf{skip};\ \mid x := E;\ \mid \mathbf{if}\ B\ \mathbf{then}\ S \mid \mathbf{while}\ B\ \mathbf{do}\ S \mid \mathbf{bail}\ B\ \mathbf{to}\ S\\
&\mathrm{Stm} \ni S ::= \epsilon \mid cS
\end{aligned}$$

where ϵ stands for the empty string. Thus, any statement $S \in \mathrm{Stm}$ 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 $S_1, S_2 \in \mathrm{Stm}$ then $S_1S_2 \in \mathrm{Stm}$, where S_1S_2 denotes a simple string concatenation of S_1 and S_2. We denote by $\mathrm{State}_{GP} \triangleq \mathrm{Store} \times \mathrm{Stm}$ the set of states for this language. The baseline small-step operational semantics $\to_B \subseteq \mathrm{State}_{GP} \times \mathrm{State}_{GP}$ is standard and is given in continuation-style (where $K \in \mathrm{Stm}$):

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

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

$$\mathrm{Trace}^{GP} \triangleq \{\sigma \in \mathrm{State}^{+}_{GP} \mid \forall i \in [0, |\sigma| - 1).\, \sigma_i \to_B \sigma_{i+1}\}$$

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

$$\mathbf{T}_{GP}[\![S]\!] = \mathrm{Trace}^{GP}_S \triangleq \{\sigma \in \mathrm{Trace}^{GP} \mid \sigma_0 = \langle \rho, S\rangle,\ \rho \in \mathrm{Store}_S\}.$$

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 Stm can be compiled into $\mathrm{Program}$ by resorting to an *injective* labeling function $\boldsymbol{l} : \mathrm{Stm} \to \mathbb{L}$ that assigns different labels to different statements.

Definition 10.1 (***Language compilation***). The "first command" compilation function $\mathsf{C} : \mathrm{Stm} \to \wp(\mathbb{C})$ is defined as follows:

$$
\begin{aligned}
\mathsf{C}(\epsilon) &\triangleq \{\boldsymbol{l}(\epsilon) : \mathrm{skip} \to \text{Ł}\}\\
\mathsf{C}\big(S' \equiv (\textbf{skip}; K)\big) &\triangleq \{\boldsymbol{l}(S') : \mathrm{skip} \to \boldsymbol{l}(K)\}\\
\mathsf{C}\big(S' \equiv (x := E; K)\big) &\triangleq \{\boldsymbol{l}(S') : x := E \to \boldsymbol{l}(K)\}\\
\mathsf{C}\big(S' \equiv ((\textbf{if } B \textbf{ then } S)K)\big) &\triangleq \{\boldsymbol{l}(S') : B \to \boldsymbol{l}(SK), \boldsymbol{l}(S') : \neg B \to \boldsymbol{l}(K)\}\\
\mathsf{C}\big(S' \equiv ((\textbf{while } B \textbf{ do } S)K)\big) &\triangleq \{\boldsymbol{l}(S') : \mathrm{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (S \textbf{ while } B \textbf{ do } S))K)\}\\
\mathsf{C}\big(S' \equiv ((\textbf{bail } B \textbf{ to } S)K)\big) &\triangleq \{\boldsymbol{l}(S') : B \to \boldsymbol{l}(S), \boldsymbol{l}(S') : \neg B \to \boldsymbol{l}(K)\}
\end{aligned}
$$

Then, the full compilation function $\mathcal{C} : \mathrm{Stm} \to \wp(\mathbb{C})$ is recursively defined by the following clauses:

$$
\begin{aligned}
\mathcal{C}(\epsilon) &\triangleq \mathsf{C}(\epsilon)\\
\mathcal{C}(\textbf{skip}; K) &\triangleq \mathsf{C}(\textbf{skip}; K) \cup \mathcal{C}(K)\\
\mathcal{C}(x := E; K) &\triangleq \mathsf{C}(x := E; K) \cup \mathcal{C}(K)\\
\mathcal{C}((\textbf{if } B \textbf{ then } S)K) &\triangleq \mathsf{C}((\textbf{if } B \textbf{ then } S)K) \cup \mathcal{C}(SK) \cup \mathcal{C}(K)\\
\mathcal{C}((\textbf{while } B \textbf{ do } S)K) &\triangleq \mathsf{C}((\textbf{while } B \textbf{ do } S)K) \cup \mathcal{C}((\textbf{if } B \textbf{ then } (S \textbf{ while } B \textbf{ do } S))K)\\
\mathcal{C}((\textbf{bail } B \textbf{ to } S)K) &\triangleq \mathsf{C}((\textbf{bail } B \textbf{ to } S)K) \cup \mathcal{C}(S) \cup \mathcal{C}(K)
\end{aligned}
$$

Given $S \in \mathrm{Stm}$, $\boldsymbol{l}(S)$ is the initial label of $\mathcal{C}(S)$, while Ł is, as usual, the undefined label where the execution becomes stuck. □

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

Example 10.2. Consider the following program $S \in \mathrm{Stm}$ in Guo and Palsberg's syntax:

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

S is then compiled in our language by $\mathcal{C}$ in Definition 10.1 as follows:

$$
\begin{aligned}
\mathcal{C}(S) = \big\{&\boldsymbol{l}(S) : x := 0 \to \boldsymbol{l}_{\textbf{while}}, \boldsymbol{l}_{\textbf{while}} : \mathrm{skip} \to \boldsymbol{l}_{\textbf{ifwhile}},\\
&\boldsymbol{l}_{\textbf{ifwhile}} : B_1 \to \boldsymbol{l}_1, \boldsymbol{l}_{\textbf{ifwhile}} : \neg B_1 \to \boldsymbol{l}_2, \boldsymbol{l}_1 : x := 1 \to \boldsymbol{l}_{\textbf{while}},\\
&\boldsymbol{l}_2 : x := 2 \to \boldsymbol{l}_{\textbf{bail}}, \boldsymbol{l}_{\textbf{bail}} : B_2 \to \boldsymbol{l}_3, \boldsymbol{l}_{\textbf{bail}} : \neg B_2 \to \boldsymbol{l}_4,\\
&\boldsymbol{l}_3 : x := 3 \to \boldsymbol{l}_\epsilon, \boldsymbol{l}_4 : x := 4 \to \boldsymbol{l}_\epsilon, \boldsymbol{l}_\epsilon : \mathrm{skip} \to \text{Ł}\big\}.
\end{aligned}
$$

Notice that in the command $\boldsymbol{l}_{\textbf{bail}} : B_2 \to \boldsymbol{l}_3$, the label $\boldsymbol{l}_3$ stands for $\boldsymbol{l}(x := 3;)$ so that $\mathcal{C}(x := 3;) \equiv \boldsymbol{l}_3 : x := 3 \to \boldsymbol{l}_\epsilon$, i.e., after the execution of $x := 3$ the program terminates. □

Correctness for the above compilation function $\mathcal{C}$ means that for any $S \in \mathrm{Stm}$: (i) $\mathcal{C}(S) \in \mathrm{Program}$ and (ii) program traces of S and $\mathcal{C}(S)$ have the same store sequences.

$$
\begin{aligned}
\mathcal{C}^s(\langle \rho, \epsilon \rangle) &\triangleq \langle \rho, \boldsymbol{l}(\epsilon) : \text{skip} \to \text{Ł} \rangle \\
\mathcal{C}^s(\langle \rho, S \equiv (\mathbf{skip}; K) \rangle) &\triangleq \langle \rho, \boldsymbol{l}(S) : \text{skip} \to \boldsymbol{l}(K) \rangle \\
\mathcal{C}^s(\langle \rho, S \equiv (x := E; K) \rangle) &\triangleq \langle \rho, \boldsymbol{l}(S) : x := E \to \boldsymbol{l}(K) \rangle \\
\mathcal{C}^s(\langle \rho, S \equiv ((\mathbf{if}\ B\ \mathbf{then}\ S')K) \rangle) &\triangleq \begin{cases} \langle \rho, \boldsymbol{l}(S) : B \to \boldsymbol{l}(S'K) \rangle & \text{if } \mathbf{B}[\![B]\!]\rho = \textit{true} \\ \langle \rho, \boldsymbol{l}(S) : \neg B \to \boldsymbol{l}(K) \rangle & \text{if } \mathbf{B}[\![B]\!]\rho = \textit{false} \end{cases} \\
\mathcal{C}^s(\langle \rho, S \equiv ((\mathbf{while}\ B\ \mathbf{do}\ S')K) \rangle) &\triangleq \langle \rho, \boldsymbol{l}(S) : \text{skip} \to \boldsymbol{l}((\mathbf{if}\ B\ \mathbf{then}\ (S'\ \mathbf{while}\ B\ \mathbf{do}\ S'))K) \rangle \\
\mathcal{C}^s(\langle \rho, S \equiv ((\mathbf{bail}\ B\ \mathbf{to}\ S')K) \rangle) &\triangleq \begin{cases} \langle \rho, \boldsymbol{l}(S) : B \to \boldsymbol{l}(S') \rangle & \text{if } \mathbf{B}[\![B]\!]\rho = \textit{true} \\ \langle \rho, \boldsymbol{l}(S) : \neg B \to \boldsymbol{l}(K) \rangle & \text{if } \mathbf{B}[\![B]\!]\rho = \textit{false} \end{cases}
\end{aligned}
$$

Fig. 6. Definition of the state compile function $\mathcal{C}^s : \text{State}_{GP} \to \text{State}$.

In the proof we will make use of a "state compile" function $\mathcal{C}^s : \text{State}_{GP} \to \text{State}$ as defined in Figure 6. In turn, $\mathcal{C}^s$ allows us to define a "trace compile" function $\mathcal{C}^t : \mathbf{T}_{GP}[\![S]\!] \to \mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!]$ which applies state-by-state the function $\mathcal{C}^s$ to traces as follows:

$$\mathcal{C}^t(\epsilon) \triangleq \epsilon; \quad \mathcal{C}^t(s\tau) \triangleq \mathcal{C}^s(s)\mathcal{C}^t(\tau).$$

LEMMA 10.3.

(1) $\langle \rho, S \rangle \to_B \langle \rho', S' \rangle \;\Leftrightarrow\; \mathcal{C}^s(\langle \rho', S' \rangle) \in \mathbf{S}(\mathcal{C}^s(\langle \rho, S \rangle))$
(2) *$\mathcal{C}^t$ is well-defined.*

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

$[S \equiv \epsilon]$: Trivially true, since $\langle \rho, S \rangle \not\to_B$ and $\mathbf{S}\langle \rho, \boldsymbol{l}(\epsilon) : \text{skip} \to \text{Ł} \rangle = \varnothing$.

$[S \equiv x := E; K]$ ($\Rightarrow$): If $\langle \rho, x := E; K \rangle \to_B \langle \rho[x/\mathbf{E}[\![E]\!]\rho], K \rangle$, $\mathcal{C}^s(\langle \rho, x := E; K \rangle) = \langle \rho, \boldsymbol{l}(S) : x := E \to \boldsymbol{l}(K) \rangle$ and $\mathcal{C}^s(\langle \rho[x/\mathbf{E}[\![E]\!]\rho], K \rangle) = \langle \rho[x/\mathbf{E}[\![E]\!]\rho], \boldsymbol{l}(K) : A \to \boldsymbol{l}(S') \rangle$ for some action A and statement S', then, by definition of the transition semantics $\mathbf{S}$, $\langle \rho[x/\mathbf{E}[\![E]\!]\rho], \boldsymbol{l}(K) : A \to \boldsymbol{l}(S') \rangle \in \mathbf{S}\langle \rho, \boldsymbol{l}(S) : x := E \to \boldsymbol{l}(K) \rangle$.
($\Leftarrow$): If $\langle \rho'', C \rangle = \mathcal{C}^s(\langle \rho', S' \rangle) \in \mathbf{S}\langle \rho, \boldsymbol{l}(S) : x := E \to \boldsymbol{l}(K) \rangle$ then: (1) $\mathbf{E}[\![E]\!]\rho \neq \mathit{undef}$, (2) $\rho'' = \rho[x/\mathbf{E}[\![E]\!]\rho]$, and therefore $\rho' = \rho[x/\mathbf{E}[\![E]\!]\rho]$; (3) $\mathit{lbl}(C) = \boldsymbol{l}(K)$, and therefore $S' = K$. Hence, $\langle \rho, x := E; K \rangle \to_B \langle \rho[x/\mathbf{E}[\![E]\!]\rho], K \rangle = \langle \rho', S' \rangle$.

$[S \equiv \mathbf{skip}; K]$ Analogous to $S \equiv x := E; K$.

$[S \equiv (\mathbf{if}\ B\ \mathbf{then}\ T)K]$ ($\Rightarrow$): Assume that $\mathbf{B}[\![B]\!]\rho = \textit{false}$, so that $\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle \to_B \langle \rho, K \rangle$, $\mathcal{C}^s(\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle) = \langle \rho, \boldsymbol{l}(S) : \neg B \to \boldsymbol{l}(K) \rangle$ and $\mathcal{C}^s(\langle \rho, K \rangle) = \langle \rho, \boldsymbol{l}(K) : A \to \boldsymbol{l}(T') \rangle$ for some A and $T' \in \text{Stm}$. Hence, by definition of $\mathbf{S}$, $\langle \rho, \boldsymbol{l}(K) : A \to \boldsymbol{l}(T') \rangle \in \mathbf{S}\langle \rho, \boldsymbol{l}(S) : \neg B \to \boldsymbol{l}(K) \rangle$. On the other hand, if $\mathbf{B}[\![B]\!]\rho = \textit{true}$ then $\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle \to_B \langle \rho, TK \rangle$, $\mathcal{C}^s(\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle) = \langle \rho, \boldsymbol{l}(S) : B \to \boldsymbol{l}(TK) \rangle$ and $\mathcal{C}^s(\langle \rho, TK \rangle) = \langle \rho, \boldsymbol{l}(TK) : A \to \boldsymbol{l}(T') \rangle$ for some A and T'. Hence, $\langle \rho, \boldsymbol{l}(TK) : A \to \boldsymbol{l}(T') \rangle \in \mathbf{S}\langle \rho, \boldsymbol{l}(S) : B \to \boldsymbol{l}(TK) \rangle$.
($\Leftarrow$): Assume that $\mathbf{B}[\![B]\!]\rho = \textit{false}$, so that $\mathcal{C}^s(\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle) = \langle \rho, \boldsymbol{l}(S) : \neg B \to \boldsymbol{l}(K) \rangle$, and $\langle \rho'', C \rangle = \mathcal{C}^s(\langle \rho', S' \rangle) \in \mathbf{S}\langle \rho, \boldsymbol{l}(S) : \neg B \to \boldsymbol{l}(K) \rangle$. Hence: (1) $\rho'' = \rho$ and therefore $\rho' = \rho$; (2) $\mathit{lbl}(C) = \boldsymbol{l}(K)$, and therefore $S' = K$. Hence, $\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle \to_B \langle \rho, K \rangle = \langle \rho', S' \rangle$. On the other hand, if $\mathbf{B}[\![B]\!]\rho = \textit{true}$ then $\mathcal{C}^s(\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle) = \langle \rho, \boldsymbol{l}(S) : B \to \boldsymbol{l}(TK) \rangle$ and $\langle \rho'', C \rangle = \mathcal{C}^s(\langle \rho', S' \rangle) \in \mathbf{S}\langle \rho, \boldsymbol{l}(S) : B \to \boldsymbol{l}(TK) \rangle$. We thus have that: (1) $\rho'' = \rho$ and therefore $\rho' = \rho$; (2) $\mathit{lbl}(C) = \boldsymbol{l}(TK)$, and therefore $S' = TK$. Hence, $\langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ T)K \rangle \to_B \langle \rho, TK \rangle = \langle \rho', S' \rangle$.

$[S \equiv (\textbf{while } B \textbf{ do } T)K]$ ($\Rightarrow$): We have that $\langle\rho, (\textbf{while } B \textbf{ do } T)K\rangle \to_B \langle\rho, (\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K\rangle$ and $\mathcal{C}^s(\langle\rho, (\textbf{while } B \textbf{ do } T)K\rangle) = \langle\rho, \boldsymbol{l}(S) : \text{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K)\rangle$. If $\mathbf{B}[\![B]\!]\rho = \textit{true}$ then $\mathcal{C}^s(\langle\rho, (\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K\rangle) = \langle\rho, \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K) : B \to \boldsymbol{l}(T\,(\textbf{while } B \textbf{ do } T)K)\rangle$; on the other hand, if $\mathbf{B}[\![B]\!]\rho = \textit{false}$ then $\mathcal{C}^s(\langle\rho, (\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K\rangle) = \langle\rho, \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K) : \neg B \to \boldsymbol{l}(K)\rangle$. In both cases, we have that:

$$\begin{aligned}
&\langle\rho, \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K) : B \to \boldsymbol{l}(T\,(\textbf{while } B \textbf{ do } T)K)\rangle,\\
&\langle\rho, \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K) : B \to \boldsymbol{l}(K)\rangle\\
&\qquad\qquad\qquad \in \mathbf{S}\langle\rho, \boldsymbol{l}(S) : \text{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K)\rangle.
\end{aligned}$$

($\Leftarrow$): If $\langle\rho'', C\rangle = \mathcal{C}^s(\langle\rho', S'\rangle) \in \mathbf{S}\langle\rho, \boldsymbol{l}(S) : \text{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K)\rangle$ then: (1) $\rho'' = \rho$, and therefore $\rho' = \rho$; (2) $\mathit{lbl}(C) = \boldsymbol{l}((\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K)$, and therefore $S' = (\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K$. Hence, $\langle\rho, (\textbf{while } B \textbf{ do } T)K\rangle \to_B \langle\rho, (\textbf{if } B \textbf{ then } (T\,\textbf{while } B \textbf{ do } T))\,K\rangle = \langle\rho', S'\rangle$.

$[S \equiv (\textbf{bail } B \textbf{ to } T)K]$ Analogous to $S \equiv (\textbf{if } B \textbf{ then } T)K$.

Let us now turn to point (2). By the $\Rightarrow$ implication of the equivalence (1), we have that if $\tau \in \mathbf{T}_{GP}[\![S]\!]$ then $\mathcal{C}^t(\tau) \in \mathbf{T}[\![\mathcal{C}(S)]\!]$: this can be shown by an easy induction on the length of τ and by using the fact that if $\mathcal{C}^t(\tau) = \langle\rho_0, C_0\rangle\langle\rho_1, C_1\rangle \cdots \langle\rho_n, C_n\rangle$ then, for any i, $C_i \in \mathcal{C}(S)$. Moreover, since $\boldsymbol{l}(S)$ is the initial label of the compiled program $\mathcal{C}(S)$ and $\mathit{lbl}(C_0) = \boldsymbol{l}(S)$, we also notice that $\mathcal{C}^t(\tau) \in \mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!]$. Therefore, $\mathcal{C}^t$ is a well-defined function. □

Let $\mathit{st} : \text{Trace}^{GP} \cup \text{Trace} \to \text{Store}^*$ be the function that returns the store sequence of any trace, that is:

$$\mathit{st}(\epsilon) \triangleq \epsilon \quad \text{and} \quad \mathit{st}(\langle\rho, S\rangle\sigma) \triangleq \rho \cdot \mathit{st}(\sigma).$$

Also, given a set X of traces, let $\alpha_{st}(X) \triangleq \{\mathit{st}(\sigma) \mid \sigma \in X\}$. Then, correctness of the compilation function $\mathcal{C}$ goes as follows:

THEOREM 10.4 (**CORRECTNESS OF LANGUAGE COMPILATION**). *If* $S \in \text{Stm}$ *then* $\mathcal{C}(S) \in \text{Program}$ *and* $\alpha_{st}(\mathbf{T}_{GP}[\![S]\!]) = \alpha_{st}(\mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!])$.

PROOF. We define a "trace de-compile" function $\mathcal{D}^t : \mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!] \to \mathbf{T}_{GP}[\![S]\!]$ as follows. Consider a trace $\sigma = \langle\rho_0, C_0\rangle \cdots \langle\rho_n, C_n\rangle \in \mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!]$, so that $\mathit{lbl}(C_0) = \boldsymbol{l}(S)$, for any $i \in [0, n]$, $C_i \in \mathcal{C}(S)$ and for any $i \in [0, n)$, $\langle\rho_{i+1}, C_{i+1}\rangle \in \mathbf{S}[\![\mathcal{C}(S)]\!]\langle\rho_i, C_i\rangle$. Since $\mathit{lbl}(C_0) = \boldsymbol{l}(S)$, by definition of $\mathcal{C}^s$, we have that $\langle\rho_0, C_0\rangle = \mathcal{C}^s(\langle\rho_0, S\rangle)$. Then, since $\langle\rho_1, C_1\rangle \in \mathbf{S}[\![\mathcal{C}(S)]\!](\mathcal{C}^s(\langle\rho_0, S\rangle))$, there exists $S_1 \in \text{Stm}$ such that $\mathit{lbl}(C_1) = \boldsymbol{l}(S_1)$, so that, $\langle\rho_1, C_1\rangle = \mathcal{C}^s(\langle\rho_1, S_1\rangle)$. Hence, from $\mathcal{C}^s(\langle\rho_1, S_1\rangle) \in \mathbf{S}[\![\mathcal{C}(S)]\!](\mathcal{C}^s(\langle\rho_0, S\rangle))$, by the implication $\Leftarrow$ of Lemma 10.3 (1), we obtain that $\langle\rho_0, S\rangle \to_B \langle\rho_1, S_1\rangle$. Thus, an easy induction allows us to show that for any $i \in [1, n]$ there exists $S_i \in \text{Stm}$ such that

$$\langle\rho_0, S\rangle \to_B \langle\rho_1, S_1\rangle \to_B \cdots \to_B \langle\rho_n, S_n\rangle$$

and $\mathcal{C}^s(\langle\rho_i, S_i\rangle) = \langle\rho_i, C_i\rangle$. We therefore define $\mathcal{D}^t(\sigma) \triangleq \langle\rho_0, S\rangle\langle\rho_1, S_1\rangle \cdots \langle\rho_n, S_n\rangle \in \mathbf{T}_{GP}[\![S]\!]$. Moreover, we notice that $\mathit{st}(\mathcal{D}^t(\sigma)) = \mathit{st}(\sigma)$. Let us also observe that $\mathit{st} \circ \mathcal{C}^t = \mathit{st}$, since $\mathcal{C}^t$ does not affect stores.

Summing up, we obtain:

$$\begin{aligned}
\alpha_{st}(\mathbf{T}_{GP}[\![S]\!]) = &\quad [\text{since } st \circ \mathcal{C}^t = st] \\
\alpha_{st}(\mathcal{C}^t(\mathbf{T}_{GP}[\![S]\!])) \subseteq &\quad [\text{by Lemma 10.3 (2), } \mathcal{C}^t \text{ is well-defined}] \\
\alpha_{st}(\mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!]) = &\quad [\text{since } st \circ \mathcal{D}^t = st] \\
\alpha_{st}(\mathcal{D}^t(\mathbf{T}^{\iota}[\![\mathcal{C}(S)]\!])) \subseteq &\quad [\text{since } \mathcal{D}^t \text{ is well-defined}] \\
\alpha_{st}(\mathbf{T}_{GP}[\![S]\!]) &
\end{aligned}$$

and this closes the proof. □

10.3. Bisimulation

Correctness of trace extraction in [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 $\langle \rho, S\rangle \overset{\tau}{\rightarrow}_B \langle \rho, S'\rangle$. Moreover, for the assignment transition $\langle \rho, x := E; K\rangle \rightarrow_B \langle \rho[x/\mathbf{E}[\![E]\!]\rho], K\rangle$, if $\delta = [x/\mathbf{E}[\![E]\!]\rho]$ denotes the corresponding store update of ρ then this transition is also denoted by $\langle \rho, x := E; K\rangle \overset{\delta}{\rightarrow}_B \langle \rho[x/\mathbf{E}[\![E]\!]\rho], K\rangle$. Let $Act \triangleq \{\delta \mid \delta$ is a store update$\} \cup \{\tau\}$. Then, for a nonempty sequence of actions $s = a_1 \cdots a_n \in Act^+$, we define:

$$\langle \rho, S\rangle \overset{s}{\Rightarrow}_B \langle \rho', S'\rangle \quad \text{iff} \quad \langle \rho, S\rangle \overset{\tau}{\rightarrow}{}^{*}_{B} \circ \overset{a_1}{\rightarrow}_B \circ \overset{\tau}{\rightarrow}{}^{*}_{B} \cdots \overset{\tau}{\rightarrow}{}^{*}_{B} \circ \overset{a_n}{\rightarrow}_B \circ \overset{\tau}{\rightarrow}{}^{*}_{B} \langle \rho', S'\rangle,$$

namely, there may be any number of silent transitions either in front of or following any a_i-transition $\overset{a_i}{\rightarrow}_B$. Moreover, if $s \in Act^+$ is a nonempty sequence of actions then $\hat{s} \in Act^*$ denotes the possibly empty sequence of actions where all the occurrences of τ are removed.

Definition 10.5 (**[Guo and Palsberg 2011]**). A relation $R \subseteq \mathrm{Store} \times \mathrm{Stm} \times \mathrm{Stm}$ is a *bisimulation* when $R(\rho, S_1, S_2)$ implies:

(1) if $\langle \rho, S_1\rangle \overset{a}{\rightarrow}_B \langle \rho', S_1'\rangle$ then $\langle \rho, S_2\rangle \overset{\hat{a}}{\Rightarrow}_B \langle \rho', S_2'\rangle$, for some $\langle \rho', S_2'\rangle$ such that $R(\rho', S_1', S_2')$;
(2) if $\langle \rho, S_2\rangle \overset{a}{\rightarrow}_B \langle \rho', S_2'\rangle$ then $\langle \rho, S_1\rangle \overset{\hat{a}}{\Rightarrow}_B \langle \rho', S_1'\rangle$, 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 $\rho \in \mathrm{Store}$, denoted by $S_1 \approx_\rho S_2$, if $R(\rho, S_1, S_2)$ for some bisimulation R. □

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

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

$$\alpha_{sc}^{\rho}(X) \triangleq \{sc(\tau) \in \mathrm{Store}^* \mid \tau \in X, \exists S, \tau'.\, \tau = \langle \rho, S\rangle\tau'\}.$$

It is worth remarking that α_{sc}^{ρ} is a weaker abstraction than α_{sc} defined in Section 6, that is, for any $X, Y \in \wp(\mathrm{Trace}^{GP})$, $\alpha_{sc}(X) = \alpha_{sc}(Y) \Rightarrow \alpha_{sc}^{\rho}(X) = \alpha_{sc}^{\rho}(Y)$ (while the converse does not hold in general).

THEOREM 10.6. *For any* $S_1, S_2 \in \mathrm{Stm}$, $\rho \in \mathrm{Store}$, *we have that* $S_1 \approx_\rho S_2$ *iff* $\alpha_{sc}^{\rho}(\mathbf{T}_{GP}[\![S_1]\!]) = \alpha_{sc}^{\rho}(\mathbf{T}_{GP}[\![S_2]\!])$.

PROOF. ($\Rightarrow$): We prove that if $R(\rho, S_1, S_2)$ holds for some bisimulation R then $\alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_1 \rrbracket) \subseteq \alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_2 \rrbracket)$ (the reverse containment is symmetric), that is, if $sc(\tau) \in \mathrm{Store}^*$ for some $\tau \in \mathbf{T}_{GP}\llbracket S_1 \rrbracket$ such that $\tau = \langle \rho, S_1 \rangle \tau'$ then there exists some $\psi \in \mathbf{T}_{GP}\llbracket S_2 \rrbracket$ such that $\psi = \langle \rho, S_2 \rangle \psi'$ and $sc(\tau) = sc(\psi)$. Let us then consider $\tau \in \mathbf{T}_{GP}\llbracket S_1 \rrbracket$ such that $\tau = \langle \rho, S_1 \rangle \tau'$. If $\tau' = \epsilon$ then we pick $\langle \rho, S_2 \rangle \in \mathbf{T}_{GP}\llbracket S_2 \rrbracket$ so that $sc(\langle \rho, S_1 \rangle) = \rho = sc(\langle \rho, S_2 \rangle)$. Otherwise, $\tau = \langle \rho, S_1 \rangle \tau' \in \mathbf{T}_{GP}\llbracket S_1 \rrbracket$, with $\epsilon \neq \tau' = \tau'' \langle \mu, S \rangle$. We prove by induction on $|\tau'| \geq 1$ that there exists $\psi = \langle \rho, S_2 \rangle \psi'' \langle \mu, T \rangle \in \mathbf{T}_{GP}\llbracket S_2 \rrbracket$ such that $sc(\tau) = sc(\psi)$ and $R(\mu, S, T)$.
($|\tau'| = 1$): In this case, $\tau = \langle \rho, S_1 \rangle \langle \mu, S \rangle \in \mathbf{T}_{GP}\llbracket S_1 \rrbracket$, so that $\langle \rho, S_1 \rangle \xrightarrow{a}_B \langle \mu, S \rangle$. Since, by hypothesis, $R(\rho, S_1, S_2)$ holds, we have that $\langle \rho, S_2 \rangle \overset{\hat{a}}{\Rightarrow}_B \langle \mu, T \rangle$, for some T, and $R(\mu, S, T)$. Let $\psi \in \mathbf{T}_{GP}\llbracket S_2 \rrbracket$ be the trace corresponding to the sequence of transitions $\langle \rho, S_2 \rangle \overset{\hat{a}}{\Rightarrow}_B \langle \mu, T \rangle$. Then, by definition of $\overset{\hat{a}}{\Rightarrow}_B$, we have that $sc(\tau) = sc(\psi)$, and, by definition of bisimulation, $R(\mu, S, T)$ holds.
($|\tau'| > 1$): Here, $\tau' = \tau'' \langle \mu, S \rangle$ and $\tau = \langle \rho, S_1 \rangle \tau' \in \mathbf{T}_{GP}\llbracket S_1 \rrbracket$, with $|\tau''| = |\tau'| - 1 \geq 1$. Hence, $\tau'' = \tau''' \langle \eta, U \rangle$. By inductive hypothesis, there exists $\psi = \langle \rho, S_2 \rangle \psi'' \langle \eta, V \rangle \in \mathbf{T}_{GP}\llbracket S_2 \rrbracket$ such that $sc(\langle \rho, S_1 \rangle \tau''' \langle \eta, U \rangle) = sc(\langle \rho, S_2 \rangle \psi'' \langle \eta, V \rangle)$ and $R(\eta, U, V)$. Since $\langle \eta, U \rangle \xrightarrow{a}_B \langle \mu, S \rangle$ and $R(\eta, U, V)$ holds, we obtain that $\langle \eta, V \rangle \overset{\hat{a}}{\Rightarrow}_B \langle \mu, T \rangle$, for some T, and $R(\mu, S, T)$ holds. Let $\langle \eta, V \rangle \cdots \langle \mu, T \rangle$ be the sequence of states corresponding to the sequence of transitions $\langle \eta, V \rangle \overset{\hat{a}}{\Rightarrow}_B \langle \mu, T \rangle$ so that we pick $\langle \rho, S_2 \rangle \psi'' \langle \eta, V \rangle \cdots \langle \mu, T \rangle \in \mathbf{T}_{GP}\llbracket S_2 \rrbracket$. The condition $R(\mu, S, T)$ already holds. Moreover, by definition of $\overset{\hat{a}}{\Rightarrow}_B$, we have that $sc(\langle \eta, U \rangle \langle \mu, S \rangle) = sc(\langle \eta, V \rangle \cdots \langle \mu, T \rangle)$, and therefore we obtain $sc(\tau) = sc(\langle \rho, S_1 \rangle \tau''' \langle \eta, U \rangle \langle \mu, S \rangle) = sc(\langle \mu, S_2 \rangle \psi'' \langle \eta, V \rangle \cdots \langle \mu, T \rangle)$.

($\Leftarrow$): We first observe the following property ($*$), which is a straight consequence of the fact that $\rightarrow_B$ is a deterministic relation: If $S \in \mathrm{Stm}$ and $\sigma, \tau \in \mathbf{T}\llbracket S \rrbracket$ are such that $\sigma_0 = \langle \mu, S \rangle = \tau_0$ and $|\tau| \leq |\sigma|$ then there exists some ψ such that $\sigma = \tau\psi$.
Given $\rho \in \mathrm{Store}$, we assume that $\alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_1 \rrbracket) = \alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_2 \rrbracket)$ and we then define the following relation R:

$$R \triangleq \{(\rho, S_1, S_2)\} \cup \{(\mu, T_1, T_2) \mid \langle \rho, S_1 \rangle \cdots \langle \mu, T_1 \rangle \in \mathbf{T}\llbracket S_1 \rrbracket,\ \langle \rho, S_2 \rangle \cdots \langle \mu, T_1 \rangle \in \mathbf{T}\llbracket S_2 \rrbracket,$$
$$sc(\langle \rho, S_1 \rangle \cdots \langle \mu, T_1 \rangle) = sc(\langle \rho, S_2 \rangle \cdots \langle \mu, T_1 \rangle)\}.$$

We show that R is a bisimulation, so that $R(\rho, S_1, S_2)$ follows.
(case A) Assume that $\langle \rho, S_1 \rangle \xrightarrow{a}_B \langle \rho', S'_1 \rangle$. Then, since $\langle \rho, S_1 \rangle \langle \rho', S'_1 \rangle \in \mathbf{T}\llbracket S_1 \rrbracket$ and $\alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_1 \rrbracket) = \alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_2 \rrbracket)$, we have that there exists $\tau = \langle \rho, S_2 \rangle \cdots \in \mathbf{T}\llbracket S_2 \rrbracket$ such that $sc(\langle \rho, S_1 \rangle \langle \rho', S'_1 \rangle) = sc(\tau)$. Hence, τ necessarily has the following shape:

$$\tau = \langle \rho, S_2 \rangle \langle \rho, U_1 \rangle \cdots \langle \rho, U_n \rangle \langle \rho', V_1 \rangle \cdots \langle \rho', V_m \rangle$$

where $n \geq 0$ ($n = 0$ means that $\langle \rho, U_1 \rangle \cdots \langle \rho, U_n \rangle$ is indeed the empty sequence) and $m \geq 1$. This therefore means that $\langle \rho, S_2 \rangle \overset{\hat{a}}{\Rightarrow}_B \langle \rho', V_m \rangle$, 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 $\delta = \langle \rho, S_1 \rangle \cdots \langle \mu, T_1 \rangle \in \mathbf{T}\llbracket S_1 \rrbracket$, $\sigma = \langle \rho, S_2 \rangle \cdots \langle \mu, T_2 \rangle \in \mathbf{T}\llbracket S_2 \rrbracket$ and $sc(\delta) = sc(\sigma)$. Hence, let us suppose that $\langle \mu, T_1 \rangle \xrightarrow{a}_B \langle \mu', T'_1 \rangle$. Then, since $\delta \langle \mu', T'_1 \rangle \in \mathbf{T}\llbracket S_1 \rrbracket$ and $\alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_1 \rrbracket) = \alpha^{\rho}_{sc}(\mathbf{T}_{GP}\llbracket S_2 \rrbracket)$, we have that there exists $\tau = \langle \rho, S_2 \rangle \cdots \in \mathbf{T}\llbracket S_2 \rrbracket$ such that $sc(\delta \langle \mu', T'_1 \rangle) = sc(\tau)$.

(case B1). If $|\tau| \leq |\sigma|$ then, by the property ($*$) above, $\sigma = \tau\psi$, for some ψ. Hence, $sc(\tau) = sc(\delta \langle \mu', T'_1 \rangle)$ is a prefix of $sc(\sigma) = sc(\delta)$. Consequently, $sc(\delta \langle \mu', T'_1 \rangle)$ can be a prefix of $sc(\delta)$ only if $sc(\delta \langle \mu', T'_1 \rangle) = sc(\delta)$, so that the action a is τ and $\mu' = \mu$, that is, $\langle \mu, T_1 \rangle \xrightarrow{\tau}_B \langle \mu, T'_1 \rangle$. We thus consider the empty transition sequence

$$(T_1)\quad \langle \rho, (\mathbf{if}\ B\ \mathbf{then}\ (S\,\mathbf{while}\ B\ \mathbf{do}\ S))\,K\rangle \to_T \langle \rho, (\mathbf{while}\ B\ \mathbf{do}\ S)K, \epsilon, S(\mathbf{while}\ B\ \mathbf{do}\ S)K\rangle \qquad \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{true}$$

$$(T_2)\quad \langle \rho, K_w, t, \mathbf{skip}; K\rangle \to_T \langle \rho, K_w, t(\mathbf{skip};), K\rangle$$

$$(T_3)\quad \langle \rho, K_w, t, x := E; K\rangle \to_T \langle \rho[x/\mathbf{E}[\![E]\!]\rho], K_w, t(x := E;), K\rangle$$

$$(T_4)\quad \langle \rho, K_w, t, (\mathbf{if}\ B\ \mathbf{then}\ S)K\rangle \to_T \begin{cases} \langle \rho, K_w, t(\mathbf{bail}\ B\ \mathbf{to}\ (SK)), K\rangle & \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{false} \\ \langle \rho, K_w, t(\mathbf{bail}\ \neg B\ \mathbf{to}\ K), SK\rangle & \text{if } \mathbf{B}[\![B]\!]\rho = \mathit{true} \end{cases}$$

$$(T_5)\quad \langle \rho, K_w, t, (\mathbf{while}\ B\ \mathbf{do}\ S)K\rangle \to_T \begin{cases} \langle \rho, K_w, t(\mathbf{skip};), (\mathbf{if}\ B\ \mathbf{then}\ (S\,\mathbf{while}\ B\ \mathbf{do}\ S))\,K\rangle & \\ \qquad\qquad\qquad\qquad \text{if } K_w \not\equiv (\mathbf{while}\ B\ \mathbf{do}\ S)K & \\ \langle \rho, O(\mathbf{while}\ B\ \mathbf{do}\ t, \rho)K\rangle \quad \text{if } K_w \equiv (\mathbf{while}\ B\ \mathbf{do}\ S)K & \end{cases}$$

$$(T_6)\quad \langle \rho, K_w, t, S\rangle \to_T \langle \rho', S'\rangle \quad \text{if } K_w \not\equiv S \text{ and } \langle \rho, S\rangle \to_B \langle \rho', S'\rangle$$

Fig. 7. Definition of the tracing relation $\to_T$.

$\langle \mu, T_2\rangle \overset{\hat{\tau}}{\Rightarrow} \langle \mu, T_2\rangle$, so that from $sc(\delta\langle \mu, T_1'\rangle) = sc(\sigma)$, by definition of R we obtain that $R(\mu, T_1', T_2)$ holds.
(case B2). If $|\tau| > |\sigma|$ then, by $(*)$ above, $\tau = \sigma\psi$, for some ψ, i.e., $\tau = \sigma \cdots \langle \mu'', T_2'\rangle$, for some μ'' and T_2'. Since $sc(\langle \rho, S_2\rangle \cdots \langle \mu, T_2\rangle) = sc(\langle \rho, S_1\rangle \cdots \langle \mu, T_1\rangle)$ and $sc(\langle \rho, S_2\rangle \cdots \langle \mu, T_2\rangle \cdots \langle \mu'', T_2'\rangle) = sc(\langle \rho, S_1\rangle \cdots \langle \mu, T_1\rangle\langle \mu', T_1'\rangle)$, we derive that $\mu'' = \mu'$ and $\langle \mu, T_2\rangle \overset{a}{\Rightarrow} \langle \mu'' = \mu', T_2'\rangle$. By definition of R, $R(\mu', T_1', T_2')$ holds.

This closes the proof. □

10.4. Hot Paths

Let us recall the set of rules that define the tracing transitions in Guo and Palsberg [2011] model. Let $\mathrm{tState}_{GP} \triangleq \mathrm{Store} \times \mathrm{Stm} \times \mathrm{Stm} \times \mathrm{Stm}$ 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, $\mathrm{State}^e_{GP} \triangleq \mathrm{State}_{GP} \cup \mathrm{tState}_{GP}$ denotes the corresponding extended notion of state, which encompasses the trace recording mode. Then, the relation $\to_T \subseteq \mathrm{State}^e_{GP} \times \mathrm{State}^e_{GP}$ is defined by the clauses in Figure 7, where $O : \mathrm{Stm} \times \mathrm{Store} \to \mathrm{Stm}$ is a “sound” optimization function that depends on a given store. Correspondingly, the trace semantics $\mathbf{T}_{GP}[\![S]\!] \subseteq (\mathrm{State}^e_{GP})^+$ of a program $S \in \mathrm{Stm}$ is naturally extended to the relation $\to_{B,T} \triangleq \to_B \cup \to_T \subseteq \mathrm{State}^e_{GP} \times \mathrm{State}^e_{GP}$.

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 **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 9, nested hot paths are not allowed.

(vi) Observe that when tracing a loop **while** B **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 **do** $t \equiv$ **while** B **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, $O(\textbf{while } B \textbf{ do } t, \rho) = \textbf{while } B \textbf{ do } t$.

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

$$\langle \rho, Q\rangle \to^*_{B,T} \langle \rho', (\textbf{while } B \textbf{ do } S)K\rangle \to^*_{B,T} \langle \rho'', (\textbf{while } B \textbf{ do } S)K, t, (\textbf{while } B \textbf{ do } S)K\rangle.$$

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

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

Thus, $\mathit{loop}_{GP}(\tau)$ contains sequences of commands without store. The map $\alpha^{GP}_{hot} : \wp(\mathrm{Trace}) \to \wp(\mathbb{C}^+)$ then lifts loop_{GP} to sets of traces as usual: $\alpha^{GP}_{hot}(T) \triangleq \cup_{\tau \in T}\, \mathit{loop}_{GP}(\tau)$. Then, let us consider a GP hot path t as recorded by a transition sequence τ:

$$\begin{aligned} \tau \triangleq\ & \langle \rho, S_0 \equiv (\textbf{while } B \textbf{ do } S)K\rangle \to_B \\ & \langle \rho, S_1 \equiv (\textbf{if } B \textbf{ then } (S\, \textbf{while } B \textbf{ do } S))\, K\rangle \to_T \\ & \langle \rho, (\textbf{while } B \textbf{ do } S)K, \epsilon, S_2 \equiv S(\textbf{while } B \textbf{ do } S)K\rangle \to_T \\ & \cdots \to_T \\ & \langle \rho', (\textbf{while } B \textbf{ do } S)K, t', S_n\rangle \to_T \\ & \langle \rho'', (\textbf{while } B \textbf{ do } S)K, t, S_{n+1} \equiv (\textbf{while } B \textbf{ do } S)K\rangle \end{aligned} \tag{‡}$$

where $\mathbf{B}[\![B]\!]\rho = \mathit{true}$. Hence, the S_i's occurring in τ are the current statements to be evaluated. With a negligible abuse of notation, we assume that $\tau \in \mathbf{T}_{GP}[\![(\textbf{while } B \textbf{ do } S)K]\!]$, that is, the arrow symbols $\to_B$ and $\to_T$ are taken out of the sequence τ. By Lemma 10.3 (2), we therefore consider the corresponding execution trace $\mathcal{C}^t(\tau)$ of the compiled program $\mathcal{C}((\textbf{while } B \textbf{ do } S)K)$, where the state compile function $\mathcal{C}^s$ in Figure 6, when applied to states in trace recording mode, is assumed to act on the current store and the program to be evaluated, that is, $\mathcal{C}^s(\langle \rho, K_w, t, S\rangle) = \mathcal{C}^s(\langle \rho, S\rangle)$. We thus obtain:

$$\begin{aligned} \mathcal{C}^t(\tau) \triangleq\ & \langle \rho, C_0 \equiv \boldsymbol{l}((\textbf{while } B \textbf{ do } S)K) : \mathrm{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (S\, \textbf{while } B \textbf{ do } S))\, K)\rangle \\ & \langle \rho, C_1 \equiv \boldsymbol{l}((\textbf{if } B \textbf{ then } (S\, \textbf{while } B \textbf{ do } S))\, K) : B \to \boldsymbol{l}(S\, (\textbf{while } B \textbf{ do } S)K)\rangle \\ & \langle \rho, C_2 \equiv \boldsymbol{l}(S(\textbf{while } B \textbf{ do } S)K) : A_2 \to \boldsymbol{l}(T)\rangle \\ & \cdots \\ & \langle \rho', C_n \equiv \boldsymbol{l}(S_n) : A_n \to \boldsymbol{l}((\textbf{while } B \textbf{ do } S)\, K)\rangle \\ & \langle \rho'', C_{n+1} \equiv \boldsymbol{l}((\textbf{while } B \textbf{ do } S)K) : \mathrm{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (S\, \textbf{while } B \textbf{ do } S))\, K)\rangle. \end{aligned}$$

We therefore obtain a hot path $hp_t = C_0 C_1 \cdots C_n \in \mathit{loop}_{GP}(\mathcal{C}^t(\tau))$, i.e. $hp_t \in \alpha^{GP}_{hot}(\mathbf{T}^\iota[\![\mathcal{C}((\textbf{while } B \textbf{ do } S)K)]\!])$, where $\mathit{lbl}(C_0) = \boldsymbol{l}((\textbf{while } B \textbf{ do } S)K) = \mathit{suc}(C_n)$. 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 (T_5), S_{n+1} is necessarily the first occurrence of $(\textbf{while } B \textbf{ do } S))\,K$ as current program to be evaluated in the trace τ, so that, for any $k \in (0, n]$, $lbl(C_k) \neq \boldsymbol{l}((\textbf{while } B \textbf{ do } S)K)$. We have thus shown that any GP hot path arising from a trace τ generates a corresponding hot path extracted by our selection map $loop_{GP}$ on the compiled trace $\mathcal{C}^t(\tau)$:

LEMMA 10.7. *Let $Q_w \equiv (\textbf{while } B \textbf{ do } S)K$. If t is a GP hot path for Q_w where $\tau \equiv \langle \rho, Q_w \rangle \to^*_{B,T} \langle \rho', Q_w, t, Q_w \rangle$ is the transition sequence (‡) that records t, then there exists a hot path $hp_t = C_0 C_1 \cdots C_n \in \alpha^{GP}_{hot}(\mathbf{T}^\iota \llbracket \mathcal{C}(Q_w) \rrbracket)$ such that, for any $i \in [0, n]$, $lbl(C_i) = \boldsymbol{l}(S_i)$, and, in particular, $lbl(C_0) = \boldsymbol{l}(Q_w) = suc(C_n)$.*

Example 10.8. Let us consider the while statement Q_w of the program in Example 2.1:

$$Q_w \equiv \textbf{while } (x \leq 20) \textbf{ do } (x := x + 1;\ (\textbf{if } (x\%3 = 0) \textbf{ then } x := x + 3;))$$

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 7 yield the following trace t for Q_w:

$$t \equiv x := x + 1;\ \textbf{bail } (x\%3 = 0) \textbf{ to } (x := x + 3;\ Q_w).$$

On the other hand, the compiled program $\mathcal{C}(Q_w) \in \wp(\mathbb{C})$ is as follows:

$$\begin{aligned}
\mathcal{C}(Q_w) = \big\{ & D_0 \equiv \boldsymbol{l}_{\textbf{while}} : \text{skip} \to \boldsymbol{l}_{\textbf{ifwhile}}, \\
& D_1 \equiv \boldsymbol{l}_{\textbf{ifwhile}} : (x \leq 20) \to \boldsymbol{l}_1,\ D_1^c \equiv \boldsymbol{l}_{\textbf{ifwhile}} : \neg(x \leq 20) \to \boldsymbol{l}_\epsilon, \\
& D_2 \equiv \boldsymbol{l}_1 : x := x + 1 \to \boldsymbol{l}_{\textbf{if}}, \\
& D_3 \equiv \boldsymbol{l}_{\textbf{if}} : (x\%3 = 0) \to \boldsymbol{l}_2,\ D_3^c \equiv \boldsymbol{l}_{\textbf{if}} : \neg(x\%3 = 0) \to \boldsymbol{l}_{\textbf{while}}, \\
& D_4 \equiv \boldsymbol{l}_2 : x := x + 3 \to \boldsymbol{l}_{\textbf{while}},\ D_5 \equiv \boldsymbol{l}_\epsilon : \text{skip} \to \text{Ł} \big\},
\end{aligned}$$

where labels have the following meaning:

$$\begin{aligned}
\boldsymbol{l}_{\textbf{while}} &\triangleq \boldsymbol{l}(Q_w) \\
\boldsymbol{l}_{\textbf{ifwhile}} &\triangleq \boldsymbol{l}(\textbf{if } (x \leq 20) \textbf{ then } (x := x + 1; (\textbf{if } (x\%3 = 0) \textbf{ then } x := x + 3;)\ Q_w)) \\
\boldsymbol{l}_1 &\triangleq \boldsymbol{l}(x := x + 1; (\textbf{if } (x\%3 = 0) \textbf{ then } x := x + 3;)\ Q_w)) \\
\boldsymbol{l}_{\textbf{if}} &\triangleq \boldsymbol{l}((\textbf{if } (x\%3 = 0) \textbf{ then } x := x + 3;)\ Q_w)) \\
\boldsymbol{l}_2 &\triangleq \boldsymbol{l}(x := x + 3; Q_w).
\end{aligned}$$

Hence, in correspondence with the trace t, we obtain the hot path $hp_t = D_0 D_1 D_2 D_3^c \in \alpha^{GP}_{hot}(\mathbf{T}^\iota \llbracket \mathcal{C}(Q_w) \rrbracket)$. In turn, this hot path hp_t corresponds to the 2-hot path hp_1 consisting of the analogous sequence of commands, which has been selected in Example 4.1. □

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 $Q_w \equiv (\textbf{while } B \textbf{ do } S)K \in \mathrm{Stm}$ and its compilation $P_w \triangleq \mathcal{C}(Q_w) \in \wp(\mathbb{C})$. Observe that, by Definition 10.1 of compilation $\mathcal{C}$, a hot path $C_0 \cdots C_n \in \alpha^{GP}_{hot}(\mathbf{T}\llbracket P_w \rrbracket)$ for the compiled program P_w always arises in correspondence with some while loop $\textbf{while } B' \textbf{ do } S'$ occurring in Q_w and therefore has necessarily the following shape:

$$\begin{aligned}
&C_0 \equiv \boldsymbol{l}((\textbf{while } B' \textbf{ do } S')J) : \text{skip} \to \boldsymbol{l}((\textbf{if } B' \textbf{ then } (S' \textbf{ while } B' \textbf{ do } S'))\,J)\\
&C_1 \equiv \boldsymbol{l}((\textbf{if } B' \textbf{ then } (S' \,(\textbf{while } B' \textbf{ do } S')))\,J) : B' \to \boldsymbol{l}(S' \,(\textbf{while } B' \textbf{ do } S')\,J)\\
&C_2 \equiv \boldsymbol{l}(S' \,(\textbf{while } B' \textbf{ do } S')\,J) : A_2 \to \boldsymbol{l}(T_3)\\
&\quad \cdots\\
&C_n \equiv \boldsymbol{l}(T_n) : A_n \to \boldsymbol{l}((\textbf{while } B' \textbf{ do } S')J)
\end{aligned}$$

The GP hot path extraction scheme for Q_w described by the rules in Figure 7 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* $extr^{GP}_{hp}(P_w)$ of P_w for the hot path $hp = C_0C_1 \cdots C_n \in \alpha^{GP}_{hot}(\mathbf{T}[\![P_w]\!])$ is defined as follows:

(1) If for any $i \in [2, n]$, $cmpl(C_i) \notin P_w$ then $extr^{GP}_{hp}(P_w) \triangleq P_w$;
(2) Otherwise:

$$\begin{aligned}
extr^{GP}_{hp}(P_w) \triangleq P_w &\cup \{\ell_i : act(C_i) \to \ell_{next(i)} \mid i \in [0, n]\}\\
&\cup \{\ell_i : \neg act(C_i) \to L^c_{next(i)} \mid i \in [0, n],\, cmpl(C_i) \in P_w\}. \qquad \square
\end{aligned}$$

Clearly, $extr^{GP}_{hp}(P)$ 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 10.4.

Example 10.10. Let us consider the programs Q_w and $\mathcal{C}(Q_w)$ of Example 10.8 and the hot path $hp_t = D_0D_1D_2D^c_3 \in \alpha^{GP}_{hot}(\mathbf{T}[\![\mathcal{C}(Q_w)]\!])$ which corresponds to the trace $t \equiv x := x + 1;$ **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:

$$\begin{aligned}
extr^{GP}_{hp_t}(\mathcal{C}(Q_w)) \triangleq \big\{&D_0 \equiv \boldsymbol{l}_{\textbf{while}} : \text{skip} \to \boldsymbol{l}_{\textbf{ifwhile}},\, D_1 \equiv \boldsymbol{l}_{\textbf{ifwhile}} : (x \le 20) \to \boldsymbol{l}_1,\\
&D^c_1 \equiv \boldsymbol{l}_{\textbf{ifwhile}} : \neg(x \le 20) \to \boldsymbol{l}_\epsilon,\, D_2 \equiv \boldsymbol{l}_1 : x := x + 1 \to \boldsymbol{l}_{\textbf{if}},\\
&D_3 \equiv \boldsymbol{l}_{\textbf{if}} : (x\%3 = 0) \to \boldsymbol{l}_2,\, D^c_3 \equiv \boldsymbol{l}_{\textbf{if}} : \neg(x\%3 = 0) \to \boldsymbol{l}_{\textbf{while}},\\
&D_4 \equiv \boldsymbol{l}_2 : x := x + 3 \to \boldsymbol{l}_{\textbf{while}},\, D_5 \equiv \boldsymbol{l}_\epsilon : \text{skip} \to \text{Ł}\big\} \cup\\
\big\{&\ell_0 : \text{skip} \to \ell_1,\, \ell_1 : x \le 20 \to \ell_2,\, \ell_1 : \neg(x \le 20) \to \boldsymbol{l}_\epsilon,\\
&\ell_2 : x := x + 1 \to \ell_3,\, \ell_3 : \neg(x\%3 = 0) \to \ell_0,\, \ell_3 : (x\%3 = 0) \to \boldsymbol{l}_2\big\}.
\end{aligned}$$

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

```
while (x ≤ 20) do
      x := x + 1;
      bail (x%3 = 0) to (x := x + 3; Q_w)
```

whose compilation yields the following program:

$$\begin{aligned}
\mathcal{C}(Q_t) = \big\{&\boldsymbol{l}_{\textbf{while}_t} : \text{skip} \to \boldsymbol{l}_{\textbf{ifwhile}_t},\, \boldsymbol{l}_{\textbf{ifwhile}_t} : (x \le 20) \to \boldsymbol{l}_{1t},\, \boldsymbol{l}_{\textbf{ifwhile}_t} : \neg(x \le 20) \to \boldsymbol{l}_\epsilon,\\
&\boldsymbol{l}_{1t} : x := x + 1 \to \boldsymbol{l}_{\textbf{bail}},\, \boldsymbol{l}_{\textbf{bail}} : (x\%3 = 0) \to \boldsymbol{l}_{\textbf{bail}_{\mathit{true}}},\, \boldsymbol{l}_{\textbf{bail}} : \neg(x\%3 = 0) \to \boldsymbol{l}_{\textbf{while}_t},\\
&\boldsymbol{l}_{\textbf{while}} : \text{skip} \to \boldsymbol{l}_{\textbf{ifwhile}},\, \boldsymbol{l}_{\textbf{ifwhile}} : (x \le 20) \to \boldsymbol{l}_1,\, \boldsymbol{l}_{\textbf{ifwhile}} : \neg(x \le 20) \to \boldsymbol{l}_\epsilon,\\
&\boldsymbol{l}_1 : x := x + 1 \to \boldsymbol{l}_{\textbf{if}},\, \boldsymbol{l}_{\textbf{if}} : (x\%3 = 0) \to \boldsymbol{l}_2,\, \boldsymbol{l}_{\textbf{if}} : \neg(x\%3 = 0) \to \boldsymbol{l}_{\textbf{while}},\\
&\boldsymbol{l}_{\textbf{bail}_{\mathit{true}}} : x := x + 3 \to \boldsymbol{l}_{\textbf{while}},\, \boldsymbol{l}_\epsilon : \text{skip} \to \text{Ł}\big\}
\end{aligned}$$

with the following new labels:

$$
\begin{aligned}
\boldsymbol{l}_{\mathbf{while}_t} &\triangleq \boldsymbol{l}(\mathbf{while}\ (x \leq 20)\ t)\\
\boldsymbol{l}_{\mathbf{ifwhile}_t} &\triangleq \boldsymbol{l}(\mathbf{if}\ (x \leq 20)\ \mathbf{then}\ (t\ (\mathbf{while}\ (x \leq 20)\ t)))\\
\boldsymbol{l}_{1t} &\triangleq \boldsymbol{l}(t\ (\mathbf{while}\ (x \leq 20)\ t))\\
\boldsymbol{l}_{\mathbf{bail}} &\triangleq \boldsymbol{l}((\mathbf{bail}\ (x\%3 = 0)\ \mathbf{to}\ (x := x + 3; Q_w))(\mathbf{while}\ (x \leq 20)\ t))
\end{aligned}
$$

while observe that $\boldsymbol{l}_{\mathbf{bail}_{true}} \triangleq \boldsymbol{l}(x := x + 3; Q_w) = \boldsymbol{l}_2$. It is then immediate to check that the programs $\mathcal{C}(Q_t)$ and $extr^{GP}_{hp_t}(\mathcal{C}(Q_w))$ are equal up to the following label renaming of $extr^{GP}_{hp_t}(\mathcal{C}(Q_w))$:

$$\{\ell_0 \mapsto \boldsymbol{l}_{\mathbf{while}_t}, \ell_1 \mapsto \boldsymbol{l}_{\mathbf{ifwhile}_t}, \ell_2 \mapsto \boldsymbol{l}_{1t}, \ell_3 \mapsto \boldsymbol{l}_{\mathbf{bail}}\}. \quad \square$$

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 $\langle \rho, (\mathbf{while}\ B\ \mathbf{do}\ S)K \rangle \to^*_{B,T} \langle \rho', (\mathbf{while}\ B\ \mathbf{do}\ S)K, t, (\mathbf{while}\ B\ \mathbf{do}\ S)K \rangle$ and let $hp_t \in \alpha^{GP}_{hot}(\mathbf{T}^{\iota}[\![\mathcal{C}((\mathbf{while}\ B\ \mathbf{do}\ S)K)]\!])$ be the corresponding GP hot path as determined by Lemma 10.7. Then, $\mathcal{C}((\mathbf{while}\ B\ \mathbf{do}\ t)K) \cong extr^{GP}_{hp_t}(\mathcal{C}((\mathbf{while}\ B\ \mathbf{do}\ S)K))$.*

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

$$
\begin{array}{ll}
\langle \rho, S_{-2} \equiv (\mathbf{while}\ B\ \mathbf{do}\ S)K \rangle \to_B & \\
\langle \rho, S_{-1} \equiv (\mathbf{if}\ B\ \mathbf{then}\ (S\ \mathbf{while}\ B\ \mathbf{do}\ S))\ K \rangle \to_T & [\text{with } \mathbf{B}[\![B]\!]\rho = \textit{true}]\\
\langle \rho_0 \triangleq \rho, (\mathbf{while}\ B\ \mathbf{do}\ S)K, t_0 \equiv \epsilon, S_0 \equiv S(\mathbf{while}\ B\ \mathbf{do}\ S)K \rangle \to_T & \\
\langle \rho_1, (\mathbf{while}\ B\ \mathbf{do}\ S)K, t_1 \equiv c_1, S_1 \rangle \to_T & \\
\cdots \to_T & \\
\langle \rho_n, (\mathbf{while}\ B\ \mathbf{do}\ S)K, t_n \equiv t_{n-1}c_n, S_n \rangle \to_T & \\
\langle \rho_{n+1} \triangleq \rho', (\mathbf{while}\ B\ \mathbf{do}\ S)K, t \equiv t_n c_{n+1}, S_{n+1} \equiv (\mathbf{while}\ B\ \mathbf{do}\ S)K \rangle &
\end{array}
$$

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, $t = c_1 ... c_n c_{n+1}$, for some commands $c_i \in \text{Cmd}$, and the corresponding hot path $hp_t \equiv H_{-2}H_{-1}H_0...H_n$ as determined by Lemma 10.7 is as follows:

$$
\begin{aligned}
H_{-2} &\triangleq \boldsymbol{l}(S_{-2}) : \text{skip} \to \boldsymbol{l}(S_{-1})\\
H_{-1} &\triangleq \boldsymbol{l}(S_{-1}) : B \to \boldsymbol{l}(S_0) && [\text{because } \mathbf{B}[\![B]\!]\rho = \textit{true}]\\
H_0 &\triangleq \boldsymbol{l}(S_0) : A_0 \to \boldsymbol{l}(S_1)\\
H_1 &\triangleq \boldsymbol{l}(S_1) : A_1 \to \boldsymbol{l}(S_2)\\
&\cdots\\
H_n &\triangleq \boldsymbol{l}(S_n) : A_n \to \boldsymbol{l}(S_{-2})
\end{aligned}
$$

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 $(*)$):

(1) $S_i \equiv \mathbf{skip}; J \quad \Rightarrow \quad A_i \equiv \text{skip}\ \ \&\ \ c_{i+1} \equiv \mathbf{skip};\ \ \&\ \ S_{i+1} \equiv J$
(2) $S_i \equiv x := E; J \quad \Rightarrow \quad A_i \equiv x := E\ \ \&\ \ c_{i+1} \equiv x := E;\ \ \&\ \ S_{i+1} \equiv J$
(3) $S_i \equiv (\mathbf{if}\ B'\ \mathbf{then}\ S')J\ \ \&\ \ \mathbf{B}[\![B']\!]\rho_i = \textit{true} \quad \Rightarrow$
$A_i \equiv B'\ \ \&\ c_{i+1} \equiv \mathbf{bail}\ \neg B'\ \mathbf{to}\ J\ \&\ S_{i+1} \equiv S'J$

(4) $S_i \equiv (\textbf{if } B' \textbf{ then } S')J \;\;\&\;\; \mathbf{B}[\![B']\!]\rho_i = \mathit{false} \quad\Rightarrow$
$A_i \equiv \neg B' \;\&\; c_{i+1} \equiv \textbf{bail } B' \textbf{ to } (S'J) \;\&\; S_{i+1} \equiv J$

(5) $S_i \equiv (\textbf{while } B' \textbf{ do } S')J \;\;\&\;\; (\textbf{while } B' \textbf{ do } S')J \neq (\textbf{while } B \textbf{ do } S)K \quad\Rightarrow$
$A_i \equiv \text{skip} \;\;\&\;\; c_{i+1} \equiv \textbf{skip};\;\;\&\;\; S_{i+1} \equiv (\textbf{if } B' \textbf{ then } (S'(\textbf{while } B' \textbf{ do } S')))J$

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 $\mathit{extr}^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K)) = \mathcal{C}((\textbf{while } B \textbf{ do } S)K)$. 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, $(\textbf{while } B \textbf{ do } t)K \equiv (\textbf{while } B \textbf{ do } S)K$, so that the thesis follows trivially.
Thus, we assume that H_k, with $k \in [0, n]$, is the first conditional command occuring in the sequence $H_0...H_n$. Case (2) of Definition 10.9 applies, so that:

$$\begin{aligned}\mathit{extr}^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K)) = \mathcal{C}((\textbf{while } B \textbf{ do } S)K) \cup&\\ \{\ell_{-2} : \text{skip} \to \ell_{-1},\, \ell_{-1} : B \to \ell_0,\, \ell_{-1} : \neg B \to \boldsymbol{l}(K),&\\ \ell_0 : A_0 \to \ell_1, ..., \ell_n : A_n \to \ell_{-2}\} \cup&\\ \{\ell_i : \neg A_i \to \boldsymbol{l}(S_{\mathit{next}(i)})^c \mid i \in [0, n],\, A_i \in \text{BExp}\}.&\end{aligned}$$

Moreover, we have that:

$$\begin{aligned}\mathcal{C}((\textbf{while } B \textbf{ do } S)K) = &\\ &\big\{\boldsymbol{l}((\textbf{while } B \textbf{ do } S)K) : \text{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (S\,(\textbf{while } B \textbf{ do } S)))\,K),\\ &\;\boldsymbol{l}((\textbf{if } B \textbf{ then } (S\,(\textbf{while } B \textbf{ do } S)))\,K) : B \to \boldsymbol{l}(S(\textbf{while } B \textbf{ do } S)K),\\ &\;\boldsymbol{l}((\textbf{if } B \textbf{ then } (S\,(\textbf{while } B \textbf{ do } S)))\,K) : \neg B \to \boldsymbol{l}(K)\big\}\\ &\cup \mathcal{C}(S(\textbf{while } B \textbf{ do } S)K) \cup \mathcal{C}(K)\end{aligned}$$

$$\begin{aligned}\mathcal{C}((\textbf{while } B \textbf{ do } t)K) = &\\ &\big\{\boldsymbol{l}((\textbf{while } B \textbf{ do } t)K) : \text{skip} \to \boldsymbol{l}((\textbf{if } B \textbf{ then } (t\,(\textbf{while } B \textbf{ do } t)))\,K),\\ &\;\boldsymbol{l}((\textbf{if } B \textbf{ then } (t\,(\textbf{while } B \textbf{ do } t)))\,K) : B \to \boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K),\\ &\;\boldsymbol{l}((\textbf{if } B \textbf{ then } (t\,(\textbf{while } B \textbf{ do } t)))\,K) : \neg B \to \boldsymbol{l}(K)\big\}\\ &\cup \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) \cup \mathcal{C}(K)\end{aligned}$$

We first show that $\mathcal{C}((\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} \mathit{extr}^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$. We consider the following label renaming:

$$\begin{aligned}\boldsymbol{l}((\textbf{while } B \textbf{ do } t)K) &\mapsto \ell_{-2}\\ \boldsymbol{l}((\textbf{if } B \textbf{ then } (t\,(\textbf{while } B \textbf{ do } t)))\,K) &\mapsto \ell_{-1}\\ \boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) &\mapsto \ell_0\end{aligned}$$

so that it remains to show that $\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} \mathit{extr}^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$. 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, $S_0 \equiv x := E; T(\textbf{while } B \textbf{ do } S)K$, $S_1 \equiv T(\textbf{while } B \textbf{ do } S)K$, $A_0 \equiv x := E$. In this case,

$$\begin{aligned}\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) = \{\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : x := E \to \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)\}&\\ \cup\, \mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K).&\end{aligned}$$

Hence, it is enough to consider the relabeling $\boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \mapsto \ell_1$ and to show that $\mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} \mathit{extr}^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$.

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

(iii) $c_1 \equiv$ **skip**; and $S_0 \equiv ($**while** B' **do** $S')T($**while** B **do** $S)K$. Thus, $S_1 \equiv ($**if** B' **then** $(S'($**while** B' **do** $S')))T($**while** B **do** $S)K$ and $A_0 \equiv$ skip. Here, we have that

$$\begin{aligned}\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) = \{&\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \text{skip} \rightarrow \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)\}\\ &\cup \mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K).\end{aligned}$$

Again, it is enough to consider the relabeling $\boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \mapsto \ell_1$ and to show that $\mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} extr^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$.

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

$$\begin{aligned}\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) = \{&\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \neg B' \rightarrow \boldsymbol{l}(T(\textbf{while } B \textbf{ do } S)K),\\ &\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : B' \rightarrow \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)\}\\ &\cup \mathcal{C}(T(\textbf{while } B \textbf{ do } S)K) \cup \mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K),\end{aligned}$$

$$\begin{aligned}\mathcal{C}(S(\textbf{while } B \textbf{ do } S)K) = \{&\boldsymbol{l}(S(\textbf{while } B \textbf{ do } S)K) : B' \rightarrow \boldsymbol{l}(S'T(\textbf{while } B \textbf{ do } S)K),\\ &\boldsymbol{l}(S(\textbf{while } B \textbf{ do } S)K) : \neg B' \rightarrow \boldsymbol{l}(T(\textbf{while } B \textbf{ do } S)K)\}\\ &\cup \mathcal{C}(S'T(\textbf{while } B \textbf{ do } S)K) \cup \mathcal{C}(T(\textbf{while } B \textbf{ do } S)K).\end{aligned}$$

Hence, since $\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) \mapsto \ell_0$ and $A_0 \equiv B'$, it is enough to consider the relabeling $\boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \mapsto \ell_1$ and to show that $\mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} extr^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$.

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

$$\begin{aligned}\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) = \{&\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : B' \rightarrow \boldsymbol{l}(S'T(\textbf{while } B \textbf{ do } S)K),\\ &\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \neg B' \rightarrow \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)\}\\ &\cup \mathcal{C}(S'T(\textbf{while } B \textbf{ do } S)K) \cup \mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K),\end{aligned}$$

while $\mathcal{C}(S(\textbf{while } B \textbf{ do } S)K)$ is the same as in the previous point (iv). Hence, since $\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) \mapsto \ell_0$ and $A_0 \equiv \neg B'$, it is enough to consider the relabeling $\boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \mapsto \ell_1$ and to show that $\mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} extr^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$.

Thus, in order to prove this containment, it remains to show that $\mathcal{C}(t'(\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} extr^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$. If $t' = \epsilon$ then the containment boils down to $\mathcal{C}((\textbf{while } B \textbf{ do } t)K) \subseteq_{/\cong} extr^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K))$ which is therefore proved. Otherwise, $t' = c_2t''$, 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, $extr^{GP}_{hp_t}(\mathcal{C}((\textbf{while } B \textbf{ do } S)K)) \subseteq_{/\cong} \mathcal{C}((\textbf{while } B \textbf{ do } t)K)$. For the trace $t = c_1c_2...c_{n+1}$, we know by $(*)$ that each command c_i either is in $\{$**skip**;, $x := E;\}$ or is one of the two following bail commands (cf. cases (3) and (4) in $(*)$):

$$\textbf{bail } \neg B' \textbf{ to } (T(\textbf{while } B \textbf{ do } S)K), \qquad \textbf{bail } B' \textbf{ to } (S'T(\textbf{while } B \textbf{ do } S)K).$$

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 $c_1...c_{k-1}$ consists of skip and assignment commands only, we have that $\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) \supseteq \mathcal{C}(c_k...c_{n+1}(\textbf{while } B \textbf{ do } t)K)$. Hence, either $\mathcal{C}(c_k...c_{n+1}(\textbf{while } B \textbf{ do } t)K) \supseteq \mathcal{C}(T(\textbf{while } B \textbf{ do } S)K)$ or $\mathcal{C}(c_k...c_{n+1}(\textbf{while } B \textbf{ do } t)K) \supseteq \mathcal{C}(T(\textbf{while } B \textbf{ do } S)K)$. In both cases, we obtain that $\mathcal{C}(c_k...c_{n+1}(\textbf{while } B \textbf{ do } t)K) \supseteq \mathcal{C}((\textbf{while } B \textbf{ do } S)K)$, so that $\mathcal{C}((\textbf{while } B \textbf{ do } S)K) \subseteq \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K) \subseteq \mathcal{C}((\textbf{while } B \textbf{ do } t)K)$. Thus, it remains to show that

$$\begin{aligned}&\{\ell_{-2} : \text{skip} \to \ell_{-1},\, \ell_{-1} : B \to \ell_0,\, \ell_{-1} : \neg B \to \boldsymbol{l}(K)\} \cup \\ &\{\ell_i : A_i \to \ell_{\mathit{next}(i)} \mid i \in [0,n]\} \cup \{\ell_i : \neg A_i \to \boldsymbol{l}(S_{\mathit{next}(i)})^c \mid i \in [0,n],\, A_i \in \text{BExp}\}\end{aligned}$$

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

$$\begin{aligned}\ell_{-2} &\mapsto \boldsymbol{l}((\textbf{while } B \textbf{ do } t)K)\\ \ell_{-1} &\mapsto \boldsymbol{l}((\textbf{if } B \textbf{ then } (t\,(\textbf{while } B \textbf{ do } t)))\,K)\\ \ell_0 &\mapsto \boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K)\end{aligned}$$

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

(i) $A_0 \equiv \text{skip}$ because $S_0 \equiv \textbf{skip}; T(\textbf{while } B \textbf{ do } S)K$. Here, $t = \textbf{skip}; t'$. Hence, $\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \text{skip} \to \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \in \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K)$ and it is enough to use the relabeling $\ell_1 \mapsto \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)$.

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

(iii) $A_0 \equiv \text{skip}$ because $S_0 \equiv (\textbf{while } B' \textbf{ do } S')T(\textbf{while } B \textbf{ do } S)K$. Here, $t = \textbf{skip}; t'$. Here, again, $\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \text{skip} \to \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \in \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K)$, so that it is enough to use the relabeling $\ell_1 \mapsto \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)$.

(iv) $A_0 \equiv B'$ because $S_0 \equiv (\textbf{if } B' \textbf{ then } S')T(\textbf{while } B \textbf{ do } S)K$ and $\mathbf{B}[\![B']\!]\rho_0 = \mathit{true}$. Thus, $t = (\textbf{bail } \neg B' \textbf{ to } (T(\textbf{while } B \textbf{ do } S)K))t'$ and $S_1 \equiv T(\textbf{while } B \textbf{ do } S)K$. Note that $\boldsymbol{l}(S_1)^c = \boldsymbol{l}(T(\textbf{while } B \textbf{ do } S)K)$. Hence,

$$\begin{aligned}&\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \neg B' \to \boldsymbol{l}(T(\textbf{while } B \textbf{ do } S)K) \in \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K),\\ &\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : B' \to \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \in \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K).\end{aligned}$$

Once again, the relabeling $\ell_1 \mapsto \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)$ allows us to obtain that $\ell_0 : B' \to \ell_1$ and $\ell_0 : \neg B' \to \boldsymbol{l}(T(\textbf{while } B \textbf{ do } S)K)$ are in $\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K)$.

(v) $A_0 \equiv \neg B'$ because $S_0 \equiv (\textbf{if } B' \textbf{ then } S')T(\textbf{while } B \textbf{ do } S)K$ and $\mathbf{B}[\![B']\!]\rho_0 = \mathit{false}$. Here, $t = (\textbf{bail } B' \textbf{ to } (S'T(\textbf{while } B \textbf{ do } S)K))t'$ and $S_1 \equiv T(\textbf{while } B \textbf{ do } S)K$. Note that $\boldsymbol{l}(S_1)^c = \boldsymbol{l}(S'T(\textbf{while } B \textbf{ do } S)K)$. Hence,

$$\begin{aligned}&\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : B' \to \boldsymbol{l}(S'T(\textbf{while } B \textbf{ do } S)K) \in \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K),\\ &\boldsymbol{l}(t(\textbf{while } B \textbf{ do } t)K) : \neg B' \to \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K) \in \mathcal{C}(t(\textbf{while } B \textbf{ do } t)K).\end{aligned}$$

Thus, through the relabeling $\ell_1 \mapsto \boldsymbol{l}(t'(\textbf{while } B \textbf{ do } t)K)$ we obtain that $\ell_0 : \neg B' \to \ell_1$ and $\ell_0 : B' \to \boldsymbol{l}(S'T(\textbf{while } B \textbf{ do } S)K)$ are in $\mathcal{C}(t(\textbf{while } B \textbf{ do } t)K)$.

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* $P \in \text{Program}$, $hp = C_0 \cdots C_n \in \alpha^{GP}_{hot}(\mathbf{T}[\![P]\!])$*, we have that* $\alpha_{sc}(\mathbf{T}[\![\mathit{extr}^{GP}_{hp}(P)]\!]) = \alpha_{sc}(\mathbf{T}[\![P]\!])$.

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 on the correctness of trace extraction. Let us observe that since α_{sc} is a stronger abstraction than α^{ρ}_{sc} and, by Theorem 10.6, we know that α^{ρ}_{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: $\alpha^{\rho}_{sc}(\mathbf{T}[\![extr^{GP}_{hp}(P)]\!]) = \alpha^{\rho}_{sc}(\mathbf{T}[\![P]\!])$.

11. CONCLUSION AND FURTHER WORK

This article put forward a formal model of tracing JIT compilation which allows: (1) an easy definition of program hot paths—that is, most frequently executed program traces; (2) to prove the correctness of a hot path extraction transform of programs; (3) to prove the correctness of dynamic optimizations confined to hot paths, such as dynamic type specialization along a hot path. Our approach is based on two main ideas: the use of a standard trace semantics for modeling the behavior of programs and the use of abstract interpretation for defining the notion of hot path as an abstraction of the trace semantics and for proving the correctness of hot path extraction and optimization. We have shown that this framework is more flexible than Guo and Palsberg [2011] model of tracing JIT compilation, which relies on a notion of correctness based on operational program bisimulations, and allows to overcome some limitations of [Guo and Palsberg 2011] on selection and annotation of hot paths and on the correctness of optimizations such as dead store elimination. We expect that most optimizations employed by tracing JIT compilers can be formalized and proved correct using the proof methodology of our framework.

We see a number of interesting avenues for further work on this topic. As a significant example of optimization implemented by a practical tracing compiler, it would be worth 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 think that our framework could be adapted 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, the main ideas of our model could be useful to study and relate the foundational differences between traditional static vs dynamic tracing compilation.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their helpful comments.

REFERENCES

K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi. 2014. The Hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages (OOPSLA 2014)*. ACM, New York, NY, USA, 777–790. DOI:http://dx.doi.org/10.1145/2660193.2660199

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. DOI:http://dx.doi.org/10.1145/349299.349303

R. Barbuti, 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. DOI:http://dx.doi.org/10.1016/S0020-0190(99)00042-3

S. Bauman, R. Bolz, C.F. Hirschfeld, V. Krilichev, T. Pape, J.G. Siek, and S. Tobin-Hochstadt. 2015. Pycket: A tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 22–34. DOI:http://dx.doi.org/10.1145/2784731.2784740

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. DOI:http://dx.doi.org/10.1145/1869459.1869517

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. DOI:http://dx.doi.org/10.1145/1993498.1993508

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. DOI:http://dx.doi.org/10.1145/1929501.1929508

C.F. Bolz, A. Cuni, M. Fijalkowski, 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. DOI:http://dx.doi.org/10.1145/1565824.1565827

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. DOI:http://dx.doi.org/10.1145/237721.237776

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. DOI:http://dx.doi.org/10.1016/S1571-0661(05)80168-9 Proceedings of the 13th Annual Conference on Mathematical Foundations of Progamming 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. DOI:http://dx.doi.org/10.1145/512950.512973

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. DOI:http://dx.doi.org/10.1145/567752.567778

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. DOI:http://dx.doi.org/10.1145/503272.503290

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. DOI:http://dx.doi.org/10.1145/2535838.2535866

Ecma International. 2015. Standard ECMA-262, ECMAScript 2015 Language Specification, 6th Edition. http://www.ecma-international.org/ecma-262/6.0. (2015).

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

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. DOI:http://dx.doi.org/10.1145/1542476.1542528

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. DOI:http://dx.doi.org/10.1145/1926385.1926450

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. DOI:http://dx.doi.org/10.1007/3-540-49727-7_12

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. DOI:http://dx.doi.org/10.1145/2093157.2093176

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. DOI:http://dx.doi.org/10.1145/2579673

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. DOI:http://dx.doi.org/10.1109/CGO.2011.5764692

F. Logozzo. 2009. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems and Structures* 35, 2 (2009), 100–142. DOI:http://dx.doi.org/10.1016/j.cl.2005.01.001

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. (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. DOI:http://dx.doi.org/10.1145/964001.964002

X. Rival and L. Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 26 (2007), 51 pages. DOI:http://dx.doi.org/10.1145/1275497.1275501

T. Schilling. 2013. *Trace-based Just-In-Time Compilation for Lazy Functional Programming Languages*. Ph.D. Dissertation. University of Kent, UK.

D.A. Schmidt. 1998. Trace-based abstract interpretation of operational semantics. *Lisp Symb. Comput.* 10, 3 (1998), 237–271. DOI:http://dx.doi.org/10.1023/A:1007734417713

F. Spoto and T. Jensen. 2003. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.* 25, 5 (2003), 578–630. DOI:http://dx.doi.org/10.1145/937563.937565

M.N. Wegman and F.K. Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. DOI:http://dx.doi.org/10.1145/103135.103136