A Flexible Framework for Studying Trace-Based Just-In-Time Compilation

Maarten Vandercammen[a], Stefan Marr[b], Coen De Roover[a]

[a]*Software Languages Lab, Vrije Universiteit Brussel, Belgium*
[b]*Institute for System Software, Johannes Kepler University, Austria*

Abstract

Just-in-time compilation has proven an effective, though effort-intensive, choice for realizing performant language runtimes. Recently introduced JIT compilation frameworks advocate applying meta-compilation techniques such as partial evaluation or meta-tracing on simple interpreters to reduce the implementation effort.

However, such frameworks are few and far between. Designed and highly optimized for performance, they are difficult to experiment with. We therefore present STRAF, a minimalistic yet flexible Scala framework for studying trace-based JIT compilation. STRAF is sufficiently general to support a diverse set of language interpreters, but also sufficiently extensible to enable experiments with trace recording and optimization. We demonstrate the former by plugging two different interpreters into STRAF. We demonstrate the latter by extending STRAF with e.g., constant folding and type-specialization optimizations, which are commonly found in dedicated trace-based JIT compilers. The evaluation shows that STRAF is suitable for prototyping new techniques and formalisms in the domain of trace-based JIT compilation.

Keywords: trace-based JIT compilation, optimization, operational semantics

1. Introduction

Constructing a dedicated just-in-time compiler for a language requires significant engineering effort. The Truffle [23] and RPython [4] frameworks

Email addresses: Maarten.Vandercammen@vub.ac.be (Maarten Vandercammen), stefan.marr@jku.at (Stefan Marr), Coen.De.Roover@vub.ac.be (Coen De Roover)

address this problem by reducing the language-specific engineering that is required by applying partial evaluation and meta-tracing to relatively simple interpreters. It has recently been shown [16] that the technique of meta-tracing is capable of lifting the performance of a meta-traced interpreter to the same order of magnitude of a dedicated just-in-time compiler, while requiring less engineering effort from the developers of this interpreter.

However, several open research questions for (meta-)trace-based compilation remain. For example, how can the warm-up time of the compiler be reduced, and how can the problem of trace explosion be addressed to avoid tracing an exponential number of paths. Although RPython has proven itself as a framework for constructing performant language runtimes, its performance focus makes it difficult to adapt the framework itself or experiment with various compilation strategies. Addressing the aforementioned research questions by experimenting in RPython is therefore a complex undertaking. We therefore introduce STRAF, a minimalistic Scala framework with the aim of facilitating further experiments in trace-based JIT compilation. STRAF is designed not as a performant competitor to RPython, but as an extensible research vehicle for studying tracing compilation. It enables experiments with dynamic analyses of traces, with strategies for their optimization, and with the various ways in which executions, traces, and optimizations interact. STRAF can therefore be used as a testbed for various experimental strategies in trace-based compilation. Once researchers feel that these strategies have been sufficiently explored, they may be implemented in a mature trace-based compiler to be further developed and evaluated.

As the main priority of STRAF is to achieve an extensible and minimalistic tracing framework, we separate the tracing mechanism in STRAF from the actual semantics of the language being executed. This results in a flexible runtime that can be composed with various language interpreters, similar to meta-tracing frameworks like RPython. However, in contrast to these meta-tracers, the traces recorded by STRAF are not generic, but are specific to the interpreter that is employed. As traces are interpreter-specific, language implementers wishing to benefit from the advantages of the STRAF framework must therefore provide a number of hooks in their interpreter, e.g., to enable optimization and de-optimization, that are not required by a meta-tracing framework. The effort required for language implementers to compose their interpreter in STRAF are therefore higher than in traditional meta-tracing framework, but enables maximal decoupling of tracing and language semantics. The difference between the STRAF framework and

a general meta-tracing compiler is described in more detail in Section 3.4.

To concisely describe the framework, we formalize STRAF and provide the implementation.[1] Our implementation integrates with a Scala framework [19] for defining abstract machines through the AAM methodology [20]. This methodology provides a procedure for systematically transforming the *concrete* semantics of any language, which, when implemented, correspond to a concrete interpreter for the language, to some *abstract* semantics of this language. The abstract semantics enable finite reasoning over a program's execution and can therefore be used as the basis for a static analysis of the language. The integration of STRAF with this analysis framework only adds to STRAF's potential for experimentation.

STRAF not only offers a large degree of flexibility in terms of the language interpreters it can execute, it is also adaptable in how the framework itself can be extended. In this article, we evaluate both aspects. First, we present two different interpreters and demonstrate how they can be plugged into STRAF. Second, we show how to extend STRAF with six trace optimizations, with a heuristic for selecting hot loops for which it is effective to start tracing, and with guard tracing to mitigate the performance penalty of aborting the execution of a previously-recorded trace. These extensions are commonly found in trace-based JIT compilers. Concretely, this article makes the following contributions:

- the design, a formal specification, and a reference implementation of the minimalistic, but extensible STRAF framework into which interpreters can be plugged to construct a trace-based JIT compiler,
- an evaluation of STRAF's generality by composing it with two language interpreters,
- an evaluation of STRAF's extensibility by extending it with six trace optimizations, with a heuristic for detecting hot loops, and with the ability to start tracing from the point of a guard failure.

2. Trace-Based JIT Compilation

Trace-based JIT compilation is an alternative to the more common method-based JIT compilation. It builds on two basic assumptions: most of the exe-

[1] Available at `https://github.com/mvdcamme/scala-am`

cution time of a program is spent in loops, and several iterations of the same loop are likely to take the same path through the program [4]. Trace-based JIT compilers therefore do not limit compilation to methods, like method-based ones, but they trace and compile frequently executed, i.e., "hot" loops.

Runtimes incorporating a trace-based JIT compiler usually do so through mixed-mode execution. Initially, an interpreter executes the program and profiles loops to identify hot ones. When a hot loop is detected, the runtime starts *tracing* the execution of this loop: the operations that are performed by the interpreter while in this loop are recorded into a trace. Tracing continues until the interpreter has completed a full iteration of the loop. The recorded trace is then compiled and optimized. Subsequent iterations of this loop will execute the compiled trace.

Because a trace represents a single execution path, it must ensure that the conditions that held while the trace was being *recorded* still hold when it is *executed*. These assumptions are checked by inserting *guards* encoding the corresponding conditions in each trace. When a guard *fails*, execution of the trace is aborted and the interpreter resumes normal interpretation of the program from that point onward. The point where trace execution is aborted and interpretation restarts is called a *side-exit*. Side-exits give rise to a performance penalty, because execution of the optimized trace must be aborted and evaluation must proceed through regular interpretation of the program. To mitigate the overhead, most tracing compilers use optimized *trace bridges* to jump from one trace to another, once a guard has failed [18].

Example. Listing 1a depicts a Scheme program containing a loop. Part of the loop's corresponding trace is depicted in Listing 1b. As the expression `(= n 0)` evaluated to `false` during tracing, the tracer inserted a guard `ActionGuardFalse` that will check whether this condition still evaluates to `false` during trace execution.

3. Overview of STRAF

A language runtime implemented using STRAF consists of two main entities: an interpreter, responsible for regular program execution, and a tracing machine or *tracer*, responsible for trace recording and execution. The tracer is provided by the STRAF framework, while the interpreter is to be provided by the language developer, in a manner similar, but not identical, to meta-tracing. The tracer controls the interpreter by repeatedly asking it

```
(define (fac n)
  (if (= n 0)
      1
      (* n
         (fac (- n 1)))))

(fac 5)
```

(a) The program to be traced

```
...
ActionEvalPush("=", FrameFunCallFunction(List("n", 0)))
ActionLookupVar("=")
ActionPushValue
ActionPopKont
ActionEvalPush("n", FrameFunCallArgs(List(0)))
ActionLookupVar("n")
ActionPushValue
ActionPopKont
ActionEvalPush(0, FrameFunCallArgs(List()))
ActionLiteralValue(0)
...
ActionGuardFalse(...)
ActionEvalTraced((* n (fac (- n 1))))
...
```

(b) Part of the trace

Listing 1: A loop in a program and part of its corresponding trace.

$$
\begin{aligned}
\mathit{TracerState} ::= \ & \mathbf{ts}(\mathit{ExecutionPhase}, \\
& \quad \mathit{TracerContext}, \\
& \quad \mathit{ProgramState}, \\
& \quad \mathit{Null} + \mathit{TraceNode}) \\
\mathit{ExecutionPhase} ::= \ & \mathrm{NI} \mid \mathrm{TR} \mid \mathrm{TE} \\
\mathit{tc} \in \mathit{TracerContext} ::= \ & \mathbf{tc}(\mathit{Null} + \mathit{TraceNode},\ \mathit{TraceNodeMap}) \\
\mathit{tn} \in \mathit{TraceNode} ::= \ & \mathbf{tn}(\mathit{Label},\ \mathit{Trace},\ \mathit{ProgramState}) \\
T \in \mathit{TraceNodeMap} ::= \ & \mathit{Label} \to \mathit{TraceNode} \\
a \in \mathit{Action} ::= \ & \mathit{InterpreterAction} \mid et_{rp} \\
t \in \mathit{Trace} ::= \ & \mathit{Action}* \\
\mathit{ActionReturn} ::= \ & \mathbf{actionStep}(\mathit{ProgramState}) \\
& \mid \mathbf{guardFailed}(\mathit{RestartPoint}) \\
& \mid \mathbf{endTrace}(\mathit{RestartPoint}) \\
\mathit{TracingSignal} ::= \ & \mathbf{startLoop}(\mathit{Label}) \\
& \mid \mathbf{endLoop}(\mathit{Label}, \mathit{RestartPoint}) \\
& \mid \mathit{False} \\
\mathit{InterpreterStep} ::= \ & \mathbf{interpreterStep}(\mathit{Trace}, \mathit{TracingSignal})
\end{aligned}
$$

Figure 1: The tracing machine.

to execute a *step* when no trace is being executed, and is itself responsible for determining how execution of the program should proceed in other cases. Section 3.2 details the interface through which the tracer and the interpreter communicate. For instance, the interpreter is to send *signals* to the tracer when it reaches interesting points in the program, such as the beginning of or exit from a loop iteration.

Execution is divided into three distinct execution phases: *normal interpretation*, in which the program is interpreted without the tracer interfering, *trace recording*, in which the operations of the interpreter are recorded by the tracing machine, and *trace execution*, in which a previously recorded trace is executed. The execution phases and their transitions can be modeled as a state diagram, shown in Figure 2.

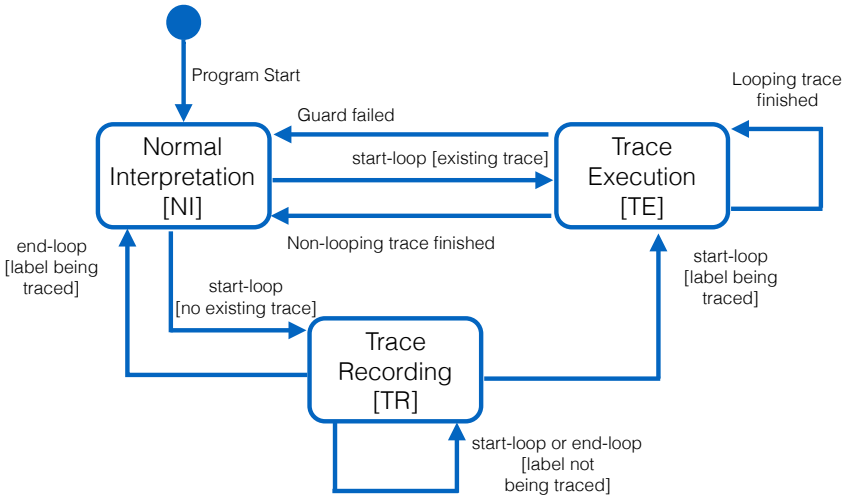


Figure 2: The three execution phases of a program.

3.1. Tracer State

The tracer is modeled as a state machine transitioning between *tracer states*. Figure 1 lists the definitions of these states: a *TracerState* consists of a reference to the aforementioned execution phase, a tracer context, a program state, and a trace node.

During the execution of the program, the tracer switches between the *ExecutionPhases*: normal interpretation (NI), trace recording (TR) and trace execution (TE) phases. Section 3.3 describes the transitions between the different states of the tracer.

The *TracerContext* is a two-tuple used by the tracer. The first component of the tuple stores the trace that is currently being recorded. This is either *Null*, if no trace is being recorded, or it is a trace node (*TraceNode*), which is a three-tuple that associates a trace with a unique label and a program state, so that this trace can later be retrieved by referencing its label. The second component, *TraceNodeMap*, stores all trace nodes, containing the traces that were previously recorded, by mapping the aforementioned labels to the trace nodes. The trace itself is a sequence of *actions* which are opaque interpreter-specific data structures that represent the operations performed by the interpreter while evaluating the program. When executing the trace, these same actions are again executed one-by-one by the interpreter. As these actions are unique to the interpreter that is used, they are not defined here. An example of some possible actions appeared in Listing 1b and is

shown again in Section 4, when discussing one possible implementation of the interpreter. However, we define one special *end-trace* action et_{rp}, whose semantics are detailed in Section 3.3.

It is assumed that interpreters are modeled as state machines operating on a program state. This requirement enables the tracer to grab the entire, current execution state of the interpreter in the form of some program state which is defined by the interpreter and remains opaque to the tracer. The implementer of the interpreter could for example model the interpreter as a CESK machine. CESK-based interpreters operate on CESK states, consisting of a control component (C), an environment (E) (mapping variables to addresses), a store (S) (mapping addresses to values) and a continuation stack (K) [12]. These interpreters are guided through the evaluation of a program by checking the state's control component, which corresponds with either an expression to next be evaluated or a continuation to be followed. The state's environment maps variables to addresses and its store maps these addresses to values. The continuation stack saves the continuations to be followed upon completing the evaluation of a (sub)expression and reaching a value. Abstracting a program's execution as a program state facilitates transitioning between the various phases of execution as both the execution of a trace and normal interpretation of the program operate on the same structure.

During the evaluation of the program, the interpreter operates on these program states and determines the next instruction, which, depending on the current execution phase, may be recorded into a trace by the tracing machine. The tracer obtains new program states from the interpreter during normal interpretation and trace recording, or by executing trace instructions during trace execution.

The last component of the tracer state either equals *Null* if no trace is currently being executed, or it contains the trace node storing the trace that is being executed.

3.2. Tracing Interface

Interpreter Functions. The tracing machine monitors and controls the execution of the interpreter through the following interface, which must be

provided by the interpreter:

$$\begin{aligned}
\texttt{step} &: \mathit{ProgramState} \to \mathit{InterpreterStep} \\
\texttt{applyAction} &: \mathit{ProgramState} \times \mathit{Action} \to \mathit{ActionReturn} \\
\texttt{restart} &: \mathit{RestartPoint} \times \mathit{ProgramState} \to \mathit{ProgramState} \\
\texttt{optimize} &: \mathit{Trace} \times \mathit{ProgramState} \to \mathit{Trace}
\end{aligned}$$

The tracer asks the interpreter to perform a single evaluation step on a program state via a two-step process. The tracer first calls the `step` function on this state. In this `step` call, the interpreter checks the state and considers which operations must be completed in this step of the evaluation. It then reifies these operations in the form of actions, i.e., data structures representing the operations to execute, and wraps a list of the computed actions in an *InterpreterStep*. As the second step in the process, the tracer then makes the interpreter actually execute these reified operations by calling `applyAction` on them and the state to compute a *ActionReturn* that contains the new, updated program state. The `restart` function enables the tracer to restart normal interpretation when a guard failure has occurred at run time. The `optimize` function takes a trace and a program state and returns a trace that is optimized with respect to the given program state. It is designed such that the tracer can consider the optimization of a trace as a black box, rendering it the responsibility of the language implementer. In Section 4, we demonstrate how an interpreter that satisfies this interface may be built.

Note that although in principle the definitions of program states and actions are specific to one particular interpreter, in practice they might be reused between different interpreters, which in turn would enable language developers to also reuse at least the `applyAction` and `optimize` functions, similar to what is done in PyPy. However, creating a set of these common elements might place further constraints on the design of the interpreter, as the interpreter would have to accommodate for these components by adapting its `step` and `restart` functions so that they employ these actions and states. Additionally, such a common program state should be sufficiently generic that it could be used by any sort of interpreter.

Program States. With the interpreter being a state machine, interpreting a program amounts to continuously executing the state transition rule that is applicable for the current program state; tracing the interpreter becomes recording the transitions performed by the interpreter state machine and

executing a trace corresponds to replaying the recorded transitions starting from the current program state. These transitions thus correspond to the aforementioned actions in STRAF. Note that our tracer does not depend on a particular definition for the program state or state transition, but this is left to the interpreter.

Actions. To enable a more fine-grained optimization of traces, the interpreter can use two sets of state transition rules: high-level and low-level transitions, i.e., actions, both operating on a program state. One high-level transition is composed of several low-level actions. As illustrated by Figure 3, executing the high-level transition is equivalent to applying each of the constituent actions consecutively. The high-level transition itself does not appear in a trace; only the low-level actions are recorded.

InterpreterStep. During the normal interpretation and trace recording phases, the tracing machine repeatedly asks the interpreter to perform a single high-level transition by calling `step`. This function takes the current program state as input and outputs an *InterpreterStep*: a two-tuple containing a list of actions to be applied on the given program state that together constitute the high-level transition that has just been performed, and possibly a *tracing signal*. When the interpreter enters a loop, it communicates this to the tracer by including a tracing signal, **startLoop**, in its response. This enables the tracer to decide whether to start tracing this loop, start executing a previously recorded trace for this loop, or do nothing at all. For this to work, the interpreter should identify each loop in the user program uniquely through a label. When interpreting a loop over multiple iterations, a **startLoop** is sent at the start of *each* iteration. If the tracer has started recording a loop after detecting a **startLoop** signal at one iteration and subsequently detects another **startLoop** for the same loop, it knows that one full iteration of the loop has been completed so it can stop recording. Conversely, when the interpreter exits a loop instead of continuing with another iteration, it includes the **endLoop** signal in the *InterpreterStep*. This enables the tracer to stop tracing in case it had been tracing this loop, so as not to trace *outside of the loop*. We will call traces whose recording is stopped via such an **endLoop** signal *non-looping traces* in contrast to *looping traces* which are terminated via a **startLoop**. The difference between looping and non-looping traces is made more clear in Listing 2. Note that in Scheme, loops are constructed by recursively calling a function. Every function call could therefore loop back

```
(define (loop n)
   (loop (+ n 1)))
```

(a) A looping function; tracing will be terminated via a **startLoop**.

```
(define (id x)
   x)
```

(b) A non-looping function; tracing will be terminated via an **endLoop**.

Listing 2: A looping versus a non-looping function.

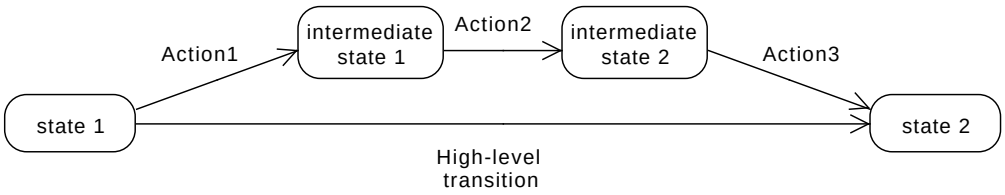


Figure 3: A high-level transition and corresponding actions.

to itself, so the interpreter sends an **startLoop** at the start of *every* function call. Should the function indeed recursively call itself, as depicted in Listing 3a, a second **startLoop** will be sent and recording will be terminated. If no recursive call takes place, as is the case for Listing 3b, the interpreter sends a **endLoop** upon returning from the function call. Upon later execution of a non-looping trace, the tracer will restart normal interpretation when it reaches the end of the trace.

Note that in the basic model of STRAF, a nested loop will be inlined when tracing the outer loop. However, it would be possible to extend STRAF such that this inlining is avoided, e.g., by aborting trace recording of an outer loop when an inner loop is detected.

Applying an Action. An action is applied by the tracer via the interface's `applyAction` function. This returns an *ActionReturn* structure, which can be: an **actionStep**, a **guardFailed** or an **endTrace**. Most actions result in an **actionStep**, which wraps the new program state that is the result of applying the action on the input state. The purpose of **guardFailed** and **endTrace** will become clear over the next sections. Note that, in this model, guards are also a kind of action.

Guard Instructions. As a trace represents a single execution, guards are inserted to ensure that a trace is only executed when the conditions that lead

to this specific path through the program are valid. Guards being actions themselves, when applied via `applyAction`, they cause the interpreter to check some condition on this state and then either return some **actionStep** in case the guard did not fail, or a **guardFailed** in case it did. The tracer detects this return value and takes action accordingly. As the generation and placement of guards is specific to the interpreted language, they need to be created by the interpreter during its processing of `step` requests and be included in the list of actions returned to the tracer via an **interpreterStep**.

Restarting Interpretation. Upon failure of a guard during the execution of a trace, or upon reaching the end of a non-looping trace, the tracer applies `restart` to the current program state and to a so-called *restart point* in order to restart normal interpretation. A restart point includes the necessary information to construct the program state from where normal interpretation must resume. Similar to actions and program states, the exact definition of a restart point intentionally depends on the interpreters. For the CESK example, a restart point could correspond to the control field of a state and point to the program expression that must now be evaluated; `restart` could then take this control and merge it with the other fields of the CESK state. Note that the design of correct restart points may depend on not only the interpreter that is used, but also on the optimizations employed by this interpreter, as the optimizations that are applied on a trace may have an effect on the restart points inside this trace. Similar to the process of optimizing traces, designing correct restart points, as well as ensuring that any interference between the trace optimizations and the restart points is resolved, is hence a responsibility of the interpreter developers.

3.3. Transition Rules

Figure 4 lists the formal semantics of the tracing machine and its interaction with the interpreter. We use this formalism to concisely describe the working of STRAF. A reference implementation for these semantics is available at `https://github.com/mvdcamme/scala-am`.

These semantics center around how the *TracerState* of the tracing machine is updated as execution of the program proceeds. For each rule, the first line matches the current configuration of the *TracerState*, the second line describes the updated *TracerState*. Subsequent lines describe conditions that must hold for for the original *TracerState* to transition such that it produces this updated *TracerState*; specifically, the third line always indicates

what the result of the interpreter's `step` function on the current program state must have been for this transition to take place. Note also that we use a helper function `applyAction*` which takes a program state and a sequence of actions as input and consecutively updates the program state with each action in the sequence, assuming the action resulted in an **actionStep**.

The helper function `applyAction*` can be recursively defined as follows:

$$\frac{t \text{ is an empty list}}{\texttt{applyAction*}(s, t) = s} \textsc{ applyActionEmpty}$$

$$\frac{\texttt{applyAction}(s, a) = \textbf{actionStep}(s')}{\texttt{applyAction*}(s, a : t) = \texttt{applyAction*}(s', t)} \textsc{ applyActionNonEmpty}$$

We also use the underscore character to match any field whose value is irrelevant. We use the notation $T[lbl]$ to look up the label lbl in the map T. If the map does not contain this label, it returns an undefined value. Similarly, we use the notation $T[lbl \mapsto tn]$ to either extend the map T with tn at lbl, if T did not yet contain lbl, or to replace the previous entry for lbl with the value tn.

Normal Interpretation. The normal interpretation phase (NI) refers to the execution stage in which no trace is being recorded or executed: the tracer only intervenes when the interpreter reaches the start of a loop, signaled by the interpreter via a *TracingSignal*, at which point the tracer may either decide to start tracing or to start executing a previously recorded trace. Figure 4a depicts the corresponding formalization.

Rule NI-ContinueInterpreting represents the most common case in which the interpreter either has not entered any loop, and the interpreter hence returns *False* instead of an actual signal, or the interpreter has exited a loop and it sends the **endLoop** tracing signal to indicate this. In both cases, the interpreter also returns the list of actions, t, that must be applied to arrive at the new program state, s'. As no actions are recorded while in the NI phase, the new tracer state is simply a copy of the old one, with the original program state replaced by the new one.

In rules NI-StartTracing and NI-StartExecuting, the interpreter enters a loop that is identified by the label lbl. The first sequence of actions, that are already part of the loop, consist of $a_1 : ... : a_n$. In rule NI-StartTracing, no trace has been recorded yet for this loop, so the tracer

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(t,\ signal) \\ signal \text{ equals either } False \text{ or } \textbf{endLoop}(lbl, rp)\end{array}}{\begin{array}{c}\textbf{ts}(\text{NI},\ tc,\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{NI},\ tc,\ s',\ Null)\end{array}}\ \text{NI-ContinueInterpreting}$$

Where s' = `applyAction*`(s, t)

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(a_1 : ... : a_n,\ \textbf{startLoop}(lbl)) \\ T[lbl] \text{ is undefined}\end{array}}{\begin{array}{c}\textbf{ts}(\text{NI},\ \textbf{tc}(\_,\ T),\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(lbl,\ a_1 : ... : a_n,\ s),\ T),\ s',\ Null)\end{array}}\ \text{NI-StartTracing}$$

Where $s' = \texttt{applyAction*}(s,\ a_1 : ... : a_n)$

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(\_,\ \textbf{startLoop}(lbl)) \\ T[lbl] = tn\end{array}}{\begin{array}{c}\textbf{ts}(\text{NI},\ tc,\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{TE},\ tc,\ s,\ tn)\end{array}}\ \text{NI-StartExecuting}$$

(a) Normal interpretation

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(a_1 : ... : a_n,\ signal) \\ signal \text{ equals either } False \text{ or } \textbf{startLoop}(lbl') \text{ or } \textbf{endLoop}(lbl', \_) \text{ with } lbl \neq lbl'\end{array}}{\begin{array}{c}\textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(lbl,\ t,\ s_s),\ T),\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(lbl,\ t : a_1 : ... : a_n,\ s_s),\ T),\ s',\ Null)\end{array}}\ \text{TR-ContinueTracing}$$

Where $s' = \texttt{applyAction*}(s,\ a_1 : ... : a_n)$

$$\frac{\texttt{step}(s) = \textbf{interpreterStep}(t',\ \textbf{startLoop}(lbl))}{\begin{array}{c}\textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(lbl,\ t,\ s_s),\ T),\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{TE},\ \textbf{tc}(Null,\ T[lbl \mapsto tn]),\ s',\ tn)\end{array}}\ \text{TR-SameStart}$$

Where $s' = \texttt{applyAction*}(s,\ t')$
$tn = \textbf{tn}(lbl,\ \texttt{optimize}(t, s_s),\ s_s)$

$$\frac{}{\texttt{applyAction}(\_,\ et_{rp}) = \textbf{endTrace}(rp)}\ \text{ApplyActionEndTrace}$$

$$\frac{\texttt{step}(s) = \textbf{interpreterStep}(t',\ \textbf{endLoop}(lbl, rp))}{\begin{array}{c}\textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(lbl,\ t,\ s_s),\ T),\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{NI},\ \textbf{tc}(Null,\ T[lbl \mapsto tn]),\ s',\ Null)\end{array}}\ \text{TR-SameEnd}$$

Where $s' = \texttt{applyAction*}(s,\ t')$
$tn = \textbf{tn}(lbl,\ \texttt{optimize}(t : et_{rp}, s_s),\ s_s)$

(b) Trace recording

$$\frac{\texttt{applyAction}(s,\ a) = \textbf{actionStep}(s')}{\begin{array}{c}\textbf{ts}(\text{TE},\ tc,\ s,\ \textbf{tn}(lbl,\ a:t,\ \_)) \rightarrow \\ \textbf{ts}(\text{TE},\ tc,\ s',\ \textbf{tn}(lbl,\ t,\ \_))\end{array}}\ \text{TE-NoSignal}$$

$$\frac{\texttt{applyAction}(s,\ a) = \textbf{guardFailed}(rp)}{\begin{array}{c}\textbf{ts}(\text{TE},\ tc,\ s,\ \textbf{tn}(lbl,\ a:t,\ \_)) \rightarrow \\ \textbf{ts}(\text{NI},\ tc,\ s',\ \mathit{Null})\end{array}}\ \text{TE-GuardFailure}$$

Where $s' = \texttt{restart}(rp,\ s)$

$$\frac{\texttt{applyAction}(s,\ a) = \textbf{endTrace}(rp)}{\begin{array}{c}\textbf{ts}(\text{TE},\ tc,\ s,\ \textbf{tn}(lbl,\ a:t,\ \_)) \rightarrow \\ \textbf{ts}(\text{NI},\ tc,\ s',\ \mathit{Null})\end{array}}\ \text{TE-TraceEnd}$$

Where $s' = \texttt{restart}(rp,\ s)$

$$\frac{T[lbl] = tn}{\begin{array}{c}\textbf{ts}(\text{TE},\ \textbf{tc}(tn,\ T),\ s,\ \textbf{tn}(lbl,\ \phi,\ \_)) \rightarrow \\ \textbf{ts}(\text{TE},\ \textbf{tc}(tn,\ T),\ s,\ tn)\end{array}}\ \text{TE-RestartLoop}$$

Where ϕ represents the empty list

(c) Trace execution

Figure 4: Transition rules between tracer-states.

starts tracing it: it changes its execution phase to indicate that it is now tracing, updates its tracer context by replacing the component representing its current trace and, as the sequence of actions $a_1 : ... : a_n$ is part of the loop to be traced, it immediately records this sequence. The actions $a_1 : ... : a_n$ are also applied to arrive at the new program state s'. This component now becomes a trace node consisting of the label *lbl* of the loop that is traced, the actions that were executed by the interpreter and that were carried back in the **interpreterStep**, as well as the old program state s. s is saved so it can later be used as input for the `optimize` function, as described in Section 5.

In rule NI-StartExecuting, the interpreter also starts a new loop iteration, but the tracer context already contains a trace for this loop: i.e., it has an entry for the loop's label *lbl*. The tracer switches its execution phase to TE to indicate it must execute this trace in the following step and the tracer replaces the trace node of the tracer state for the trace node containing the trace for the label *lbl*. As execution must now switch to the trace, the actions carried back in the **interpreterStep** are discarded entirely.

Trace Recording. In the trace recording phase (TR), all actions executed by the interpreter are recorded into a trace. Recording stops when the interpreter sends either a **startLoop** signal or an **endLoop** signal carrying the same label as the trace being recorded. Figure 4b lists the corresponding rules.

Rule TR-ContinueTracing describes the situation where the interpreter has either not entered or exited a loop, or it has entered or exited a loop different from the one currently being traced, which is indicated by respectively the **startLoop** or **endLoop** carrying a label different from the label of the loop being traced. In any case, the tracer records the interpreter's actions by appending the list of actions $a_1 : ... : a_n$ returned from the interpreter to the back of the trace. The program state is also replaced as the tracing process remains otherwise unaffected, the tracer continues tracing.

In rule TR-SameStart, the interpreter reaches the start of a loop, but this loop has the same label as the one currently being traced. This means, one full iteration of the loop is completed and tracing can stop. The trace is then optimized, making use of the program state that was saved when starting the recording of this trace, and stored in the tracer context. The actions t' carried back in the **interpreterStep** are the same actions as those that were recorded at the beginning of the trace and are hence not recorded in the trace. Execution then continues by executing this optimized trace. Rule

TR-SameEnd describes the interpreter exiting a loop that is being traced, instead of continuing with its next iteration. The interpreter sends an **endLoop** signal carrying the loop's label and a restart point rp. In response, the tracer appends the special end-trace action et_{rp} to the end of the trace. The semantics of this action are defined in the ApplyActionEndTrace rule: when this action is executed via the `applyAction` function, `applyAction` always returns an **endTrace** structure carrying the communicated restart point. This restart point can then be used to restart normal interpretation from the point of the end of this loop.

Trace Execution. In the trace execution phase (TE), the tracer is executing a previously recorded trace. Figure 4c lists the corresponding rules. Note that a guard instruction is a normal action.

Rule TE-NoSignal describes the case where a non-guard action is applied, or a guard action that did *not* fail: an action from the trace is applied on the current program state, by calling `applyAction`, and an **actionStep** is returned that contains the resulting program state. The tracer then continues by swapping its program state and moving on to the next action. In effect, this means that execution of each consecutive action in the trace happens via the interpreter.

Rule TE-GuardFailure describes a guard failure. Execution switches back to normal interpretation, restarting from the point that corresponds with the guard failure. This point is determined by applying `restart` on the restart point given by the guard and the current program state, as described in Section 3.2.

In rule TE-TraceEnd, the end of a non-looping trace has been reached. The tracer restarts normal interpretation from some program point, determined by calling the `restart` function on the current program state and the restart point associated with the end of the trace.

Rule TE-RestartLoop handles reaching the end of a regular, looping trace, which means one full iteration of the loop is completed. The trace is restarted by looking up the full trace belonging to the label and replacing the current empty one.

3.4. Difference with Meta-tracing

Meta-tracing compilers, such as the RPython framework, do not directly trace the execution of a user-program, but rather trace the execution of a language interpreter, *while this interpreter executes the user-program* [4]. By

annotating their interpreters with certain *hints* [5], e.g., for detection of loops in the user-program, language developers can guide tracing and optimization of traces. The traces are then heavily optimized to produce efficient machine code corresponding to the relevant operations performed by the user program. This enables the interpreter's implementers to employ the benefits of trace-based compilation without having to create their own dedicated tracing compiler. Furthermore, this makes it possible for developers to rapidly create interpreters with an acceptable performance level [16].

In STRAF, the tracing interface described in Section 3.2 enables developers to compose STRAF with any interpreter satisfying this interface. The purpose of the tracing interface is also similar to the purpose of the hints provided in the language interpreters for the RPython framework. In these respects, STRAF resembles meta-tracing compilation frameworks such as the RPython framework. However, although STRAF's tracing interface and RPython's hints share the same purpose, their implementation and the extent to which both are used are significantly different.

The extent of the tracing interface is far greater than that of the RPython hints: the tracing interface not only enables detection of loops, but is also used to *generate* all instructions, including guard instructions, that must be recorded into the trace. This implies that, unlike meta-tracing, traces are interpreter-specific: instructions are not generated by the tracer but by the interpreter. As a consequence, the semantics of these instructions are generally opaque to the tracer. Optimization of traces must therefore be performed by the interpreter, as opposed to the tracing compiler in the RPython framework, implying that a developer of an interpreter is also responsible for the optimization of traces.

Figure 5 illustrates the difference between the STRAF framework and a general meta-tracing compiler, such as the RPython framework. Whereas in meta-tracing, the language interpreter is generally described as running on top of the tracing compiler, in STRAF, the interpreter runs on the same level as the tracer, with the tracer delegating regular program execution to the interpreter whenever required. Note also that the traces generated by STRAF are not machine code, but that they are optimized sequences of instructions to be executed by the interpreter.

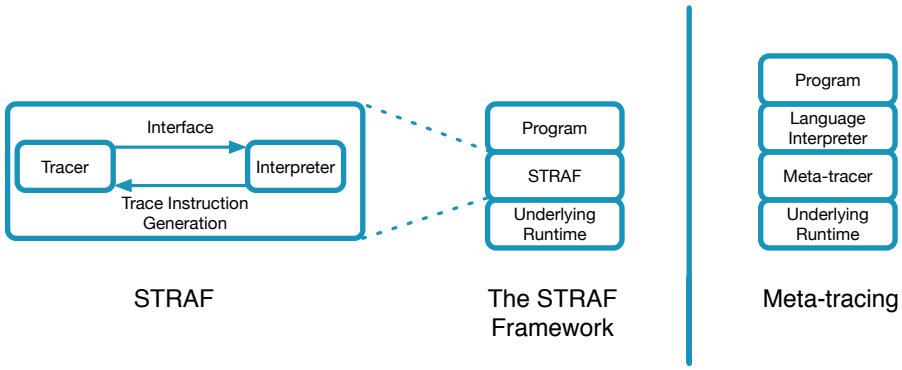


Figure 5: The difference between the STRAF framework and a meta-tracing framework

4. Evaluating STRAF's Generality

The strength of STRAF is its flexibility: it is *general* with respect to the set of interpreters that can be used, and it is *extensible*, i.e., the framework itself can be extended with new features. Our evaluation of STRAF therefore focuses on evaluating these two aspects instead of, e.g., performance. Sections 5 and 6 evaluate STRAF's extensibility. This section demonstrates the generality of STRAF by constructing interpreters for two different Scheme-like languages and integrating them into STRAF. This indicates that a variety of interpreters can be built that are in correspondence with the interface specified in Section 3.2 and, therefore, that STRAF does indeed accept multiple interpreters. At the same time, the presentation of these interpreters serves as a guide for how other suitable interpreters may be built.

4.1. Simple Scheme Interpreter

The first interpreter implements a non-trivial subset of the Scheme language. The interpreter is modeled after a variant of a CESK-machine [12] and hence satisfies the first requirement of our framework to model the interpreter as a state machine operating on some program state. Concretely, the program state consists of the standard control (C), environment (E), store (S), and continuation stack (K) components, as well as a value stack and value register. These last two components simplify implementing various transitions: the value register is used to store the value of the last evaluated subexpression, while the value stack is used to temporarily save the current environment as well as already evaluated arguments in a function call.

4.1.1. Evaluating Expressions

The interpreter determines how it should transition based on the content of its control component, which can either be an expression to be evaluated or a continuation to be followed. The `step` function of the interpreter checks the control and either calls `stepEval` with the expression to be evaluated or it calls `stepKont` with the corresponding continuation frame and the last value `v` that was evaluated. Both functions return an **interpreterStep**, as specified in the declaration of `step` in Section 3.2. Listing 3 exemplifies how an expression of the form `(set! variable exp)` is evaluated via the, partially elided, `stepEval` function.

```
1  def stepEval(e: SchemeExp): InterpreterStep = e match {
2      ...
3      case SchemeSet(variable, exp) =>
4          val actions = ActionSaveEnv() ::
5                        ActionEvalPush(exp, FrameSet(variable)) ::
6                        Nil
7          InterpreterStep(actions, SignalFalse())
8  }
9
10 def applyAction(state: ProgramState,
11                  action: Action): ActionReturn = action match {
12     ...
13     case ActionEvalPush(exp, frame) =>
14         ActionStep(state.copy(control = ControlExp(exp),
15                     kstack = state.kstack.push(frame)))
16 }
```

Listing 3: Evaluating a `set!`-expression.

In the case of a `set!` expression, the returned **interpreterStep** includes an `ActionSaveEnv()`, for saving the current environment on the value stack, and an `ActionEvalPush(exp, FrameSet(variable)))`, for simultaneously replacing the control component by the expression `exp`, as its value will have to be computed next, and pushing the continuation `FrameSet` on the continuation stack. Finally, the `SignalFalse` component indicates that the evaluation of a `set!` expression cannot trigger the beginning nor the end of a loop directly.

The returned actions are consecutively applied via the `applyAction` function. In the case of the Scheme interpreter, the actions are data structures to be interpreted. For example, an `ActionEvalPush` is handled by returning an **actionStep** (which is one possible *ActionReturn*) containing a copy of

the input program state with the continuation pushed onto the stack `kstack` and the control component replaced by the expression `exp`.

This example demonstrates how interpretation of a program can be decomposed into *selecting* which actions to use (`step`) and *applying* them (`applyAction`).

4.1.2. Loops

This subset of Scheme does not offer iterative looping constructs such as `for`. Loops in an execution therefore stem from recursion, as is the case for the factorial function depicted in Listing 4.

```
1 (define (fac n)
2   (if (< n 2)
3       1
4       (* n (fac (- n 1)))))
```

Listing 4: A factorial function.

Since the interpreter cannot generally know whether a function is recursive, it signals the possible start of a loop during the evaluation of *every* function application. Function application starts when all of its arguments are evaluated; arguments are evaluated by consecutively pushing and popping `FrameFunCallArgs` continuation frames, as depicted in Listing 5. Each `FrameFunCallArgs` contains a reference to the evaluated operator, i.e., the function to be applied, and the list of arguments yet to be evaluated. If there is still an argument `arg` left to be evaluated, `stepKont` returns an `ActionEvalPush` to evaluate this argument next. If no arguments remain, the interpreter starts evaluating the function's body: it moves to the first expression of the function's body and it pushes a `FrameFunBody` to evaluate the rest of the body afterwards. Since this is the proper start of the function application, the interpreter passes a **startLoop** tracing signal along in the **interpreterStep**. The label of the signal corresponds to the AST of the body of the function to be applied. If this signal causes the tracer to start recording, tracing continues until the interpreter reaches the start of this function again, as this indicates that one iteration of the loop has been completed.

4.1.3. Non-Looping Traces

Listing 6 depicts how the **endLoop** tracing signal is sent when the interpreter reaches the end of a function application, i.e., when the list of ex-

```
def stepKont(v: Value, frame: Frame): InterpreterStep = frame match {
   ...
   case FrameFunCallArg(fun, arg : args) =>
      val acts = ... :+
                 ActionEvalPush(arg, FrameFunCallArg(fun, args)))
      InterpreterStep(acts, SignalFalse)
   case FrameFunCallArg(fun, Nil) =>
      val acts = ... :+
                 ActionEvalPush(fun.body.head,
                                FrameFunBody(fun.body,
                                             fun.body.tail)
      InterpreterStep(acts, SignalStartLoop(functionValue.body))
}
```

Listing 5: Evaluating a function application.

pressions still to be evaluated in the function's body is `Nil`, while handling a `FrameFunBody` continuation frame. The label used in this signal is once more the full AST of the function's body, which was passed via the `FrameFunBody` continuation. As specified in Section 3.3, an **endLoop** signal must carry a restart point for restarting normal interpretation after completing the execution of the non-looping trace, so a `RestartTraceEnded` structure is included in the signal.

```
def stepKont(v: Value, frame: Frame): InterpreterStep = frame match {
   case FrameFunBody(body, Nil) =>
      InterpreterStep(..., SignalEndLoop(body, RestartTraceEnded()))
}
```

Listing 6: Completing a function application.

4.1.4. Guards

Listing 7 shows how guard instructions are inserted into the trace in STRAF. The listing depicts the evaluation of `if`-expressions, at the point at which the predicate has already been evaluated.

The interpreter checks the value of the evaluated predicate and determines whether to evaluate the consequence or the alternative branch. It adds a guard instruction that corresponds to the taken branch: if the condition was `true`, the interpreter returns `ActionGuardTrue` and passes a reference to the other branch, i.e., the branch `alt` that was *not* taken.

Listing 8 shows how such an `ActionGuardTrue` is handled. When this guard instruction is reached during the execution of the trace, the value of

```scala
case class ActionGuardTrue(rp: RestartPoint) extends Action
case class RestartGuardIf(exp: SchemeExp) extends RestartPoint

def stepKont(v: Value, frame: Frame): InterpreterStep = frame match {
   case FrameIf(cons, alt) =>
      if (v.isTrue()) {
         val actions = ActionGuardTrue(RestartGuardIf(alt)) :: ...
         InterpreterStep(actions, SignalFalse)
      } else { ... }
}
```

Listing 7: Continuing the evaluation of an if-expression.

the condition is stored in the value register, similar to how the condition's value was stored there during the recording of the trace. The value register is therefore checked: if the value was again `true`, nothing needs to be done so an **actionStep** is returned. Otherwise, the guard has failed so a **guardFailed** is returned.

```scala
def applyAction(state: ProgramState,
                action: Action): ActionReturn = action match {
   case ActionGuardTrue(rp) =>
      if (state.v.isTrue()) {
        ActionStep(this)
      } else {
        GuardFailed(rp)
      }
}
```

Listing 8: Handling an `ActionGuardTrue`.

4.1.5. Restarting

Listing 9 depicts part of the interpreter's implementation of the interface's `restart` function, for generating new program states based on a restart point and the current program state. If the restart point is an `RestartGuardIfFailed` (cf. Listing 7), it contains a reference to the branch that was *not* taken during the recording of the trace, and `restart` must only generate a copy of the input program state with its control component replaced by the given branch.

The implementation of guards and the `restart` function demonstrate that it is feasible to provide the functionality of trace guards by including a restart point structure and a `restart` function.

```
def restart(state: ProgramState,
            rp: RestartPoint): ProgramState = rp match {
   case RestartGuardIfFailed(exp) =>
      state.copy(control = ControlExp(exp))
}
```

Listing 9: Partial implementation of the `restart` function.

4.2. Non-deterministic Ambeval Interpreter

To further demonstrate the generality of STRAF, we instantiate it with a second interpreter: an implementation for Abelson and Sussman's non-deterministic *ambeval* [1, Chapter 4].

4.2.1. Introduction

We first exemplify the non-determinism supported by this interpreter before discussing its implementation. Listing 10 shows a function that, upon exhaustive backtracking, returns all pairs of elements from two lists of which the sum is prime. It relies on the predefined function `an-element-of` which selects an element from the given list, and returns another element upon backtracking.

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

Listing 10: Example of a non-deterministic program [1, Chapter 4].

Ambiguous programs make use of a primitive `amb` expression, which selects a value among its arguments, creating a choice point in the execution of the program. The `an-element-of` function passes its input list to an `amb` expression to select *some* element from this list. `require` evaluates the given `prime?` predicate and causes evaluation of the program to *fail* when the predicate is false. When the program fails, execution backtracks to the last choice point: in this case, to the point where a value for the variable `b` was chosen. This causes `b` to be bound to a new element of the list. When no more elements remain, execution backtracks further to the definition of `a` where the process is repeated. The program can therefore be thought of as *non-deterministic*; any possible value is considered for each *ambiguous*

variable but only those values that satisfy all requirements are eventually used.

4.2.2. Implementation

The ambeval interpreter is challenging due to the possible interactions between backtracking and tracing. Its implementation is modeled once more after a CESK machine, with the exception that a separate *failure continuation stack* complements the regular continuation stack in the interpreter states. When the interpreter encounters an `amb`-expression, it pushes a `FrameAmb` continuation on this new stack. When execution fails, the interpreter pops a continuation from this same stack such that it can continue from the last `amb`-expression with another value.

To restart execution from the last `amb`-expression, the interpreter must undo any changes made in the meantime. For instance, variables that have since been defined should be removed from the environment. To enable undoing such actions, for each action that is applied, an opposite action is wrapped in a `FrameUndoAction` and pushed onto the failure continuation stack, as illustrated by Figure 6. When a failure is triggered, the interpreter executes the undo-actions saved on the failure stack, thereby restoring the program state from the time at which the `amb`-expression it is restarting from was evaluated. Eventually, the interpreter will pop the `FrameAmb` continuation from the stack, at which point stack rewinding is complete.

In general, adapting the interpreter such that it saves these undo-actions on the failure continuation stack does not interfere with tracing. The ambeval interpreter traces functions in a way that is identical to the previous interpreter: by sending a **startLoop** signal to the tracer upon entering the body of a function and sending an **endLoop** signal upon its exit. However, care must be taken during stack rewinding when a function is being traced. If the execution were to backtrack *behind the function call that is being traced*, tracing should be aborted as this situation is similar to exiting from a function before the function loops. Listing 11 exemplifies how undo-actions are applied and how the **endLoop** signal is sent.

Section 4.1.2 explained that the interpreter pushes a `FrameFunBody` onto the regular continuation stack when starting a function application. When this continuation is therefore popped again while backtracking, the interpreter has reached the point in the program at which it started evaluating the function application. If the tracer is tracing the function application that it is now returning from, tracing should stop here, as it would backtrack

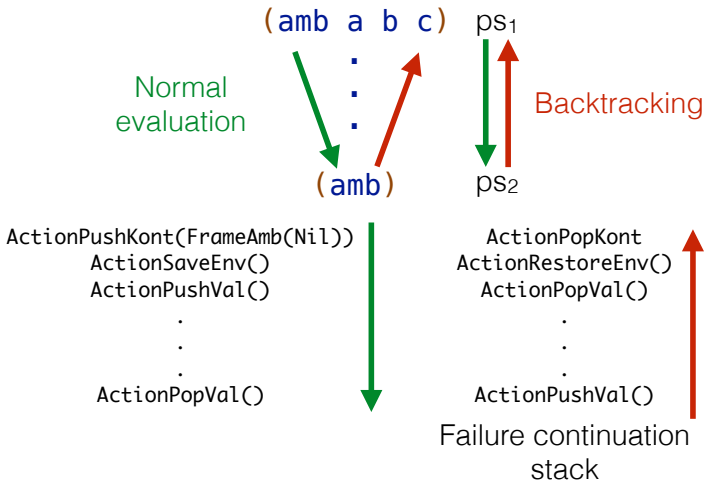


Figure 6: Undoing actions after execution has failed.

```
def stepKont(v: Value, frame: Frame): InterpreterStep = frame match {
   // Rewinding the failure continuation stack
   case FrameUndoAction(reverseAction) => reverseAction match {
      // Undoing a push-continuation
      case ActionPopKont() => getTopKont() match {
         // Undoing a push of a FrameFunBody continuation
         case FrameFunBody(body, Nil) =>
            InterpreterStep(..., SignalEndLoop(body, RestartTraceEnded()))
      }
   }
}
```

Listing 11: Backtracking out of an function application.

out of the function otherwise. The interpreter therefore sends an **endLoop** signal with a restart point of the form `RestartTraceEnded`.

No further changes need to be made to this Ambeval interpreter to make it satisfy the required interface.

4.3. Conclusion

The interpreters presented here demonstrate the variety of interpreters that can be plugged into STRAF, and exemplify executable implementations of the structures and signals described in Section 3.2. Together, they serve to demonstrate the generality of STRAF: its tracer can be reused to construct runtimes for different languages.

5. Optimizing Traces

We now provide a first demonstration of STRAF's *extensibility* by using its `optimize` hook to implement a set of optimizations that are common in the literature. For brevity, we give only a high-level description of the added optimizations, but their implementation is available online.[2] The traces on which we apply these optimizations are recorded by the simple Scheme runtime presented in Section 4.1.

Optimization of traces is encoded in the interpreter's interface via the `optimize` function, which takes a trace as input as well as the program state that was observed by the tracing machine when it started recording the given trace. As the application of actions is deterministic,[3] saving the program state that was observed at the start of the trace recording enables the optimizer to reconstruct, if necessary, each program state as it could have been observed while applying the corresponding actions during the recording of the trace. These program states provide the optimizations with all available concrete information, such as the contents of the store and the environment and hence the values that were observed for all variables in the program. The states can be discarded after completing the optimizations.

We implemented six different optimizations. Four of these represent well-known and widely used optimizations in the domain of (trace-based) JIT compilation:

Constant folding (O1) [9] Applications of arithmetic primitives that only use constants as arguments are replaced by the resulting value.

Arithmetic operations type specialization (O2) [7] Applications of generic arithmetic primitives, e.g., a generic plus operation, are optimistically replaced by the equivalent type-specialized operation, e.g., a plus operation specialized for floating point operands, if it was observed that all of its arguments belong to the same type. A guard is inserted to verify whether the types of the arguments remain the same at run time.

Variable folding (O3) The set of all free variables in a trace, i.e., the set of variables that are neither defined nor assigned to inside the trace,

[2] `https://github.com/mvdcamme/scala-am/blob/master/src/main/scala/tracing/SchemeTraceOptimizer.scala`

[3] In practice, there are some instances of non-determinism, e.g., `random`, so the framework includes additional information for some actions while recording a trace.

is computed. The trace is extended with a loop-invariant header containing, for each variable, an action for saving the current value of the variable in a specified register. Each *read* instance of these variables is then replaced by an action for looking up this variable in the register, thereby avoiding a more costly double lookup of the variable through the environment and the store.

Action merging (O4) Some actions that are likely to appear immediately behind each other in a trace are merged. Applying the merged action then has the same result as applying both actions separately. Increasing the granularity of actions decreases the total time spent in dispatching actions. For example, an action for looking up a value could in practice likely be followed by an action for popping a continuation from the continuation stack; these actions could hence be merged into one action to perform both operations. The effect of the granularity of opcodes in traces was previously described in [8].

In addition, we implemented two optimizations that respectively remove redundant saves and restores of the environment (**O5**), and pushes and pops of the continuation stack (**O6**).

Listing 12 demonstrates how the type specialization optimization, can be implemented and how it uses the starting program state to compute the state that would have been recorded for *each* action in the trace. This state is then used to retrieve the operands of arithmetic operations. The optimization then checks whether each operand is of the same type and if so replaces the generic operation by its equivalent type-specialized operation instead.

The actual `optimize` function pipelines each of these six optimizations, passing along its input program start state to each individual optimization that requires it. The purpose of adding these optimizations is not to provide a performant execution environment for Scheme, but to demonstrate that STRAF is sufficiently extensible to support them. The `optimize` hook with its two input parameters sufficed to implement all six optimizations, without requiring changes to the framework itself. All implementations together, moreover, amount to a mere 500 lines of Scala code.

5.1. Benchmark Results

To illustrate the effectiveness of the listed optimizations, we include Figure 7, depicting the median time required, with the 95% confidence intervals

```
def typeSpecialize(trace: Trace, start: ProgramState): Trace = {
  /*
   * replaceActions takes as input a tuple of an action in the
   * trace and its associated program state (i.e., the state achieved
   * by consecutively applying each action up til now in the trace
   * on the given start-state) and returns either a new, more specific,
   * action if the action corresponds to the application of an
   * arithmetic operation on a set of operands all of the same type,
   * or the same action otherwise.
   */
  def replaceAction(tuple: (Action, ProgramState)): Action = {
    val (action, state) = tuple
    // The operands are saved on the stack in the state
    val operands = getOperands(action, state)
    if (isArithmeticOperation(action) && (allSameType(operands)) {
      // We call replaceOperation to replace this action
      // with its equivalent, type-specialized action.
      // The generation of a guard is not depicted here.
      replaceOperation(action, typeOf(operands))
      // Action cannot be specialized
    } else {
      action
    }
  }
  // We compute the consecutive program states that correspond
  // with applying each action in the trace via weaveStates, zipping
  // program states and actions together.
  val zipTrace: List[(Action, ProgramState)] = weaveStates(trace, start)
  zipTrace.map(replaceAction) }
```

Listing 12: Pseudo-implementation of type specialization.

included, for STRAF to execute a set of benchmarks when no optimizations are applied. These results serve as the baseline for Figure 8, which depicts the median execution time, normalized with respect to this baseline, of executing the same benchmarks, first with each of the six optimizations enabled individually (O1-6), and finally with all optimizations enabled (All).

The benchmarks were executed on an Intel I7-4870HQ CPU at 2.50GHz with 6MB cache and 16GB RAM, running 64bit OS X 10.11.4 and Scala 2.11.7 on the Java Hotspot VM 25.92. Each benchmark is executed 30 times in a separate JVM. Measurements are taken after JVM warmup was completed: we observed stable measurements after two iterations of execution of the program. The median of the results and its 95%-confidence interval are reported.

While executing these benchmarks, we used the tracing compilation features described in Section 6 by defining a tracing threshold of 10 and enabling guard tracing.

The results show that the first four optimizations generally do not provide significant performance improvements. Removing redundant saves and restores of the environment (O5) and removing redundant pushes and pops of continuation frames (O6), do offer a small performance increase in some cases. Notably, the collatz benchmark is significantly slower when applying these two optimizations. This benchmark produces one large trace of which the execution always leads to a guard failure quickly. Thus, the overhead of applying both optimizations on this trace is never recouped, as execution of the trace is quickly aborted.

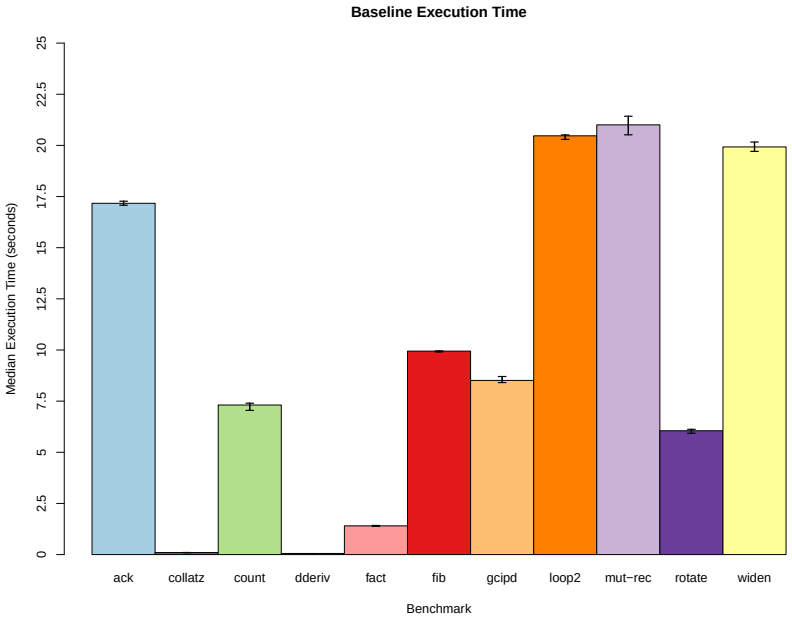


Figure 7: Median execution time for the baseline execution (no optimizations applied).

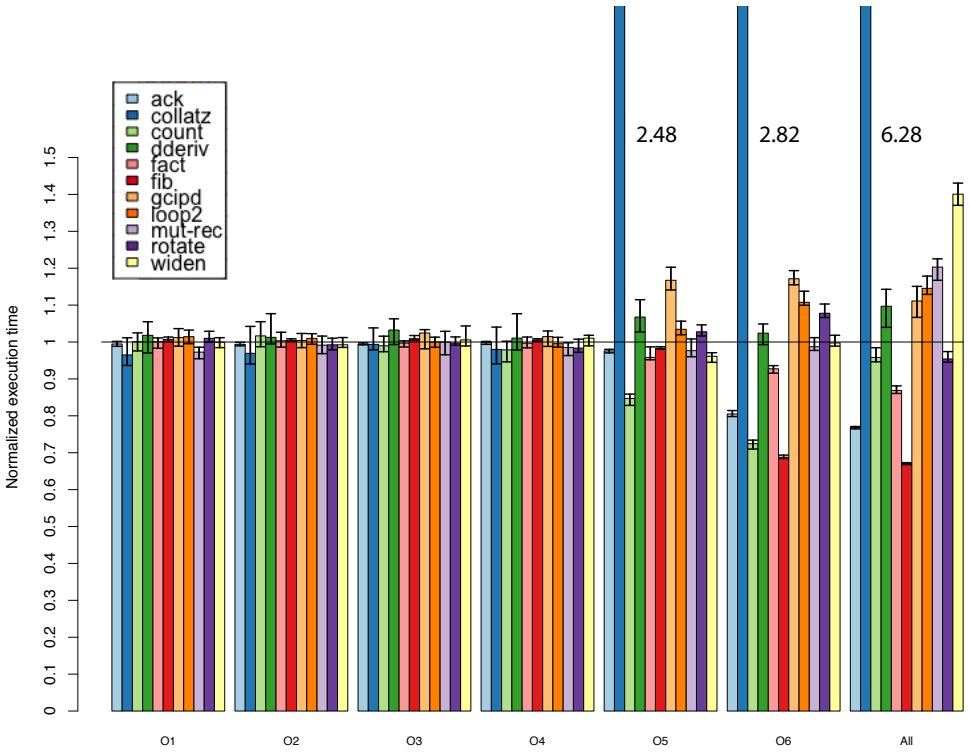


Figure 8: Normalized execution times with each of the six optimizations applied individually (O1-6), and with all optimizations applied (All).

Figure 9 depicts the total number of traces that were recorded during the execution of each benchmark. This number includes both regular traces and traces produced after a guard failure has occurred (see Section 6.2). Note that both the trace recording and optimization process as well as the benchmarks themselves are completely deterministic. The number of recorded traces therefore does not vary over time. Also note the high number of (guard) traces produced by the collatz benchmark indicating that traces are quickly aborted due to failing guards, which in turn leads to more traces being recorded, so that the computational overhead of optimizing the trace is never recouped.

Regardless of their effectiveness, adding these optimizations to STRAF indicates that the framework is extensible, and that optimizations can be implemented by using the `optimize` hook.

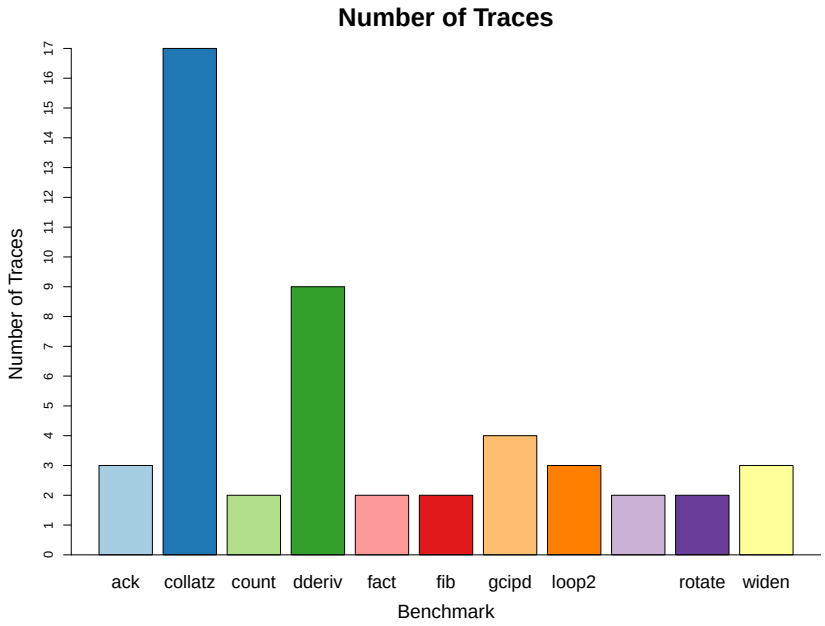


Figure 9: The total number of traces recorded during the execution of the benchmark.

5.2. Additional Performance Metrics

Since we experiment with a highly conceptualized and thus comparably inefficient interpreter, the optimizations are not as effective as they are in optimized systems. Inspired by Brunthaler [6], we thus also measure their effects on a different set of metrics beside performance and we compare with the baseline, unoptimized execution of the benchmarks. This gives us a notion of the effect of the optimizations on the traces.

1. The effectiveness of the action merging optimization (**O4**), the removal of redundant environment saves and restores (**O5**), as well as the removal of redundant continuation pushes and pops (**O6**), is measured by total length of the generated traces. For brevity, we only report the combination of the three optimizations.
2. The type specialization (**O2**) optimizations is measured by the number of non-type specialized arithmetic operations that are applied.
3. The variable folding optimization (**O3**) is measured by the number of variable lookups.

Note that all figures depicting the baseline results use a log-scaled y-axis.

5.2.1. Trace lengths

Figure 10b illustrates that the action merging optimization, the removal of redundant environment saves and restores and the removal of redundant continuation pushes and pops is effective at reducing the length of traces by at least 50% in all cases.

5.2.2. Generic arithmetic operations

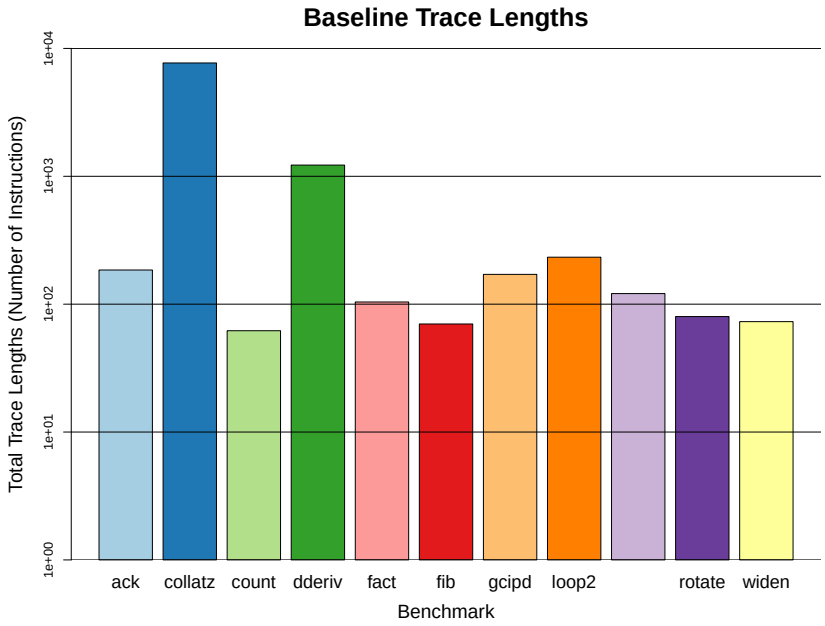
Figure 11 depicts the total number of generic, non-type-specialized arithmetic operations that are executed during the total lifetime of each program, both while executing a trace or while interpreting the program. When enabling the type-specialization optimization (Figure 11b) the number of generic operations that are executed in the ack, count, loop2, mut-rec and rotate programs, drops down to almost zero, as all arithmetic operations that take place in a trace are successfully type-specialized. The only generic arithmetic operations left for these benchmarks are those that are executed outside of a trace. In the case of the fact, fib, gcipd and widen benchmarks, the number of generic arithmetic operations also significantly decreases. As the dderiv benchmark does not use any arithmetic operations inside a traced part of the program, the optimization is ineffective.

5.2.3. Variable Lookups

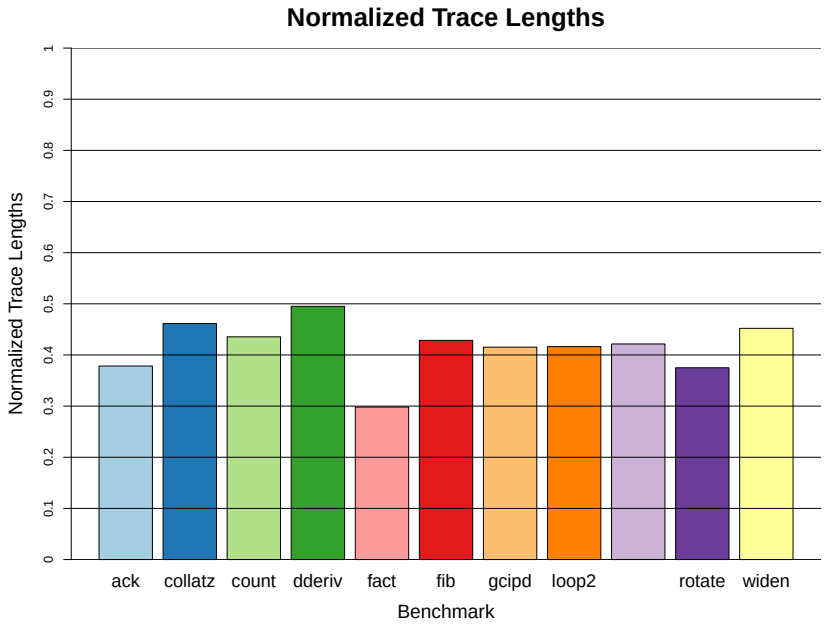
Figure 12 depicts the total number of times a variable is looked up during the execution of each program, again both while executing a trace and while interpreting the program. As the variable folding optimization avoids lookups of free variables in the trace by placing these variables in read-only registers before executing the trace, we expect the number of variable lookups to drop significantly, depending on the amount of free variables. As shown in Figure 12b, the number of variable lookups indeed significantly drops across all benchmarks, from 13% for the collatz benchmark, to 67% for the fact benchmark.

5.2.4. Constant Folding

In the case of this limited set of benchmarks, the constant folding optimization has no effect on any benchmark, as none of the programs contain any arithmetic expression that only makes use of constant values.

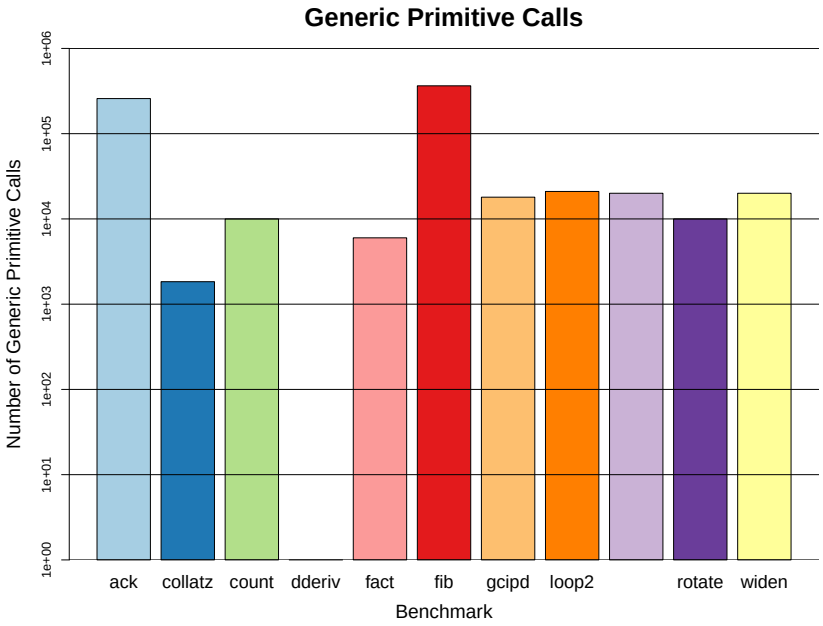


(a) The total, combined length of all recorded traces when no optimizations have been enabled.

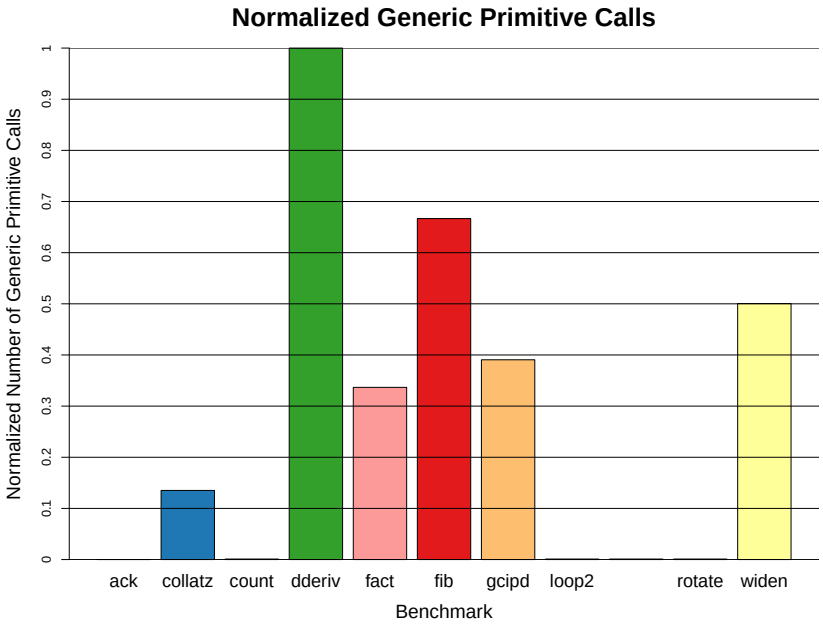


(b) The total, combined length of all recorded traces (normalized to Figure 10a) with optimizations **O4**, **O5** and **O6** have all been enabled.

Figure 10: Measuring the total length of all, combined traces.

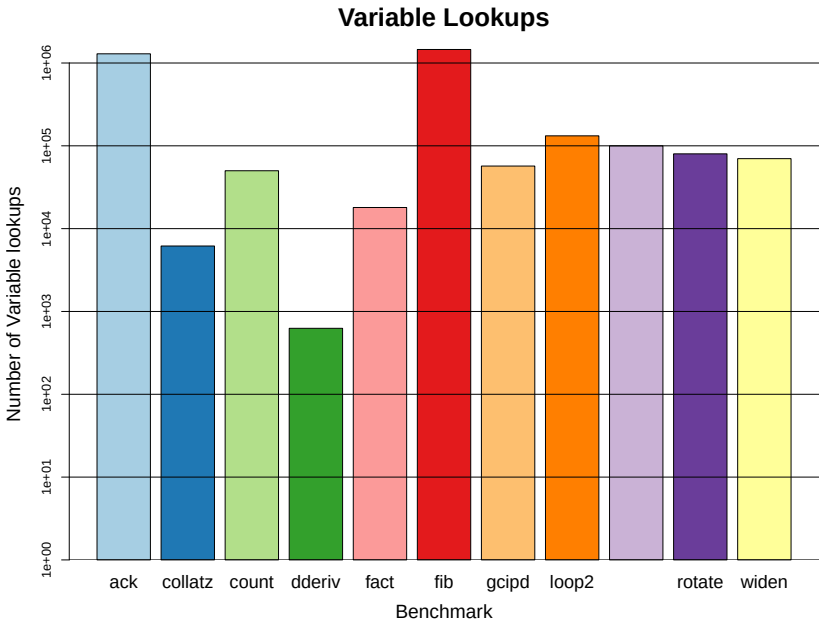


(a) The number of generic, i.e., not type-specialized, arithmetic operations when no optimizations are enabled.

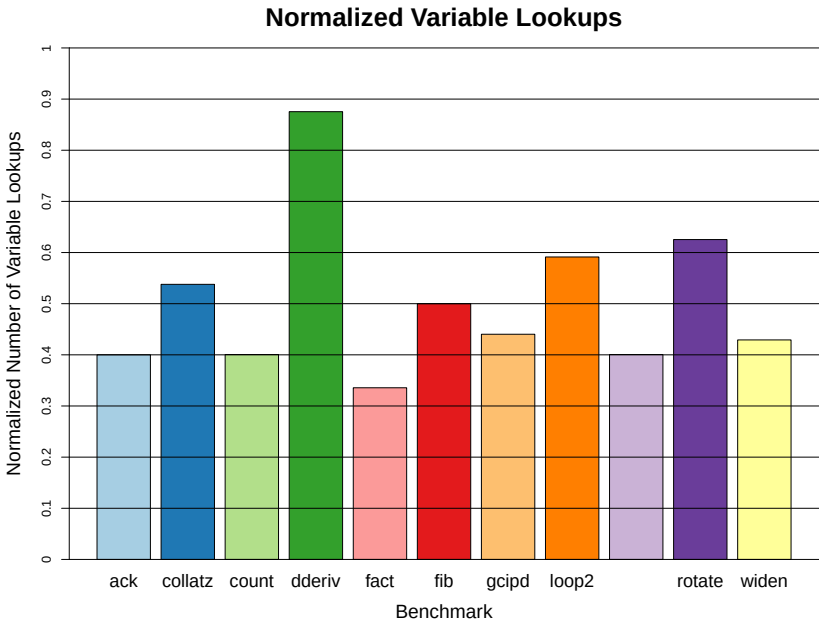


(b) The number of generic, i.e., not type-specialized, arithmetic operations (normalized to Figure 11a) with only optimization **O2** enabled.

Figure 11: Measuring the number of generic arithmetic operations.



(a) The number of variable lookups when no optimizations are enabled.



(b) The number of variable lookups (normalized to Figure 12a) with only optimization **O3** is enabled.

Figure 12: Measuring the total number of variable lookups.

6. Evaluating STRAF's Adaptability

This section evaluates STRAF's *extensibility* by adding two mechanisms that are widely used in trace-based JIT compilers. Specifically, we add a heuristic for detecting hot loops and a mechanism for starting traces from guard failures. These mechanisms build on the framework itself, so we describe them as variations on the semantics presented in Section 3, although they are directly included in our implementation. By demonstrating how STRAF can be extended with these features, we provide an intuition of the effort for further extending or adapting STRAF. As we will show, including these mechanisms requires only minimal changes.

6.1. Hot Loop Detection

Tracing compilation is most effective when applied to the parts where a program spends most of its time (i.e., the *hot* parts) [14]; optimizing rarely executed parts can reduce overall performance, because of the time it takes to do the tracing and optimizations [13].

In STRAF, tracing of a loop starts immediately once this loop is executed for the first time. Although this is adequate for a basic implementation, it does not correspond well to the state-of-the-practice in tracing JIT compilers. Therefore, the first evaluation of the adaptability of STRAF is a heuristic to detect hot loops in a program's execution. A program loop is called "hot" once it has been completed at least a fixed amount of times, i.e., once a *threshold* has been exceeded. This type of hot loop detection is used for instance in HotpathVM [13], TraceMonkey [14], and SPUR [3].

6.1.1. Extending the Tracing Machine

To detect hot loops, we extend the tracing machine to count the number of iterations that have been completed for each loop, as shown in Figure 13. To this end, our extension implements a *LabelCounterMap* that associates a trace label to a counter, similar to how *TraceNodeMap* associates a label to a trace node. When the interpreter enters a loop, i.e., when it sends the **startLoop** signal, the counter for the loop's label is updated.

6.1.2. Semantics

To add hot loop detection, we need to adapt the tracing semantics. Specifically, we need to change how tracing is *started*, i.e., how we transition from the normal interpretation phase of trace execution to the trace recording

$$tc \in \mathit{TracerContext} ::= \mathbf{tc}(\mathit{Null} + \mathit{TraceNode},\ \mathit{TraceNodeMap},\ \mathit{LabelCounterMap})$$
$$L \in \mathit{LabelCounterMap} ::= \mathit{Label} \to \mathbb{N}$$

Figure 13: The updated tracing machine for hot loop detection.

phase. The execution of traces and tracing itself remain unchanged. We therefore only have to update the rules for normal interpretation by the tracing machine (cf. Figure 14).

In rule NI-FirstEncounter, a loop carrying a label is entered that has not yet been seen before. Hence, no corresponding entry in the list of label counters exists yet. The tracing machine therefore creates such a label counter, adds it to the list, and continues interpretation. Rule NI-FirstEncounter specifies the case in which a loop is entered with a label that is not yet hot; its counter is still below the threshold that is required to start tracing. The label's counter is updated and interpretation continues as before. In rule NI-LoopHot, a label is encountered that has become hot, causing the tracing machine to start tracing. Note that rules Ni-ContinueInterpreting and NI-StartExecuting have remained unchanged and are therefore still in effect.

6.2. Guard Tracing

Guard tracing mitigates the performance penalties of guard failures. Normally, a guard failure aborts the execution of a trace and normal interpretation is restarted; the compiled and heavily-optimized trace is abandoned to interpret unoptimized code. Additionally, returning back to the interpreter is a slow operation on its own [2, 8].

Guard tracing enables tracing from the point of a guard failure. Whenever the guard fails again, execution is switched directly to the new trace instead of restarting interpretation. Guard tracing is used for instance by RPython [18], Dynamo [2], and SPUR [3]. In practice, guard tracing is only started when the guard failed often enough. For simplicity, we start tracing immediately after a guard failure. Although suboptimal in practice, this is adequate for a basic implementation of guard tracing.

We call a trace recorded for a guard failure a *guard trace*, and a trace recorded from a loop a *label trace*. When a guard failed and a guard trace is recorded, we say the trace with the failing guard *spawned* the guard trace.

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(t,\ \textbf{startLoop}(\mathit{lbl})) \\ T[\mathit{lbl}] \text{ is undefined} \\ L[\mathit{lbl}] \text{ is undefined}\end{array}}{\begin{array}{c}\textbf{ts}(\text{NI},\ \textbf{tc}(\mathit{Null},\ T,\ L),\ s,\ \mathit{Null}) \rightarrow \\ \textbf{ts}(\text{NI},\ \textbf{tc}(\mathit{Null},\ T,\ L[\mathit{lbl} \mapsto 1]),\ s',\ \mathit{Null})\end{array}}\ \text{NI-FirstEncounter}$$

Where $s' = \texttt{applyAction*}(s,\ t)$

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(t,\ \textbf{startLoop}(\mathit{lbl})) \\ T[\mathit{lbl}] \text{ is undefined} \\ k < \mathit{Threshold}\end{array}}{\begin{array}{c}\textbf{ts}(\text{NI},\ \textbf{tc}(\mathit{Null},\ T,\ L),\ s,\ \mathit{Null}) \rightarrow \\ \textbf{ts}(\text{NI},\ \textbf{tc}(\mathit{Null},\ T,\ L[\mathit{lbl} \mapsto k+1]),\ s',\ \mathit{Null})\end{array}}\ \text{NI-LoopNotHot}$$

Where $s' = \texttt{applyAction*}(s,\ t)$
$k = L[\mathit{lbl}]$

$$\frac{\begin{array}{c}\texttt{step}(s) = \textbf{interpreterStep}(a_1 : ... : a_n,\ \textbf{startLoop}(\mathit{lbl})) \\ T[\mathit{lbl}] \text{ is undefined} \\ k >= \mathit{Threshold}\end{array}}{\begin{array}{c}\textbf{ts}(\text{NI},\ \textbf{tc}(\mathit{Null},\ T,\ L),\ s,\ \mathit{Null}) \rightarrow \\ \textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(\mathit{lbl},\ a_1 : ... : a_n,\ s),\ T,\ L),\ s',\ \mathit{Null})\end{array}}\ \text{NI-LoopHot}$$

Where $s' = \texttt{applyAction*}(s,\ a_1 : ... : a_n)$
$k = L[\mathit{lbl}]$

Figure 14: The updated semantics of the normal interpretation phase.

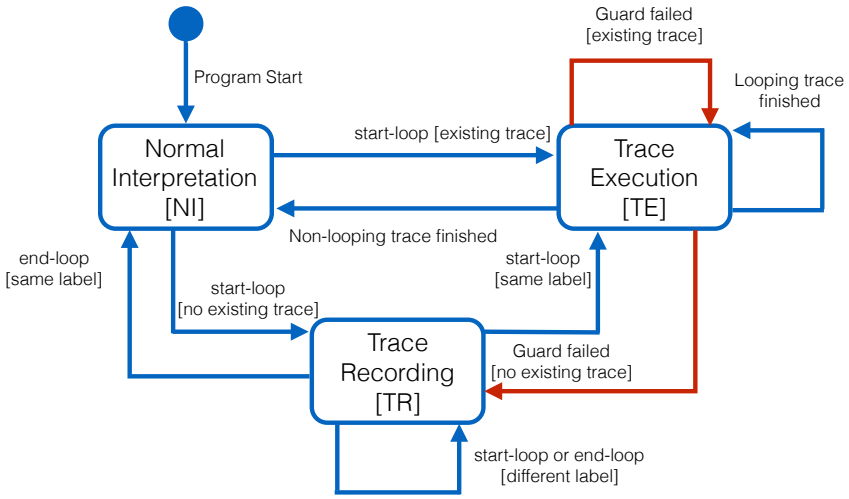


Figure 15: The tracing diagram updated with guard tracing semantics.

Figure 15 depicts how the tracing state diagram presented in Figure 2 is updated to accommodate these changes to the tracing model: the edge corresponding with a guard failure in a trace in the original diagram has been replaced with two edges, depicted in red, corresponding with the cases where a guard failure leads to either the recording of a new guard trace, or to the execution of such a trace.

6.2.1. Extending the Tracing Machine

For guard tracing, we first redefine how traces are identified. Originally, traces could be identified only through their label. Guard traces however can share labels, namely the label of the the trace that spawned the guard trace. To once again uniquely identify all traces, we use *trace identifiers* or trace ids. Figure 16 formalizes these trace ids. A label trace id **ltid** just wraps a label. A guard trace id **gtid** carries both a label and a *guard identifier* uniquely representing the failing guard. We also change *TraceNodeMap* and *LabelCounterMap* so that they map these trace ids, instead of the labels, to trace nodes and counters respectively.

As we build this feature on top of the previous extension, hot loop detection, all other parts of the tracing machine are identical to that of Section 6.1. Language developers must, however, provide a mechanism of their choosing for identifying individual guard instructions.

$$
\begin{aligned}
\mathit{tid} \in \mathit{TraceId} &::= \mathbf{ltid}(\mathit{Label}) \\
&\quad \mid \mathbf{gtid}(\mathit{Label},\ \mathit{GuardID}) \\
\mathit{gid} \in \mathit{GuardID} &::= \mathit{Identifier} \\
\mathit{tn} \in \mathit{TraceNode} &::= \mathbf{tn}(\mathit{TraceId},\ \mathit{Trace},\ \mathit{ProgramState}) \\
T \in \mathit{TraceNodeMap} &::= \mathit{TraceId} \to \mathit{TraceNode} \\
L \in \mathit{LabelCounterMap} &::= \mathit{TraceId} \to \mathbb{N}
\end{aligned}
$$

Figure 16: The updated tracing machine for guard tracing.

6.2.2. Guard Instructions

Making guards carry an identifier can be accomplished by making the identifier a parameter of the guard instruction. We create this identifier alongside the guard itself; i.e., when the guard is inserted into a trace by the tracer during trace recording. When a **guardFailed** signal is returned after the execution of an action, we include the guard identifier of the guard that just failed:

$$\mathbf{guardFailed}(\mathit{RestartPoint},\ \mathit{GuardID})$$

6.2.3. Semantics

We now describe how the semantics of the tracing machine need to be updated. As guard tracing has no effect on the normal interpretation of the program, we only alter the evaluation rules for trace recording and trace execution as depicted by Figure 17.

Trace Recording. Recording a guard trace is identical to recording a label trace, up to the level of stopping their recording. Recording is stopped when the interpreter sends a **startLoop** or **endLoop** signal carrying the label of the trace that initially spawned the guard trace that is now being recorded. The updated trace recording rules are therefore almost identical to the old rules, except that they now account for the fact that a trace identifier can take two different forms. For brevity's sake we fuse these forms together by defining a function `label` which, given a trace id, extracts the label used in that trace id. Labels are components of both kinds of trace ids, so retrieving the label of a trace id is always possible.

$$\frac{\texttt{step}(s) = \textbf{interpreterStep}(t',\ \textbf{startLoop}(lbl))}{\begin{array}{c}\textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(tid,\ t,\ s_s),\ T,\ L),\ s,\ Null) \rightarrow \\ \textbf{ts}(\text{TE},\ \textbf{tc}(Null,\ T[tid \mapsto tn],\ L),\ s',\ \textbf{tn}(\textbf{ltid}(lbl),\ t',\ \_))\end{array}}\ \text{TR-SameStart'}$$

Where $s' = \texttt{applyAction*}(s,\ t')$
$tn = \textbf{tn}(tid,\ \texttt{optimize}(t, s_s),\ s_s)$
$lbl = \texttt{label}(tid)$

(a) Trace recording

$$\frac{\texttt{applyAction}(s,\ a) = \textbf{guardFailed}(rp,\ gid)}{\begin{array}{c}\textbf{ts}(\text{TE},\ tc,\ s,\ \textbf{tn}(tid,\ a:t,\ s_s)) \rightarrow \\ \textbf{ts}(\text{TE},\ tc,\ s',\ tn)\end{array}}\ \text{TE-GuardTraceExists}$$

Where $s' = \texttt{restart}(rp,\ s)$
$tn = T[gid]$

$$\frac{\begin{array}{c}\texttt{applyAction}(s,\ a) = \textbf{guardFailed}(rp,\ gid) \\ T[gid] \text{ is undefined}\end{array}}{\begin{array}{c}\textbf{ts}(\text{TE},\ \textbf{tc}(Null,\ T,\ L),\ s,\ \textbf{tn}(tid,\ a:t,\ \_)) \rightarrow \\ \textbf{ts}(\text{TR},\ \textbf{tc}(\textbf{tn}(\textbf{gtid}(lbl,\ gid),\ \phi,\ s'),\ T,\ L),\ s',\ Null)\end{array}}\ \text{TE-RecordGuardTrace}$$

Where $s' = \texttt{restart}(rp,\ s)$
$lbl = \texttt{label}(tid)$
ϕ represents the empty list

$$\frac{}{\begin{array}{c}\textbf{ts}(\text{TE},\ tc,\ s,\ \textbf{tn}(tid,\ \phi,\ s_s)) \rightarrow \\ \textbf{ts}(\text{TE},\ tc,\ s,\ tn)\end{array}}\ \text{TE-RestartLoop'}$$

Where $lbl = \texttt{label}(tid)$
$tn = T[tid]$
ϕ represents the empty list

(b) Trace execution

Figure 17: Transition rules for enabling guard tracing.

Only rule TR-SameStart requires some adaptation. Recall that once recording of a trace is finished with a **startLoop** signal, execution of the new trace is started immediately. This still holds true after the addition of guard traces, but finishing the recording of a guard trace causes the tracer to start executing the label trace that initially spawned the guard trace, instead of the guard trace itself.

Rules TR-ContinueTracing and TR-SameEnd remain the same, except that the label of the recorded trace is first extracted via the `label` function before it is compared with the label used in the **startLoop** or **endLoop** signal.

Trace Execution. The trace execution phase must be updated to account for the novel handling of guard failures.

When a guard fails, the tracer must check whether a guard trace for the corresponding guard identifier already exists. Rule NI-GuardTraceExists states that if a guard trace exists, the tracer swaps the trace that is currently being executed for this guard trace. Rule NI-RecordGuardTrace expresses that if there is no existing trace, the tracer starts tracing the guard. The guard trace id is constructed by combining the label of the trace being executed with the guard identifier carried back in the **guardFailed** signal. The tracer restarts interpretation by constructing a new program state from the restart point carried back in the guard and the current program state.

Note that we use the generic term *tid* to refer to the trace id of a trace node. It is entirely possible that a guard failure during the execution of a guard trace triggers the tracing machine to start recording a guard trace *for that guard trace*. In that case, the label of the new trace's id and the id of the trace that was aborted both equal the label of the label trace that spawned the initial guard trace.

Rule NI-RestartLoop' specifies that if the tracer reaches the end of a looping trace, no matter which kind of trace is being executed, the tracer restarts the loop by finding the label trace node associated to the label of the trace currently being executed. In other words, if the tracer has reached the end of a looping guard trace, it does not restart this guard trace itself, but rather the label trace that initially spawned this guard trace. Rules TE-NoSignal and TE-TraceEnd remain unchanged.

6.3. Conclusion

We have extended the implementation and the formal semantics of STRAF with a means to detect hot loops and with the capability to start recording traces from guard failures. Although our extensions correspond to straightforward incarnations of these techniques, they are testament to the flexibility of STRAF. This is important given its purpose as an enabler of future studies of various tracing compilation features. The changes required for these extensions are minimal, suggesting that similar features might be added without extensive modifications, too.

7. Discussion

STRAF aims to be a minimalistic, but extensible framework for rapid prototyping of various techniques related to trace-based JIT compilation — such as program analysis and trace optimization. Experiments that have been proven feasible in our framework, can then be transposed to frameworks such as the the Mu Micro VM [22] to evaluate their performance in an environment that more closely resembles real-world runtimes.

As an example for such an experiment, we plan to investigate whether and how dynamic compilation could benefit from advanced static knowledge of the program. We hypothesize that optimizations could benefit from extending their scope of available information with data that lies beyond the boundaries of the trace. To this end, STRAF is integrated into SCALA-AM [19], a Scala abstract interpretation framework using the AAM methodology [20] which enables an abstract machine, e.g., a CESK machine [12], for a language to double as both a concrete interpreter and as a static analyzer for this language. We can therefore reuse the same abstract machine for performing static analysis of a program, and for plugging the abstract machine in as a concrete interpreter into STRAF. As an abstract machine always models a state machine, the abstract machine already satisfies one of the requirements for interpreters that were outlined in Section 3.2.

Sections 5 and 6 demonstrated how new features, respectively an optimization mechanism and two additional tracing mechanisms, can be added to the compiler. Although we cannot predict the extent of the changes that will be required for other extensions, they were minimal for the hot loop detection and guard tracing techniques.

However, STRAF also has its limitations. Tracing can only be applied on interpreters that are state-based and which satisfy the interface defined

in Section 3.2. Much of the responsibility for the resulting compiler's correctness lies with the language implementers. They are required to build an interpreter that sends the correct tracing signals to the tracer at the right times. This is necessary for generating the proper guards and for correctly restarting normal interpretation after a guard failure.

Furthermore, with our chosen design of giving maximal flexibility to the interpreters on how state and behavior are implemented, optimizations are currently bound to a specific interpreter. While similar interpreters could reuse optimizations, this is currently not possible in general.

Finally, in its current state, STRAF itself does not feature a native code compiler. While this avoids a significant amount of complexity, it also restricts experimentation to a conceptual level. We argue that this is indeed by design and simplifies prototyping of ideas and their early evaluation.

8. Related Work

We described an early formalization and Scheme implementation of STRAF in prior work [21]. This article refines that formalization, provides a reference implementation in Scala that is integrated into an existing framework for executing abstract machines using the AAM methodology [19], and demonstrates the frameworks' generality and extensibility. The former by composing it with two different interpreters, and the latter by adding six optimizations, a hot loop detection heuristic, and a guard tracing mechanism.

As related work, we consider the RPython, Truffle, and SPUR meta-compilation frameworks and formal models meant to investigate trace-based JIT compilation.

8.1. RPython and Truffle

Section 3.4 already compared STRAF and the RPython framework with respect to the difference to meta-tracing. Although RPython has some similarity to our model, its purpose is to create performant interpreters with JIT facilities. Our framework is aimed to study tracing compilation in general.

Similar to the RPython framework, the Truffle framework facilitates the development of performant abstract-syntax-tree (AST) interpreters [23]. It provides mechanisms for dynamic self-optimization of a program's AST structure. Furthermore, it uses the Graal compiler [17] to partially evaluate the ASTs at run time and then generate efficient native code for a modified version of Java's HotSpot VM [24].

SPUR [3] is a trace-based JIT compiler for Microsoft's Common Intermediate Language (CIL) [11], the intermediate language to which languages such as C# and VisualBasic are compiled. By first compiling these languages to a common intermediate language and then using SPUR to execute this intermediate language, only one JIT compiler has to be constructed to serve all languages that are translatable to CIL. In their evaluation, Bebenita et al. [3] demonstrated that compiling a Javascript program to CIL and subsequently executing the compiled code via SPUR achieves performance on par with a dedicated trace-based JIT compiler for Javascript.

While RPython, Truffle, and SPUR can all be used to investigate the effect of JIT compilation on user programs, one is limited in further experimentation. For instance, to study the effect of JIT compilation features, one has to adapt a highly complex framework that has grown over many years and is laced with performance compromises, making certain experiments hard, if not infeasible. In contrast, STRAF is designed for this purpose. Furthermore, we also provide a complete formal description of our framework to support precise reasoning over such experiments.

8.2. Formalized Tracing Models

Guo and Palsberg [15] and Dissegna et al. [10] set out to formally prove the soundness of certain optimizations on traces. To this end, they developed small, formal models of trace-based compilation. Although they are successful in proving soundness of optimizations and have delivered rather small tracing models, both models are tightly coupled to one particular execution semantics, i.e., one particular interpreter for some language. Any changes to the execution semantics also require extensive changes to the model of the tracing compiler itself. For the first model, detailed execution semantics are included, but for every rule in these semantics, a near-identical copy of the rule has to be specified: the first rule expresses how execution of an input program should proceed in the case the compiler is not tracing, while the second rule states how this should be done when a trace is being recorded. The second model suffers from similar disadvantages, in that their tracing compiler is difficult to adapt.

9. Conclusion and Future Work

STRAF is a framework for experimenting with trace-based JIT compilation for interpreters. It is designed to study the various effects tracing

compilation can have on the execution of programs. The framework is flexible both in the wide variety of interpreters it supports, and in the extensions to the basic tracing scheme that it supports.

We evaluated the first aspect with two interpreters and show how they can be plugged into STRAF. The first interpreter is for an applicative Scheme dialect, while the second one is an *ambeval* evaluator, which uses back tracking. This demonstrates that a wide range of languages can be supported.

The second aspect is evaluated by both extending an interpreter with an optimization mechanism and by adapting STRAF's semantics with extensions for two common features: a loop hotness detection mechanism to improve the selection of loops to trace, and a guard tracing mechanism to mitigate the performance penalty of aborting the execution of a trace. Although our extensions correspond to a naive implementation of these mechanisms, they indicate that researchers who wish to include additional mechanisms need only make minimal changes to the basic STRAF framework.

In future work, we wish to investigate how to employ static analyses to further improve dynamic compilation facilities. To this end, we integrated STRAF in a larger framework for developing static analyses via abstract interpretation.

Acknowledgements

We thank the reviewers for their close reading and helpful critique. We also thank Laurence Tratt for his extensive comments on an earlier version of this text. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1996.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *Proceedings of the ACM International OOPSLA Conference*, OOPSLA '10, 2010.

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th ICOOOLPS Workshop*, 2009.

[5] C. F. Bolz, A. Cuni, M. Fijlakowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, 2011.

[6] S. Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th ECOOP*, 2010.

[7] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, University of Irvine, Department of Computer Science, 2007.

[8] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International VEE Conference*, 2009.

[9] N. Corporation. Constant folding. `http://www.compileroptimizations.com/category/constant_folding.htm`. Accessed: 2016-05-24.

[10] S. Dissegna, F. Logozzo, and F. Ranzato. Tracing compilation by abstract interpretation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT POPL Symposium*, 2014.

[11] ECMA. International Standard ECMA-335, Common Language Infrastructure, 2012.

[12] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN POPL Symposium*, 1987.

[13] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International VEE Conference*, 2006.

[14] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN PLDI Conference*, 2009.

[15] S.-y. Guo and J. Palsberg. The essence of compiling with traces. *SIGPLAN Not.*, 46(1):563–574, Jan 2011.

[16] S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM International OOPSLA Conference*, 2015.

[17] OpenJDK. Graal project. URL `http://openjdk.java.net/projects/graal/`.

[18] D. Schneider and C. F. Bolz. The efficient handling of guards in the design of RPython's tracing JIT. In *Proceedings of the 6th ACM VMIL Workshop*, 2012.

[19] Q. Stievenart. Abstract machine experiments using Scala. `https://github.com/acieroid/scala-am`. Accessed: 2016-06-07.

[20] D. Van Horn and M. Might. Abstracting abstract machines. *ACM Sigplan Notices*, 45(9):51–62, Sep 2010.

[21] M. Vandercammen, J. Nicolay, S. Marr, J. De Koster, T. D'Hondt, and C. De Roover. A formal foundation for trace-based JIT compilers. In *Proceedings of the 13th WODA*, 2015.

[22] K. Wang, Y. Lin, S. M. Blackburn, M. Norrish, and A. L. Hosking. Draining the swamp: Micro virtual machines as solid foundation for language development. In *Proceedings of the 1st SNAPL Conference*, 2015.

[23] C. Wimmer and T. Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd SPLASH Conference*, 2012.

[24] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of Onward! 2013*, 2013.