A formal foundation for trace-based JIT compilers

Vandercammen, Maarten; Nicolay, Jens; Marr, Stefan; De Koster, Joeri Jean-Marie; D'Hondt, Theo; De Roover, Coen

Published in:
Proceedings of the 13th International Workshop on Dynamic Analysis (WODA)

DOI:
10.1145/2823363.2823369

Publication date:
2015

A Formal Foundation for Trace-based JIT Compilers

Maarten Vandercammen* Jens Nicolay* Stefan Marr† Joeri De Koster*
Theo D'Hondt* Coen De Roover*

*Vrije Universiteit Brussel, Belgium
† Johannes Kepler University Linz, Austria
*firstname.lastname@vub.ac.be † stefan.marr@jku.at

Abstract

Trace-based JIT compilers identify frequently executed program paths at run-time and subsequently record, compile and optimize their execution. In order to improve the performance of the generated machine instructions, JIT compilers heavily rely on dynamic analysis of the code. Existing work treats the components of a JIT compiler as a monolithic whole, tied to particular execution semantics. We propose a formal framework that facilitates the design and implementation of a tracing JIT compiler and its accompanying dynamic analyses by decoupling the tracing, optimization, and interpretation processes. This results in a framework that is more configurable and extensible than existing formal tracing models. We formalize the tracer and interpreter as two abstract state machines that communicate through a minimal, well-defined interface. Developing a tracing JIT compiler becomes possible for arbitrary interpreters that implement this interface. The abstract machines also provide the necessary hooks to plug in custom analyses and optimizations.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: semantics; D.3.4 [*Processors*]: compilers, optimization

Keywords tracing JIT compilation, operational semantics, dynamic analysis

1. Introduction

Just-in-time (JIT) compilation is a technique where, instead of statically compiling and optimizing an entire program upfront, an execution engine observes the program's execution and a JIT compiler emits machine code at run-time. Doing so allows the compiler to take into account specific characteristics of the program's execution when generating machine instructions, such as the values or types of the expressions that are executed. Dynamic analysis is therefore essential to the process of JIT compilation, since JIT compilers use these kinds of analyses to optimize the instructions that they generate at run-time [6].

The few formal models that exist on tracing compilation [2, 5] are irreversibly tied to one particular execution model for one particular programming language and treat the different components of a tracing JIT compiler – interpreter, tracer, compilers, and optimizers – as a monolithic whole with strong coupling between them. Investigating different execution models requires extensive changes to the language semantics used by these models. They are geared more toward exploring soundness of trace optimizations instead of enabling experiments in dynamic analysis.

We propose a formal framework that facilitates the design and implementation of a tracing JIT compiler and the dynamic analyses on which it relies by decoupling the tracing, optimization and interpretation processes, resulting in a complete framework that is more configurable and extensible than existing formal tracing models. The main benefit of our model is that it enables applying tracing compilation to, and subsequent dynamic analysis of, any arbitrary interpreter that satisfies a small, fixed interface (Section 3.2). Except for this minimal interface, the interpreter is otherwise treated as a black box and no particular implementation is specified. Our model also provides the necessary hooks to plug in custom analyses and optimizations. We do not define any concrete trace optimizations, but because of the strong decoupling of components, trace optimizations can be added in a modular way without being tied to any particular tracer or interpreter.

2. Trace-based JIT Compilation

Trace-based JIT compilation is a variant of JIT compilation that builds on two basic assumptions: most of the execution 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 [1]. Starting from these two premises, tracing

```
(let ((a 5))
  (while (not (zero? a))
    (complex-function (set! a (sub1 a)))))
```

```
...
push_continuation(appk(sub1))
lookup_variable(a)
pop_continuation()
apply_native(sub1)
pop_continuation()
...
guard_while()
...
```

Figure 1. Program with a traceable loop, and part of the corresponding trace.

compilers do not limit themselves to the compilation of methods, like method-based JIT compilers, but they trace frequently executed, "hot" loops in general.

Trace-based JIT compilation is usually performed in a mixed-mode execution environment [1], consisting of both an interpreter and a JIT compiler. In a first phase, the interpreter executes the program but simultaneously profiles the code, in order to identify hot loops. When a hot loop is detected, the interpreter starts *tracing* the execution of this loop: the operations that are performed by the interpreter during the execution of this loop are recorded into a trace. Tracing continues until the interpreter has completed one full iteration of the loop. Because the trace is a recording of the operations performed by the interpreter, function calls are automatically inlined in the trace. Once tracing has completed, the recorded trace is compiled and optimized. Subsequent iterations of this loop then execute the compiled trace instead of the original loop.

Because a trace is a representation of a single execution path, we must ensure that the conditions that caused the interpreter to select this path during the *recording* of the trace are still valid during the *execution* of the trace. Tracing JIT compilers check assumptions by adding *guards* to a 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 process of aborting trace execution and restarting interpretation is called a *side-exit*. Side-exits give rise to a runtime performance penalty, because the interpreter state must be correctly restored before normal interpretation can resume.

Example Figure 1 depicts a LISP-like program containing a loop that may be traced, followed by a part of the trace that would be recorded when executing this program. At some point during program execution, the interpreter might decide that the while loop is hot. The interpreter then traces one full iteration of the loop, starting from the beginning of the loop, continuing through the assignment, and terminating the trace when the start of the loop is reached again. Tracing the interpreter's actions comes down to recording the consecutive instructions the machine executes to update the state on which it operates. The example trace contains the guard $guard\_while()$ that checks whether the value computed for the condition of the while loop during trace execution corresponds with the boolean `#t` observed during trace recording.

$$
\begin{aligned}
TracerState &= \mathbf{ts}(ExecutionPhase, \\
&\qquad TracerContext, \\
&\qquad ProgramState, \\
&\qquad TraceNode) \\
ExecutionPhase &= NI \\
&\mid TR \\
&\mid TO \\
&\mid TE \\
tc \in TracerContext &= \mathbf{tc}(False + TraceNode,\ TraceNode*) \\
tn \in TraceNode &= \mathbf{tn}(Label, Trace) \\
\tau \in Trace &= TraceInstruction* \\
TraceInstruction &= ProgramState \to InstructionReturn \\
InstructionReturn &= \mathbf{traceStep}(ProgramState) \\
&\mid \mathbf{guardFailed}(Restartpoint) \\
TracingSignal &= \mathbf{loop}(Label) \\
&\mid False \\
restart &= Restartpoint \times ProgramState \to ProgramState \\
InterpreterReturn &= \mathbf{step}(ProgramState, \\
&\qquad Trace, \\
&\qquad TracingSignal)
\end{aligned}
$$

Figure 2. The tracing machine.

3. Tracing Machine

The tracing machine is modeled as a state machine transitioning between tracer states, as formalized in Figure 2. We give an overview of its different components.

3.1 Tracer State

We capture the state of the tracing machine in a $TracerState$, consisting of an execution phase, a tracer context, a program state, and a trace node.

During the execution of the program, the tracing machine switches between four distinct execution phases, indicated by $ExecutionPhase$: normal interpretation (NI), trace recording (TR), trace optimization (TO), and trace execution (TE). The execution phases and possible transitions can be modeled as a state diagram, as shown in Figure 3. We describe the transition between the different tracer states and execution phases of our tracing machine in Section 3.3.

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 $False$, if no trace is being recorded, or it consists of a trace node ($TraceNode$), which

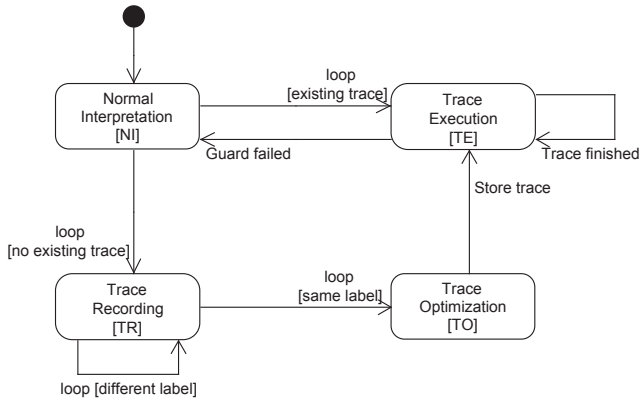

Figure 3. The four execution phases of a program.

$$\texttt{step} : \mathit{ProgramState} \mapsto \mathit{InstructionReturn}$$
$$\texttt{restart} : \mathit{Restartpoint} \times \mathit{ProgramState} \mapsto \mathit{ProgramState}$$

Figure 4. Minimal tracing interface for interpreters.

is a simple structure used to associate a trace with a unique label, so that this trace can later be retrieved by referencing its label. The second component of the tracer context is a list of all trace nodes containing the traces that have previously been recorded.

The $\mathit{ProgramState}$ is defined by the interpreter and is opaque to the tracing machine. The interpreter operates directly on program states, while 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 False if no trace is currently being executed, or it contains the trace node storing the trace that is being executed.

3.2 Tracing Interface

In our framework, the tracer monitors and controls the execution of the interpreter by using the minimal interface depicted in Figure 4.

It is assumed that the interpreter can be modeled as a state machine operating on a $\mathit{ProgramState}$. Interpreting a program then comes down to following a fixed set of state transition rules; tracing the interpreter can be modeled as recording the transitions that are applied by the interpreter and executing a trace is done by replaying all recorded state transitions on the current program state.

To allow for a more fine-grained optimization of traces, we allow the interpreter to use two sets of state transition rules: high-level and low-level transitions, both operating on a program state. One high-level transition is composed of several low-level transitions, i.e., executing the high-level

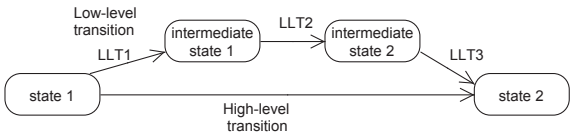

Figure 5. High-level and low-level state transitions.

transition is equivalent to applying each of the constituent low-level transitions consecutively.

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 $\mathit{InterpreterReturn}$: a three-tuple containing the resulting program state, the set of low-level state transitions, or *trace instructions*, that together constitute the high-level transition that has been performed, and possibly a *tracing signal*. During the tracing phase of the program's execution, the tracer appends the trace instructions to its current trace. The tracing signal is used by the interpreter to indicate that it has reached the start of a loop. This allows 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 machine should uniquely identify each loop in the user program through a label.

Traces contain guard instructions at certain locations to ensure that the control flow of the executed trace remains valid. In this model, we implement guards as a kind of trace instruction: a guard takes a program state as input, checks some condition in the state and signals back to the tracer that the condition is either still valid or has become invalid, i.e., that the guard has failed. Since the generation and placement of guards depends on the semantics of the language under consideration, the interpreter is expected to create the necessary guards when applying `step` and return them through the $\mathit{InterpreterReturn}$, mixed with the other trace instructions that are returned, so that, if any trace is being recorded, the guards are automatically inserted into the trace.

As we implement guards as a form of low-level transition, guards must have the same interface as "regular" trace instructions. Each trace instruction takes a program state as input and returns an $\mathit{InstructionReturn}$. This $\mathit{InstructionReturn}$ can either be a **traceStep** or a **guardFailed**. By using these two structures, we allow for both the execution of guard and non-guard instructions. Executing a non-guard instruction results in a **traceStep** being returned, carrying the new program state. A guard instruction, however, may return either of both structures. If the condition that is guarded is invalid, a **guardFailed** can be returned so that the tracer can detect this and take actions accordingly. If the condition is still valid, the guard can sim-

ply return its input, or any other program state it wishes to return.

We also require the interpreter to implement a mechanism to restart normal interpretation from the point of a guard failure. To this end, we require the existence of a function `restart` in the interface and we define the concept of *restartpoints*. The exact definition of a restartpoint is intentionally left vague, so that interpreters may implement these as they wish. When a guard fails during the execution of a trace, the tracing machine applies `restart` to the current program state, i.e., the state of the program at the point of the guard failure, and the restartpoint associated with the failed guard in order to retrieve the program state from where normal interpretation must resume.

It is important to note that we never specify a concrete definition for any of the concepts that we have defined here, such as the program state or the restartpoint. Interpreters may implement these as they wish, allowing for maximal decoupling between the tracer and the interpreter. To give a concrete example however, when using a CESK machine as an interpreter [3], the program state would be the CESK state while a restartpoint may simply be the control component of this state. In this case, the restart function could then be implemented as a function that takes this control component and merges it with the rest of the CESK state, i.e., the environment, store and continuation stack.

It is also important to note that the definition of the $TracingSignal$ can be extended. Later on, one may wish to extend this framework with a set of new features which might require the interpreter to send back additional information about its execution. This can then be accomplished by adding a new set of signals to $TracingSignal$.

3.3 Transition Rules

Figure 6 depicts the transition rules between tracer states. For simplicity, we omit the trace optimization phase that occurs after the recording of a trace is finished. Instead, we fold this phase into a single, abstract `optimize` function which takes a trace as input and returns an optimized version of this trace.

Normal Interpretation The normal interpretation phase (NI) of a program's execution refers to the execution stage in which no trace is being recorded or executed. The tracing machine therefore delegates execution entirely to the interpreter and only intervenes when the interpreter has reached the start of a loop, at which point the tracer may either decide to start tracing this loop, or decide to start executing a trace that was previously recorded for this loop. The interpreter signals these kinds of events through $TracingSignal$. The formal semantics describing the execution of this phase are given in Figure 6a.

Rule (1) represents the most common case where the interpreter machine has not entered any loop. It therefore returns $False$ instead of a signal, along with the new program

$$\mathbf{ts}(NI,\ tc,\ \varsigma,\ False) \rightarrow \quad (1)$$
$$\quad \mathbf{ts}(NI,\ tc,\ \varsigma',\ False)$$
$$\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{step}(\varsigma',\ \tau,\ False)$$

$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs),\ \varsigma,\ False) \rightarrow \quad (2)$$
$$\quad \mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(lbl,\ \tau),\ TNs),\ \varsigma',\ False)$$
$$\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{step}(\varsigma',\ \tau,\ \mathbf{loop}(lbl))$$
$$\quad \text{and if no trace for } lbl \text{ has been recorded yet}$$
$$\quad \text{and where } TNs \text{ is a list of trace-nodes}$$

$$\mathbf{ts}(NI,\ tc,\ \varsigma,\ False) \rightarrow \quad (3)$$
$$\quad \mathbf{ts}(TE,\ tc,\ \varsigma',\ \mathbf{tn}(lbl,\ \tau))$$
$$\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{step}(\varsigma',\ \tau',\ \mathbf{loop}(lbl))$$
$$\quad \text{and where } \tau \text{ is the trace that has previously been recorded for } lbl$$

(a) Normal interpretation

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(lbl,\ \tau),\ TNs),\ \varsigma,\ False) \rightarrow \quad (4)$$
$$\quad \mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(lbl,\ \tau : \iota_1 : ... : \iota_n),\ TNs),\ \varsigma',\ False)$$
$$\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{step}(\varsigma',\ \iota_1 : ... : \iota_n,\ False)$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(lbl,\ \tau),\ TNs),\ \varsigma,\ False) \rightarrow \quad (5)$$
$$\quad \mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(lbl,\ \tau : \iota_1 : ... : \iota_n),\ TNs),\ \varsigma',\ False)$$
$$\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{step}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{loop}(lbl'))$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(lbl,\ \tau),\ TNs),\ \varsigma,\ False) \rightarrow \quad (6)$$
$$\quad \mathbf{ts}(TE,\ \mathbf{tc}(False,\ tn : TNs),\ \varsigma',\ tn)$$
$$\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{step}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{loop}(lbl))$$
$$\quad \text{and where } \tau' \text{ equals } \tau : \iota_1 : ... : \iota_n$$
$$\quad \text{and where } tn \text{ equals } \mathbf{tn}(lbl,\ \mathtt{optimize}(\tau'))$$

(b) Trace recording

$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(lbl,\ \iota : \tau)) \rightarrow \quad (7)$$
$$\quad \mathbf{ts}(TE,\ tc,\ \varsigma',\ \mathbf{tn}(lbl,\ \tau))$$
$$\quad \text{if } \iota(\varsigma) = \mathbf{traceStep}(\varsigma')$$

$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(lbl,\ \iota : \tau)) \rightarrow \quad (8)$$
$$\quad \mathbf{ts}(NI,\ tc,\ \varsigma',\ False)$$
$$\quad \text{if } \iota(\varsigma) = \mathbf{guardFailed}(rp)$$
$$\quad \text{and where } \varsigma' = \mathtt{restart}(rp,\ \varsigma)$$

$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(lbl,\ `())) \rightarrow \quad (9)$$
$$\quad \mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(lbl,\ \tau))$$
$$\quad \text{and where } \tau \text{ is the trace that has already been recorded for } lbl$$

(c) Trace execution

Figure 6. Transition rules between tracer-states.

state and the set of actions it has applied to compute this new state. Because the tracing machine is running in the normal interpretation phase, it has no use for these instructions and therefore immediately discards them. The new tracer state is then just a copy of the old one, where the original program state is replaced by the new program state returned by the interpreter machine.

A more interesting case arises in rules (2) and (3), when the interpreter machine enters a loop identified by the label lbl. In rule (2), no trace has been recorded yet for lbl, so the tracer starts tracing this loop. It switches its execution phase to indicate that it is now tracing and updates its tracer context by replacing the component representing its current trace. This component now becomes a trace node consisting of the label that is traced, as well as the instructions τ that have just been executed by the interpreter and that were carried back

in the **step**. The program state of the tracer state must also be updated because this program state continues to be used by the interpreter as its actions are being traced.

In rule (3), the same conditions as in the second rule apply, except that the tracer context now does contain an already recorded trace for the label lbl. In this case, the tracer must start executing this trace, so it switches its execution phase to TE and switches the trace node of the tracer state to the trace node containing the previously recorded trace for the label lbl. We again also have to update the program state because this state now serves as the input to all state transitions that have been recorded in the trace τ.

Trace Recording In the trace recording phase (TR), all actions that are executed by the interpreter are recorded into a trace. Recording stops when the interpreter again enters the same loop that is currently being traced. The tracing machine can detect that it has entered the same loop by comparing the label of this loop to the label of the trace currently being recorded. Figure 6b gives the formal semantics governing the program's execution during the trace recording phase.

Similar to rule (1), rule (4) describes the common case where the interpreter machine has not entered a loop. The tracing machine therefore records the actions that have just been executed by the interpreter: it appends the list of trace instructions $\iota_1 : ... : \iota_n$ that were returned by the interpreter through the interface and appends them to the back of the trace τ that has already been recorded so far. Furthermore, as in the normal interpretation phase, we update the program state with the state returned by the interpreter machine.

In rule (5), the interpreter reaches the start of a loop carrying a label different from the label of the loop currently being traced. Because the label is different, entering this loop has no impact on the tracing process, so the tracer continues tracing. As with the first rule, we do have to update both the program state and the trace currently being recorded.

In rule (6), the interpreter also reaches the start of a loop, but this loop does have the same label as the one currently being traced. Reaching the start of this loop implies that we have now completed one full iteration of the loop, so we can stop tracing, analyse and optimize the completed trace and store the optimized trace away in the tracer context. Note that there is no explicit transition from the trace recording to the trace optimization phase, but that this is handled implicitly via the `optimize` function. Furthermore, because we are at the start of the loop that we have just traced, we can immediately start executing the optimized trace instead of switching back to normal interpretation.

Trace Optimization ***(TO)*** Our framework leaves open which analyses and optimizations are performed or how they are implemented. Instead, they are treated as one opaque function, `optimize`, which takes a trace as input and returns an optimized version of this trace. `optimize` essentially forms the hook through which developers can plug their optimizations and analyses into the framework. Although in practice the analysis and optimization of traces is likely to be performed in the background, simultaneously to program execution, for simplicity we treat trace optimization as a sequential process in the model.

Trace Execution In the trace execution phase (TE) the tracer is executing a previously recorded trace. In Figure 6c we define the formal semantics that express how the execution of a trace should be handled. For these rules, we write $\iota(\varsigma)$ to express that we apply the trace instruction ι on the program state ς. Recall that a guard instruction is considered to be the same as any other trace instruction. In rule (7), we apply an instruction ι from the trace on the current program state and a **traceStep** is returned, containing the program state resulting from applying this instruction. The tracer then continues by swapping its program state and moving on to the next instruction in the trace. This rule represents both the case where we apply a non-guard instruction, or a guard instruction that did *not* fail.

Rule (8) expresses the case where a guard instruction has been applied and subsequently failed. The rule states that we should then switch our execution phase to normal interpretation. Additionally, interpretation should be restarted from the point in the program that corresponds with the guard failure. To find this point, we can call the restart function provided by the interpreter with the restartpoint given by the guard and the current program state, as mentioned in Section 3.2.

Rule (9) handles the case where we have reached the end of a trace. Reaching the end of the trace corresponds with finishing one full iteration of the loop, so we simply restart the trace: we look up the full trace belonging to the label of the trace we we executing and we replace the current, empty, trace by this new, full, trace.

4. Evaluation

As a validation of this model, we have implemented an interpreter for a LISP-like language. Figure 1 depicts an example of a small program in this language. Its interpreter satisfies the interface defined in Section 3.2. The implementation[1] and complete formal semantics[2] are publicly available.

The interpreter is modeled after a CESK machine [3], extended with a value register named v for storing the value of the last expression that was evaluated. Execution of a program therefore proceeds through following a set of CESK state transitions.

Due to space constraints, we provide only a partial overview of the interpreter, containing the most relevant evaluation rules. For example, evaluation of a while-expression of the form (`while` $cond$ exp) is handled by the following transition:

$$\mathbf{ps}((\texttt{while}\ cond\ exp),\ \rho,\ \sigma,\ \kappa,\ v) \rightarrow$$

[1] `https://github.com/mvdcamme/woda15/`

[2] `https://soft.vub.ac.be/~mvdcamme`

$$\mathbf{step}(\mathbf{ps}(cond,\ \rho,\ \sigma,\ \phi : \kappa,\ v),\ \{psh\_cont(\phi)\},\ False)$$
where ϕ equals $\mathbf{wcondk}(cond,\ exp)$

Evaluating such an expression thus comes down to pushing a **wcondk** continuation, containing the condition and the body of the while-expression, onto the continuation stack and continuing with the evaluation of the condition. Calling `step` on the left-hand side program state therefore results in the **step** structure on the right-hand side of the transition.

Once evaluation of $cond$ is completed, the **wcondk** continuation is popped and the following rule is triggered:

$$\begin{array}{l}\mathbf{ps}(\mathbf{wcondk}(cond,\ exp),\ \rho,\ \sigma,\ \phi : \kappa,\ v) \rightarrow \\ \quad \text{if } v \text{ equals } \#\mathtt{t}: \\ \quad \mathbf{step}(\mathbf{ps}(exp,\ \rho,\ \sigma,\ \mathbf{wbodyk}(cond,\ exp) : \phi : \kappa,\ v), \\ \qquad \{guard\_while();\ psh\_cont(\mathbf{wbodyk}(cond,\ exp))\} \\ \qquad \mathbf{loop}(exp)) \\ \quad \text{if } v \text{ equals } \#\mathtt{f}: \\ \quad \mathbf{step}(\mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ v),\ \{pop\_cont()\},\ False)\end{array}$$

If the condition was true, evaluation proceeds to the body of the while. Since the interpreter then enters the start of a loop, it sends the tracing signal **loop** to the tracer. In the case of this mini-language, the body of the while loop can be used to uniquely identify each loop. If the condition was false, we skip evaluation of the loop and pop the topmost continuation from the stack.

In each transition rule, the interpreter also returns the consecutive trace instructions that were used in the computation of the new high-level program state. For example, pushing a continuation ϕ onto the continuation stack is referred to as $psh\_cont(\phi)$ and defined as:

$$\begin{array}{l}\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v) \xrightarrow{\texttt{psh\_cont}(\phi)} \\ \quad \mathbf{traceStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v))\end{array}$$

In this mini-language, only one type of guard instruction is needed, for checking whether a while-condition is still valid. This guard is implemented as follows:

$$\begin{array}{l}\mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v) \xrightarrow{\texttt{guard\_while()}} \\ \quad \mathbf{guardFailed}(\phi) \text{ if } v \text{ equals } \#\mathtt{f} \\ \quad \mathbf{traceStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v)) \text{ if } v \text{ equals } \#\mathtt{t}\end{array}$$

This guard checks whether the while-condition, whose evaluated value is stored in the value register v, still equals $\#\mathtt{t}$. If it does, the guard just returns the input program state. Else, the guard returns a **guardFailed** containing the current continuation that rests at the top of the continuation stack. Finally, the `restart` function can be implemented as follows:

$$\mathtt{restart}(\phi,\ \mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v)) \rightarrow \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ v)$$

This function effectively generates a new program state by popping the first continuation from the continuation stack and continuing evaluation through this continuation.

5. Conclusion

We presented a formal model of trace-based JIT compilation that distinguishes itself from previous work by its focus on separating the tracing aspect from the interpretation component in the execution of a program. We achieve this by dividing a program's execution between two separate entities: a tracing machine and an interpreter machine. This decoupling was enabled by identifying the set of requirements that must be satisfied by an interpreter in order to enable tracing of its execution, and moulding these requirements into an *interface* (Section 3.2). Our model formally defines a tracing machine which interacts with the interpreter through this interface, but otherwise treats it as a black box. The resulting framework allows us to model the tracing of all interpreters that adhere to this interface. We validated our model by implementing a CESK-style interpreter that conforms to the defined interface, and demonstrated how it interacts with the tracer.

The model presented in this paper facilitates experiments with dynamic analyses because it formalizes the concept of trace-based JIT compilation and enables gathering a program's execution traces. These traces can subsequently serve as input for run-time dynamic analyses, such as type specialization, constant propagation of observed runtime values, allocation removal etc. In future work we plan to develop new analyses, or evaluate existing optimizations, in the context of a diverse set of execution traces. Alternatively, this model can be used to prove the soundness of existing trace optimizations, as has already been explored in previous formal models [2, 5].

References

[1] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proc. of the 4th Workshop of ICOOOLPS*, '09, pages 18–25.

[2] S. Dissegna, F. Logozzo, and F. Ranzato. Tracing compilation by abstract interpretation. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium of POPL*, '14, pages 47–59, 2014.

[3] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proc. of the 14th ACM SIGACT-SIGPLAN Symposium of POPL*, '87, pages 314–, 1987.

[4] M. Fulton and M. Stoodley. Compilation techniques for real-time java programs. In *Proc. of CGO*, pages 221–231, 2007.

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

[6] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *ACM SIGPLAN Notices*, volume 36, pages 180–195. ACM, 2001.