Vrije Universiteit Brussel

Faculty of Sciences and Bio-engineering Sciences
Departement of Computer Science
Software Languages Lab

The Essence of Meta-Tracing JIT Compilers

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Maarten Vandercammen

Promotors: Prof. Dr. Coen De Roover
Prof. Dr. Theo D'Hondt

Advisors: Dr. Joeri De Koster
Dr. Stefan Marr
Jens Nicolay

JUNE 2015

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen
Software Languages Lab

The Essence of Meta-Tracing JIT Compilers

Proefschrift ingediend met het oog op het behalen van de graad van
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Maarten Vandercammen

Promotors: Prof. Dr. Coen De Roover
Prof. Dr. Theo D'Hondt

Begeleiders: Dr. Joeri De Koster
Dr. Stefan Marr
Jens Nicolay

JUNI 2015

Abstract

JIT compilation is a successful strategy for the execution of dynamic programming languages, as it mitigates the performance penalties inherent to these languages. Tracing JIT compilers are an alternative to the more common method-based JIT compilers. In tracing JIT compilation, the VM identifies frequently executed, 'hot' program paths at runtime. Once a hot path is detected, the VM records all instructions in this path into a trace until execution loops back to the first instruction of this path. The trace is then compiled and subsequent iterations of this path execute the compiled trace instead of the original code.

However, despite the advantage tracing JIT compilers offer, the extra engineering effort required to develop them is often seen to not be worth the effort. The technique of *meta-tracing* offers a solution for this dilemma. Language interpreters are built on top of a common tracing JIT compiler and this compiler traces the execution of the interpreter *while it is running a user-program*. This allows language implementers to reach acceptable performance levels with only a minimum of effort required.

Until now, little attention has been paid to the formal foundations of (meta-) tracing compilation. In this thesis, we build a meta-tracing compiler from the ground up and establish a set of formal semantics describing the execution of this compiler.

We extend our framework with a number of state-of-the-art features commonly found in other (meta-) tracing compilers. We introduce a loop hotness detection feature, where a loop is only traced after a heuristic has determined it to be sufficiently hot, and guard tracing, where we try to decrease the runtime costs inherent to aborting the execution of a trace. We also investigate the concept of trace merging. By joining traces when their underlying control-flow merges, we can avoid having to trace all possible paths through a program. This improves space efficiency and averts some of the runtime overhead associated with tracing. Trace merging has not been widely deployed yet in existing tracing compilers.

Samenvatting

JIT compilatie is een successvolle manier om dynamische programmeertalen uit te voeren, aangezien het de negatieve impact op runtime performance, die eigen is aan deze talen, verzacht. Tracing JIT compilers zijn een alternatief voor de vaker gebruikte method-based JIT compilers. In tracing JIT compilatie identificeert de VM vaak uitgevoerde, 'hot', control-flow paden at runtime. Eens deze gevonden zijn, registreert de VM alle instructies die deel uitmaken van dit pad in een trace totdat executie terugspringt naar de eerste executie van dit pad. De trace wordt dan gecompileerd en volgende iteraties van het pad voeren de geoptimiseerde trace uit in plaats van de originele code.

Ondanks het voordeel die tracing JIT compilers bieden, wordt de extra complexiteit die nodig is om deze compilers te ontwikkelen vaak beschouwd als een zware hindernis. De techniek van *meta-tracing* biedt een oplossing voor dit dilemma. Language interpreters worden ontwikkeld bovenop een gemeenschappelijke tracing JIT compiler. Door deze compiler de executie van de interpreter te laten tracen, *terwijl die een user-programma uitvoert*, kunnen language implementers acceptabele performantie niveaus bereiken met een minimum aan overlast.

Tot op heden werd er weining aandacht besteed aan de formele basis van (meta-)tracing compilatie. In deze thesis ontwikkelen we een meta-tracing compiler vanaf nul en specifiëren we een set formele semantieken die de executie van deze compiler beschrijven.

We breiden ons framework uit met een aantal state-of-the-art features die vaak deel uitmaken van andere (meta-)tracing compilers, zoals loop hotness detection, waarbij een loop enkel getraced wordt als een heuristiek bepaald heeft dat deze loop voldoende 'hot' is, en guard tracing, waarbij we de negatieve performance hit die wordt opgelopen bij een trace side-exit proberen te verlagen. We onderzoeken ook het concept van trace merging. Door traces samen te voegen wanneer hun onderliggende control-flow samenvloeit, kunnen we vermijden om alle mogelijke paden doorheen de control-flow van een programma te moeten tracen. Dit verbetert de space-efficiency en vermijdt een deel van de runtime overhead die eigen is aan tracing. Trace merging werd nog niet vaak gebruikt in huidige tracing compilers.

Acknowledgements

This thesis would not have been possible without the assistance of my advisors, Joeri De Koster, Stefan Marr, Jens Nicolay, and my promotors, Theo D'Hondt and Coen De Roover. They sacrificed many hours discussing various topics, providing invaluable insights, proofreading this dissertation, helping me solve problems and errors I encountered, and generally guiding me in the creation of this thesis. Throughout the year, I could always count on them to help me along when I became stuck with some problem, no matter how trivial the issue may have been. I want to thank all of them for their patience, compassion, kindness and motivation. Sharing their wisdom and knowledge aided me in understanding the various topics relevant to this thesis.

I would also like to thank all of my friends, as well as Leen, Raf and Lander, for keeping me sane throughout the last year. The many laughs and drinks we shared, their encouragements when I was lacking motivation, and their general acts of friendship were invaluable to me.

The administration of the Software Languages Lab also deserves my gratitude, for providing the thesis students with several facilities such as the student room

Lastly, I want to express my gratitude to my parents and my brother and sister. Although their help may have been the easiest to overlook, it was by no means the least important. I could not have gotten where I am today without their help. Thank you!

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem statement and goal . . . 2
- 1.3 Contributions . . . 3
- 1.4 Roadmap . . . 4

2 Background **6**
- 2.1 Just-in-time compilation . . . 6
- 2.2 Trace-based compilation . . . 7
  - 2.2.1 Overview . . . 7
  - 2.2.2 Tracing . . . 7
  - 2.2.3 Guard instructions . . . 10
- 2.3 Meta-tracing . . . 11
  - 2.3.1 Overview . . . 11
  - 2.3.2 Example . . . 12
  - 2.3.3 Matching traces with user loops . . . 13
  - 2.3.4 Interpreter hints . . . 14

3 Related work **16**
- 3.1 Overview . . . 16
  - 3.1.1 Conception . . . 16
  - 3.1.2 HotpathVM . . . 16
  - 3.1.3 TraceMonkey . . . 17
  - 3.1.4 Other tracing compilers . . . 18
- 3.2 Formal frameworks . . . 18
- 3.3 Trace selection strategies . . . 19
- 3.4 Meta-tracing . . . 21
  - 3.4.1 The RPython project . . . 21
  - 3.4.2 Hierarchical VM layering . . . 26

4 The SLIPT language **28**
- 4.1 CESK-machines . . . 28
  - 4.1.1 Overview . . . 28
  - 4.1.2 Example . . . 29

4.2 Syntax . . . 30
4.3 CESKθ-machine . . . 32
4.3.1 Overview . . . 32
4.3.2 CESKθ definition . . . 32
4.3.3 Evaluation rules . . . 34
4.4 Low-level instruction set . . . 40
4.5 Redefining SLIPT's semantics . . . 42
4.6 Implementation . . . 45

5 Tracing semantics **47**
5.1 Introduction . . . 47
5.2 Tracing overview . . . 48
5.3 Extended Syntax . . . 54
5.4 Tracing machine . . . 54
5.5 Interface . . . 56
5.6 Guard instructions . . . 58
5.6.1 Introduction . . . 58
5.6.2 Guard examples . . . 58
5.6.3 Low-level instruction interface . . . 60
5.6.4 Guard instructions . . . 62
5.6.5 Adding guards to existing transition rules . . . 64
5.7 Normal interpretation . . . 67
5.8 Trace recording . . . 69
5.9 Trace execution . . . 70

6 Extending the tracing framework **73**
6.1 Hot loop detection . . . 73
6.1.1 Overview . . . 73
6.1.2 Extending the tracing machine . . . 74
6.1.3 Semantics . . . 75
6.2 Guard tracing . . . 76
6.2.1 Overview . . . 76
6.2.2 Extending the tracing machine . . . 77
6.2.3 Guard instructions . . . 78
6.2.4 Semantics . . . 78
6.3 Trace merging . . . 81
6.3.1 Overview . . . 81
6.3.2 Syntax . . . 85
6.3.3 Extending the tracing machine . . . 86
6.3.4 Interface . . . 87
6.3.5 Handling merging annotations . . . 87
6.3.6 Low-level instructions interface . . . 89
6.3.7 Semantics . . . 90
6.4 Validation conclusion . . . 92

7 Conclusion **93**
7.1 Summary 93
7.2 Contributions 94
7.3 Future work 95
7.3.1 Optimizations 95
7.3.2 Direct versus meta-tracing 96
7.3.3 Additional features 96
7.4 Overall conclusion 96

List of Figures

2.1 The phases used by a tracing JIT compiler 8
2.2 The logical control-flow of the Python program from Listing 2.1 10
2.3 The towers of interpreters when meta-tracing 11

3.1 Translating a language interpreter to C 22
3.2 The *promote* and *elidable* annotations in RPython 26

4.1 A set of transition rules for a CESK-machine 29
4.2 A concrete example of evaluating an assignment 30
4.3 The syntax of SLIPT 31
4.4 The CESKθ-machine 33
4.5 Handling variables and values 34
4.6 Evaluating definitions and assignments 35
4.7 Evaluating if-expressions 36
4.8 Evaluating sequences of expressions 37
4.9 Evaluating function applications 38
4.10 The definition of `bindParams` 39
4.11 Evaluating `apply`-expressions 39
4.12 The low-level instruction set 42
4.13 The LLI CESKθ-machine 45

5.1 The relation between the interpreter and the tracer 49
5.2 Transitioning between the three phases in a SLIPT program's execution 50
5.3 A SLIPT program that can be traced along with its resulting trace 53
5.4 The new definition for *Exp* 54
5.5 The tracing machine 55
5.6 The interface between the CESKθ-machine and the tracer 56
5.7 The interface between the low-level instructions and the tracer . 61
5.8 The definition of `guard-false` 62
5.9 The definition of `guard-true` 63
5.10 The definition of `guard-same-closure` 63
5.11 The definition of `guard-same-nr-of-args` 64
5.12 The new transition rules for evaluating an if-expression 65

5.13 The new transition rules for evaluating a function application . 66
5.14 The new transition rules for evaluating an `apply`-expression . . 66
5.15 Handling an **applyFailedk**$(rator,\ i)$ continuation 66
5.16 Normal interpretation if no annotation is encountered 67
5.17 Normal interpretation if a `can-close-loop` annotation is encountered 68
5.18 Normal interpretation if a `can-start-loop` annotation is encountered 68
5.19 Trace recording if no annotation is encountered 69
5.20 Trace recording if an annotation with a different label is encountered 69
5.21 Trace recording if an annotation with the label that is being traced is encountered 70
5.22 Trace execution if no guard fails 71
5.23 Trace execution if a guard fails 71
5.24 Trace execution if the end of a trace has been reached 71
5.25 Trace execution if the end of a looping trace has been reached . 72

6.1 The updated tracing machine for loop hotness detection 75
6.2 The normal interpretation rules which have not changed 75
6.3 The additional normal interpretation rules 76
6.4 The updated tracing machine for guard tracing 78
6.5 The updated definition of `guard-false` 78
6.6 The updated definition of $EventSignal$ 78
6.7 The updated trace recording rules 80
6.8 The trace execution rules which have not changed 80
6.9 The updated trace execution rules for handling guard failure . . 81
6.10 Trace execution if the end of a looping trace has been reached . 81
6.11 The logical control-flow of Listing 6.1 82
6.12 An example of how control-flow is translated into three traces . 83
6.13 An example of how trace merging solves the issue of trace explosion 84
6.14 The new definition of Exp 85
6.15 The updated tracing machine for trace merging 86
6.16 The new definition of $AnnotationSignal$ 87
6.17 A concrete example of how two traces are merged together . . . 89
6.18 The updated interface for low-level instructions 90
6.19 The implementation of the two new instructions 90
6.20 The normal interpretation rules for the two new annotations . . 91
6.21 The updated trace recording rules 91
6.22 The updated trace execution rules 92

Listings

2.1 Python code containing a traceable loop 9
2.2 The resulting pseudo trace for this loop 9
2.3 A small bytecode interpreter 13
2.4 The user-program 13
2.5 The bytecode for this user-program 13
2.6 The bytecode interpreter with a hint attached 14
3.1 Example of a false loop in Python 20
3.2 The annotations used in PyPy 22
5.1 A function in SLIPT 49
5.2 A looping function in SLIPT 51
5.3 A function whose trace can contain a side-exit 52
5.4 A function in SLIPT 53
5.5 A function where control-flow may diverge 58
5.6 Control-flow diverging because of higher-order functions 59
5.7 A more subtle example of code where guards must be introduced 60
6.1 A function where control-flow merges 82
6.2 A program causing trace explosion to arise 84
6.3 A program where trace merging may be detrimental 85
6.4 An example of how the new annotations are used 86
6.5 A nested if-expression 88
6.6 A SLIPT program with diverging control-flow 89

Chapter 1

Introduction

1.1 Context

In order to satisfy consumers' need for ever more powerful and responsive applications, researchers do not only turn their attention towards the development of increasingly sophisticated hardware, but also towards software models that optimize a program's execution, such as just-in-time (JIT) compilation. Instead of compiling an entire program upfront, a JIT compiler only compiles those parts of the code that are frequently executed. Since compilation proceeds at runtime, the compiler can exploit knowledge of the program's execution to apply optimizations that require precise information about the program's behavior, which is in general difficult to obtain for ahead-of-time compilers. JIT compilation has especially proven its merit in the context of languages that use dynamic programming features, such as late-binding, or dynamic programming languages in general. For these languages, it is difficult for static compilers to generate efficient code because they cannot know in advance which code will be called at runtime. JIT compilers on the other hand, can observe the invoked code at runtime and subsequently use this information to optimize the generated code.

In this thesis we present a trace-based JIT compiler. Trace-based JIT compilation is a technique where the compiler identifies frequently executed, hot program paths at runtime. Once a hot path is detected, the compiler records, or *traces*, each of the instructions on this path into a trace. Tracing continues across function calls and proceeds until execution jumps back to the first instruction that was recorded. The trace is then optimised and compiled. Subsequent iterations of this path execute the optimized trace instead of the original code. Because a trace is a representation of a single path through the control-flow, it consists only of a linear sequence of instructions, without any kind of control-flow at all. This makes traces suitable targets for optimization, because the absence of any control-flow, combined with the fact that a trace presents concrete information on a program's execution, increases the compiler's abil-

ity to optimize the code. However, when executing a trace, we have to ensure that the conditions that were responsible for following this path through the program are still valid when the trace is executed. For this reason, we insert guard instructions in the trace. A guard is responsible for checking a certain condition in the trace. If it finds that the condition is invalidated, execution of the trace must be aborted.

Meta-tracing is a generalization of regular tracing where we do not trace a user-program directly, but where instead we trace the execution of a language interpreter *while this interpreter is running a user-program*. Language developers who wish to use tracing compilation for their language must create only a regular interpreter, without having to write a dedicated tracing JIT compiler. By executing their interpreter using a meta-tracing compiler, language developers can reap the optimization benefits without incurring its costs in complexity. In order to maximize the effect of the trace-based optimizations, developers usually need to introduce hints for the meta-tracing compiler in their interpreter.

1.2 Problem statement and goal

Although tracing JIT compilers have existed since the early 2000s, little research has been performed towards understanding the formal foundations of tracing, and especially meta-tracing, compilation. Recently, there have been attempts to capture the concept of tracing compilation in formal semantics (Guo & Palsberg, 2011; Dissegna et al., 2014). However, both studies were geared towards a single purpose only, proving the soundness of optimizations that are applied on a trace, and could not be easily reused for other, more general, purposes, such as investigating the impact of certain features of tracing compilation on a program's execution.

Specifically, in the framework developed by Guo & Palsberg (2011) two near-identical sets of evaluation rules are necessary to describe a program's execution. One set is used to determine a program's execution during normal interpretation, i.e., when no tracing is going on whatsoever. The other set is used to describe a program's execution during the recording of a trace. This increases the complexity of the framework and makes it more difficult to extend the evaluation rules. The framework described by Dissegna et al. (2014) is also problematic: it does not model how traces are recorded concretely during the execution of a program, but rather assumes the existence of a set of possible pre-existing traces for each state of the program, based on a heuristic that is ”hard-coded” into their model.

Both frameworks prove more than adequate when used only for reasoning over the execution of a trace, but fall somewhat short in modelling all other aspects of a program's execution. Additionally, these models are both tightly integrated with one specific set of evaluation rules, for one specific programming language: their models cannot be adapted to simulate the execution of any arbitrary programming language. For these reasons, it is difficult to extend the tracing compilers described in their models with any new features. If one

wishes to formally reason about the impact of extending the tracing compiler with a particular feature, they could not use these frameworks. Lastly, these recent attempts were aimed entirely at understanding only direct tracing, and not meta-tracing.

We propose to create a meta-tracing JIT compiler and specify a set of formal semantics that express the workings of this compiler formally. This compiler is *minimalistic, yet functional*. It is capable of doing just the following:

- Modelling all aspects of a program's execution: the recording and execution of traces, as well as normal interpretation of the program without any tracing whatsoever
- Handling guard instructions and aborting the execution of traces when necessary
- Handling the hints used by language developers in their interpreters to enable meta-tracing of these interpreters

We believe that implementing these three features is adequate for developing a complete formal model of how real-world meta-tracing compilers behave. By specifying our model through a set of formal semantics, we can understand how tracing compilation in general works on a formal level. Because the proposed framework is minimalistic, we believe that the framework is easier to comprehend and extend with additional features.

In order to fulfil our goal of keeping our meta-tracing compiler minimalistic, we model our compiler as an entity that attaches itself to an existing interpreter and records its actions while this interpreter evaluates a program. Because our compiler is a meta-tracing compiler, the input to this interpreter are mainly *other interpreters*. However, because meta-tracing can be seen as a generalization of tracing in general, our compiler should also be able to function as a regular, non-meta-tracing tracing compiler. By specifying our compiler explicitly through its interaction with an interpreter, we can make our compiler configurable in the sense that it can trace the execution of any program, written in an arbitrary programming language, as long as the compiler is attached to a suitable interpreter, capable of executing this program.

1.3 Contributions

In this thesis, we develop a meta-tracing JIT compiler from scratch. To describe the entire development process in its totality, we specify a language, SLIPT, short for SLIP traced, and an interpreter for this language through a formal execution model. We then define a formal model for our meta-tracing compiler, which executes SLIPT programs by attaching itself to the specified interpreter and tracing its actions. Although the proposed compiler is capable of executing arbitrary programs when a compatible interpreter is provided, we restrict ourselves to the execution of SLIPT in this thesis.

Our compiler has the following characteristics:

- **Minimalistic**: because we build the compiler from the ground up, we carefully select which features are absolutely essential to the process of meta-tracing and which ones are not. The only features we include in our minimalistic meta-tracing compiler are: the modelling of the recording and execution of traces, as well as normal interpretation of a program, the handling of guard failures and the handling of hints included in the interpreter by language implementers. By removing all non-essential aspects of tracing compilation, we avoid adding unnecessary complexity that makes it harder to reason over the workings of the model.
- **Configurable**: in contrast with formal specifications of tracing compilers, the input language of the compiler does not have to be set in stone: interpretation of the input language is cleanly separated from the actual meta-tracing. This enables us to switch our SLIPT interpreter for any other interpreter, as long as it conforms to a specific interface, in contrast with previous work on formalizing tracing compilation.
- **Extensible**: because the compiler is minimalistic, it is easy to extend with additional features, heuristics or optimizations. This quality has not been achieved in previous attempts at formalizing tracing.
- **Executable**: we implement both the SLIPT interpreter and the tracing compiler in Racket. To the best of our knowledge, previous formal models on tracing compilation were not actually implemented but only existed as a formal specification.

1.4 Roadmap

This thesis is structured as follows:

Chapter 2: Background presents the concept of tracing and meta-tracing compilation and provides examples of traces that are produced both by direct and by meta-tracing JIT compilers. It also explains the, sometimes subtle, differences between both kinds of tracing compilers.

Chapter 3: Related work gives an overview of existing, commonly-used, state-of-the-art (meta-)tracing compilers and specifies which features, heuristics or optimizations are often found in these compilers. It also states which approaches have already been attempted towards formally understanding the concept of trace-based compilation.

Chapter 4: The SLIPT language specifies our input language, SLIPT, and presents operational semantics detailing how this language is executed. These formal semantics serve as the definition of our interpreter.

Chapter 5: Tracing semantics presents our tracing compiler and describes how tracing is performed in our framework by providing the formal semantics that express the workings of this compiler.

Chapter 6: Extending the tracing framework gives a validation for our framework by integrating a set of extensions to our compiler, indicating that our semantics are powerful enough to model non-trivial additions.

Chapter 7: Conclusion summarizes the contents of this thesis, states how the problem statement has been answered and gives some directions for possible future work.

Chapter 2

Background

2.1 Just-in-time compilation

Just-in-time (JIT) compilation is a technique where, instead of statically compiling and optimizing an entire program upfront, the execution engine observes a program's execution and a JIT compiler emits machine code at runtime. 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 certain variables that are used in the code. This compilation strategy is especially powerful when applied to languages that use dynamic programming features, such as late-binding. The presence of late-binding generally prohibits an ahead-of-time compiler from determining which code will be called at runtime, and therefore also hinders these compilers' ability to generate efficient code. Additionally, since the dynamic type of a variable is known during a program's execution, the JIT compiler can emit type-specialized code, leading to optimal performance of the generated code.

The concept of JIT compilation became popular through the development of a compiler for the programming language Self (Ungar & Smith, 1987; Smith & Ungar, 1995), although seminal work on JIT compilation had already been performed by Mitchell for the language LC^2 (Mitchell et al., 1967) and Abrams for his APL machine (Abrams, 1970; Aycock, 2003).

Most JIT compilers are method-based compilers. These compilers identify frequently executed, i.e., hot methods in a program at runtime and compile them to native code so that subsequent calls to this method trigger the execution of the compiled code, instead of the original method.

2.2 Trace-based compilation

2.2.1 Overview

Trace-based compilers are an alternative to method-based compilers. They are built on two basic assumptions (Bolz et al., 2009):

- Most of the execution time of a program is spent in loops
- Several iterations of the same loop are likely to take the same path through the program

Starting from these two premises, tracing compilers do not limit themselves to the compilation of methods, like method-based JIT compilers, but they trace hot program paths in general. Many tracing JIT compilers however focus only on compiling loops (Schneider & Bolz, 2012). For the remainder of this chapter, we focus on compiling only loops.

Trace-based JIT compilation is usually performed in a mixed-mode execution environment (Bolz, 2012), 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 frequently executed loops.

2.2.2 Tracing

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 a full iteration of the loop. Because the trace is a recording of the operations performed by the interpreter, functions are automatically inlined in the trace. Not all operations are recorded; function calls themselves and branching statements are not recorded directly into 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.

Figure 2.1 shows the various phases in a program's execution when using a tracing JIT compiler.

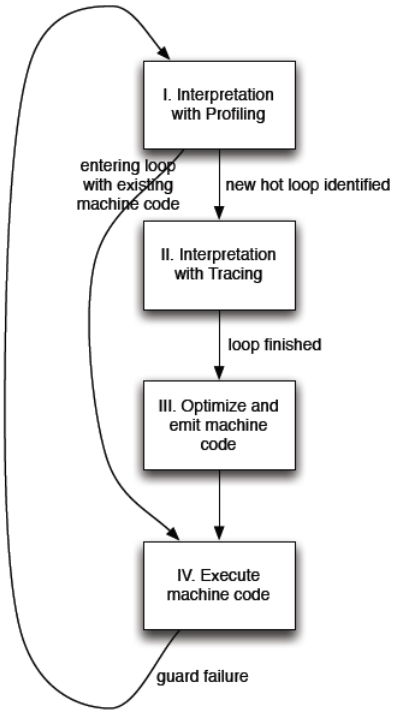


Figure 2.1: The phases used by a tracing JIT compiler, retrieved from Bolz (2012, Figure 2)

Since the trace consists of all operations that are performed by the interpreter as it executes one iteration of the loop, the recorded trace consists of a linear sequence of instructions representing the specific program path that was taken through the program while tracing. If control-flow splits at some point during the program's execution, e.g., because an if-statement is executed, then the trace contains only instructions for executing one of these branches.

In the following, we give a concrete example, adapted from (Bolz, 2012), of how tracing works. Consider the program shown in Listings 2.1. At some point in the program's execution, the interpreter might decide that the while-loop inside the function `strange_sum` is hot and that it should be traced. Tracing then starts from the beginning of this while-loop, continues through the call to `f` and terminates when the last statement in the body of the while has been executed. Since the function `f` contains an if-statement, control-flow splits in two branches there. Suppose that the if-condition evaluated to false, i.e., `b % 46` does not equal `41`, then the false-branch was selected and the resulting trace only contains instructions for executing this false-branch. Figure 2.2 shows the full logical control-flow graph of this example Python program. The path that was traced through this program is marked in blue.

Listings 2.2 shows a pseudo trace corresponding to this path. The current

values of the variables `result` and `n` from the original program are read in prior to the execution of the trace. The trace proper, which corresponds to the actual loop in the program, then follows, consisting of a sequence of bytecode instructions. At the end of the trace, execution jumps back to the beginning of the loop.

```
def f(a, b):
    if b % 46 == 41:
        return a - b
    else:
        return a + b

def strange_sum(n):
    result = 0
    while n >= 0:
        result = f(result, n)
        n -= 1
    return result
```

Listing 2.1: Python code containing a traceable loop, retrieved from Bolz (2012, Figure 3)

```
# corresponding trace:
    result0 = read(result)
    n0 = read(n)
loop:
# inside result = f(result, n)
    i0 = int_mod(n0, 46)
    i1 = int_eq(i0, 41)
    guard_false(i1)
    result1 = int_add(result0, n0)
    n1 = int_sub(n0, 1)
    i2 = int_ge(n1, 0)
    guard_true(i2)
    result0 = result1
    n0 = n1
    jump(loop)
```

Listing 2.2: The resulting pseudo trace for this loop, adapted from Bolz (2012, Figure 3)

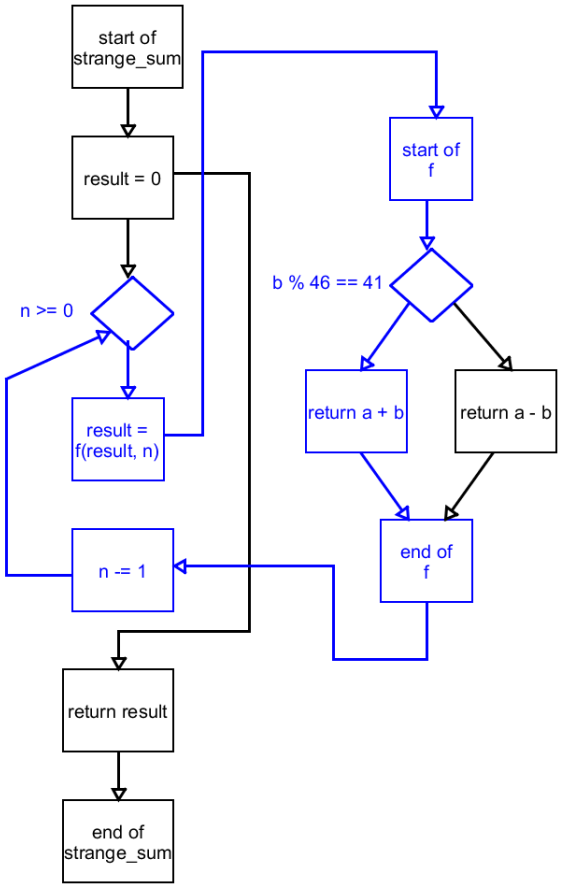


Figure 2.2: The logical control-flow of the Python program from Listing 2.1

2.2.3 Guard instructions

Because a trace is a representation of only one single 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. Concretely, for the previous example, during the recording of the trace the interpreter executed the false-branch of the if-expression inside the function `f` because the condition `b % 46 == 41` evaluated to `False`. This means that we must check during the execution of this trace whether this condition still evaluates to `False`. If it does not, execution of the trace must be aborted because an assumption on which part of the trace is resting was proven to be invalid.

This is implemented by adding *guards* to the trace. A guard checks a condition and if it finds that the condition does not hold, aborts execution of the trace. Guard instructions are inserted into the trace during its recording.

In the previous example, two guards were used. A `guard_false` was inserted to check whether the condition `b % 46 == 41` evaluates to `False` and a `guard_true` was placed to determine whether the condition `n >= 0` still evaluates to `True`. If the condition that is associated with a guard does not hold any more, we say that the guard *fails*. Execution of the trace is then aborted, and the interpreter resumes normal interpretation of the program from the point in the program corresponding to this guard failure. The process of aborting execution and restarting interpretation is called a *side-exit*. Side-exits are inherently costly to the runtime performance of the program's execution, because execution of the compiled and optimized trace must be abandoned and normal interpretation must be restarted.

Because the trace is a direct recording of the execution of a user-program, the style of trace-based compilation that we presented in this section is called *direct* tracing.

2.3 Meta-tracing

2.3.1 Overview

Whereas a direct tracing compiler records the execution of a user-program directly, *meta-tracing* refers to a configuration where a *tracing* interpreter executes another *language* interpreter while this language interpreter itself is executing a user-program. The tracing interpreter traces the execution of the language interpreter and the recorded traces are compiled by a JIT compiler. The diagram in Figure 2.3 shows how these interpreters are stacked onto each other.

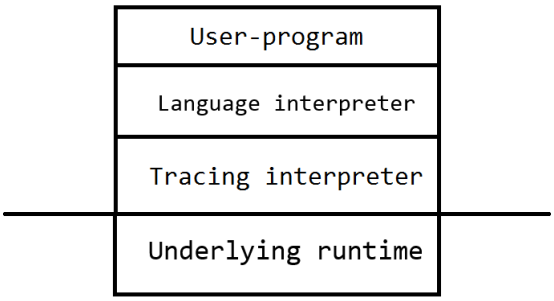


Figure 2.3: The towers of interpreters when meta-tracing

An advantage to this approach is that language implementers can create a regular interpreter for their language, run it with an already existing meta-tracing compiler, and receive the benefits of trace-based compilation without having to go through the effort of developing their own JIT compiler.

2.3.2 Example

Listings 2.3 shows the implementation of a small bytecode interpreter written in Python, based on an example given by Bolz (2012). A bytecode interpreter is an interpreter that executes programs by statically compiling them to a sequence of bytecode instructions, after which these instructions are executed. This interpreter features 256 general-purpose registers, as well as a program counter and an accumulator register. Listing 2.4 presents an example user-program, written in the language implemented by this interpreter, which resembles Python. Listing 2.5 shows the bytecode that is generated when compiling this user-program to bytecode used by the language interpreter.

```
def interpret(bytecode, acc):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        # Jumps to location if acc == 0
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if acc:
                pc = target
        # move acc to register
        elif opcode == MOV_A_R:
            n = ord(bytecode[pc])
            pc += 1
            regs[n] = acc
        # move register to acc
        elif opcode == MOV_R_A:
            n = ord(bytecode[pc])
            pc += 1
            acc = regs[n]
        # add register to acc
        elif opcode == ADD_R_TO_A:
            n = ord(bytecode[pc])
            pc += 1
            acc += regs[n]
        # decrement acc
        elif opcode == DECR_A:
            acc -= 1
        # return acc
        elif opcode == RETURN_A:
            return acc
```

Listing 2.3: A small bytecode interpreter, retrieved from Bolz (2012, Figure 5)

```
res = 0
i = a
while (i != 0):
    i--
    res += a
return res
```

Listing 2.4: The user-program

```
 # i = a
MOV_A_R 0
 # copy of a
MOV_A_R 1
loop:
 # i--
MOV_R_A 0
DECR_A
MOV_A_R 0
 # res += a
MOV_R_A 2
ADD_R_TO_A 1
MOV_A_R 2
 # if i!=0: goto loop
MOV_R_A 0
JUMP_IF_A loop
 # return res
MOV_R_A 2
RETURN_A
```

Listing 2.5: The bytecode for this user-program, retrieved from Bolz (2012, Figure 6)

2.3.3 Matching traces with user loops

Because the input to the tracing interpreter is the language interpreter featured in Listing 2.3 and not the user-program of Listing 2.4, the tracing interpreter cannot know when the user-program loops. A naive tracing interpreter is entirely oblivious of the fact that it is executing another interpreter and executes its input as it would execute any other program: the tracing interpreter identifies and traces hot loops in the language interpreter, instead of hot loops in the user program, as would be the case in direct tracing. This often leads to unac-

ceptable performance. Concretely, it might trace one iteration of the while-loop featured in Listing 2.3. This loop, called the bytecode dispatch loop, takes the next bytecode instruction in the compiled bytecode of the user-program, dispatches over the opcode of this instruction and subsequently executes it.

When this loop has been traced and the next instruction of the bytecode must be executed, the interpreter executes the trace it has just recorded. However, this trace only corresponds to the execution of one specific opcode, e.g., MOVE_R_A, while the compiled bytecode for the user-program consists of instructions with many different opcodes. Although this trace is executed in each iteration of the bytecode dispatch loop, execution immediately needs to be abandoned when it is noticed that the opcode of the current instruction does not match MOVE_R_A. In essence, this problem is caused by the fact that the tracing interpreter traces one loop of the language interpreter, even though it is likely that different iterations of the same loop take completely different paths through the loop. This problem is essential to meta-tracing compilation in general and must be solved to achieve satisfactory performance.

The issue can be solved by making the tracing interpreter trace one loop of the underlying user-program, also called *user loop* instead of a loop of the interpreter. Instead of tracing the execution of one bytecode instruction, the tracing interpreter would then trace the execution of a *sequence* of instructions. Whenever a new iteration of a user loop is started, the language interpreter would execute the same sequence of bytecode instructions again, causing the tracing interpreter to execute the compiled trace instead.

2.3.4 Interpreter hints

The tracing interpreter however does not have any mechanism to detect user loops. Because its input is the language interpreter, it is only aware of loops in the language interpreter. A common solution to this problem is to let the language implementers include hints in the language interpreter they developed (Bolz, 2012; Yermolovich et al., 2009). Listing 2.6 shows how the previous bytecode interpreter is updated to include a small hint, the annotation `can_enter_jit`, from the language developers in the implementation of the JUMP_IF_A opcode.

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                if target < pc:
                    can_enter_jit(target)
```

```
            pc = target
    elif opcode == MOV_A_R:
    ...
```

Listing 2.6: The bytecode interpreter with a hint attached, adapted from Bolz (2012, Figure 8)

Because this language interpreter is a bytecode interpreter that uses a program counter, `pc`, to refer to the instruction that is currently being executed, the program counter can be used as an indicator to detect when a user loop is occurring. The value of this program counter keeps increasing throughout the execution of this user program. For the program to loop, this counter must therefore at some point be reset to a smaller value. This means that we can detect loops by checking on the program counter's value: if it equals a value it has already had before, the user program loops.

The `can_enter_jit` hint is triggered when execution jumps back to a previous point in the program, i.e., when `target < pc` where `target` is the location to where execution must jump. The tracing interpreter processes this hint and makes a note of the fact that execution jumps back to this specific point in the user program's execution. Execution jumping back often enough to this point is an indication that this user loop is hot and that the actions of the language interpreter must be traced. Tracing terminates when this hint, with the same value as target, is triggered again. When such a hint is next encountered, using a target for which a trace already exists, the tracing interpreter executes the recorded trace instead.

Using these interpreter hints effectively allows us to trace user loops instead of loops in the language interpreter.

Chapter 3

Related work

3.1 Overview

3.1.1 Conception

The concept of recording the execution of a program path was introduced independently from each other by Bala et al. in the context of their work on the Dynamo dynamic compiler (Bala et al., 2000), and Deaver et al. while presenting Wiggins/Redstone (Deaver et al., 1999; Aycock, 2003). In both works, a JIT compiler is presented that operates on a program's binary image, i.e., the native instructions that were outputted by some other compiler. This is in contrast with later tracing JITs which work on a higher level: usually on the level of bytecode or some other intermediate representation (Gal et al., 2006, 2009; Chang et al., 2009; Bolz et al., 2009). These early compilers therefore separated the optimizations that can be performed at runtime from the optimizations that can be applied prior to execution. Both Dynamo and Wiggins/Redstone already contain the core concept behind trace-based JIT compilation: a focus on using hot program paths as the basic compilation unit, rather than methods, as was usually done in JIT compilers until then (Aycock, 2003).

3.1.2 HotpathVM

Since their debut, multiple projects have proven that trace-based JIT compilation is a viable alternative to method-based JIT compilation. HotpathVM, a tracing compiler for a JVM intended for use on embedded devices, was released in 2006 by Gal, Probst and Franz (Gal et al., 2006). Although the limited resources available to this compiler restrict the time that can be spent on compiling and optimizing code, HotpathVM is still competitive with more traditional, heavy-weight JIT compilers, showing the potential of trace-based JIT compilation. They mainly accomplish this through a novel use of SSA transformations: instead of transforming an entire control-flow graph into SSA form, only the variables that are actually used in the recorded trace are transformed

into SSA form. This approach requires that the VM explicitly imports the initial context around the trace starting point by moving local variables and stack locations that are used in the trace into SSA variables through so-called ϕ nodes. Correspondingly, SSA variables are moved back onto the stack or in local variables around every side-exit. Gal et al. have named this approach for SSA transformations *Trace* SSA (TSSA). By transforming the recorded trace into SSA form, it becomes much easier to apply compiler optimizations such as loop invariant code motion or common subexpression elimination. Combined with the inherent linearity of traces, it is possible to apply much more aggressive optimizations in a shorter timespan, and with more limited resources for the device.

HotpathVM makes a distinction between three phases during the compilation of the trace: transformation of the trace into SSA form, code analysis and code generation. Each of these phases can be completed in a single pass over the code, making it possible for the compiler to produce optimized machine code in linear time.

3.1.3 TraceMonkey

Tracing JIT compilers have recently attracted some attention as a means for optimizing dynamic languages (Bolz et al., 2009). Mozilla and Adobe jointly released two tracing JIT compilers: TraceMonkey for Javascript (Gal et al., 2009) and Tamarin-Tracing for ActionScript (Chang et al., 2009). TraceMonkey considers side-exit locations as potential trace heads: whenever the execution of a trace must be aborted, e.g., because of a guard failure, the VM starts recording a trace from the point where execution was aborted. This improves runtime performance because, later on, when execution of a trace is aborted again at a point from which a trace has been recorded, the VM can start executing this trace, instead of having to restart interpretation.

TraceMonkey introduces an innovative implementation of *trace trees*. The concept of trace trees refers to the observation that, by considering side-exit locations as potential trace heads, the trace recorder has the tendency to arrange all traces that arise in the form of a tree. This can be problematic in the case where nested loops are being traced. Usually, the inner loop in a nested loop is traced first, since it is also the first to become hot. Exiting the trace of the inner loop causes the trace compiler to start recording from the point of the exit, through the loop header of the outer loop, until execution reaches the inner loop header again. Thus, the trace for the outer loop is essentially contained in the guard trace of the inner loop trace. This can be very problematic in cases where there a lot of different side-exit locations in the inner loop trace, because the outer loop is duplicated in the guard trace for each of these locations leading to excessive amounts of code duplication and, hence, memory consumption. TraceMonkey solves this issue by recording *nested trace trees*. Each loop is represented by its own trace tree, so the code for an outer loop is contained to its own trace tree and not duplicated across the different branches of the trace tree for the inner loop. Furthermore, TraceMonkey's trace recorder

takes the control-flow graph of the program into consideration. This allows the recorder to determine whether a loop is the inner loop of another.

By combining these two aspects, a solution for this problem becomes evident. Instead of naively starting tracing at the side-exit of an inner loop and continuing recording the guard trace across the outer loop header, the trace recorder detects when it has reached the outer loop and aborts the guard trace here. Later on, if it traces the outer loop and encounters the inner loop header, it detects that a trace tree for this inner loop already exists so it records a *call* to this inner loop. Thus, TraceMonkey's approach essentially boils down to forming trace trees for each different loop and then linking these trace trees together via calls.

3.1.4 Other tracing compilers

Inoue et al. (2011) retrofitted a tracing JIT compiler from a method-based JIT compiler for Java. They compared both kinds of JIT compilations and noted that the increased compilation scope available in trace-based JIT compilation has a large positive effect on the quality of the code that is emitted by the compiler. Specifically, they found that method-based JIT compilers are very successful in programs with an execution profile containing some hot spots, but they are more limited when dealing with programs that have a flat execution profile. This is largely due to the fact that method-based JIT compilers do not aggressively inline 'cold' methods, in order to avoid unnecessary code duplication. Tracing JIT compilers do not suffer from this issue because they trace program execution paths and hence automatically inline everything. Inoue et al. also noted that tracing JIT compilation comes at the cost of an increased runtime overhead, which partially offsets the advantages gained by tracing compilation.

Other important organizations, including Microsoft and the Lua community, have released JIT compilers that are at least partly based on the techniques from tracing (Bebenita et al., 2010; Pall, 2013).

3.2 Formal frameworks

Guo and Palsberg have provided a formal foundation to model trace recording in order to prove the soundness of optimizations applied on traces by trace-based JIT compilers. Their model is based on the notion of bisimulation (Guo & Palsberg, 2011). The concept of bisimilarity was first raised in the context of concurrency theory and can intuitively be thought of as a relation between two program-states. Two states are bisimilar if each action on one state can be matched by an action on the other state, and if the resulting states can again be proven to be bisimilar (Sangiorgi, 1998). Guo & Palsberg defined a small language and identified some of the essential aspects required to enable tracing a program's execution. They described these aspects through a formal semantics

which allows them to formally reason about the workings of a tracing compiler and the optimizations that can be employed by them. Guo & Palsberg concluded that certain optimizations, such as dead-store elimination where assignments to variables that are not used in the trace, commonly used in the more traditional, method-based JIT compilation, are unsound when applied in the setting of tracing JIT compilation.

To describe a program's execution in their model, they use two sets of evaluation rules. The first set is used to describe the program's execution when no trace is being recorded. The other set of evaluation rules is almost identical to the first, but is used during the recording of a trace.

Logozzo et al. later adapted the work of Guo and Palsberg to provide another framework which can also be used to prove the soundness of optimizations ons traces. They proposed an alternative framework, based on abstract interpretation, which, as they claim, is less restrictive and more closely models real-world tracing JIT compilers (Dissegna et al., 2014). Using their framework, it becomes possible to prove the correctness of certain optimizations which could not be proven to be sound in Guo and Palsberg's model. In their framework, Dissegna et al. do not model how traces are recorded concretely. They start by defining a set of all possible traces for each program-state, based on a heuristic that determines which traces can exist for each state.

The functionality of both formal models is fairly limited. Both frameworks are bound to one particular execution model, for one particular programming language. If someone wishes to employ their model to investigate an entirely different execution model, they must convert their model to the execution semantics used by either of both formal models. Even then, these frameworks can only be used to explore the soundness of trace optimizations.

Both formal models were also aimed completely at understanding direct tracing. They cannot be used to investigate meta-tracing compilers.

Other than these two models, not much research has gone towards investigating the formal foundation of tracing JIT compilation, although in Bolz, Cuni, FijaBkowski, et al. (2011), where an allocation removal optimization is presented, a formal description of the optimization is also presented. The scope of this formal model however is entirely limited to describing the optimization that they demonstrated.

3.3 Trace selection strategies

A trace selection strategy determines when a tracing JIT compiler starts and stops tracing. These strategies are important because they have a large impact on the runtime profile of the JIT and as such deserve some attention. Several trace selection strategies have been explored in order to improve not only runtime performance, but also e.g. memory consumption in tracing JIT compilers.

The seemingly most common trace selection strategy, used in e.g., Dynamo (Bala et al., 2000), LuaJIT (Pall, 2013) and PyPy (Bolz et al., 2009), is based around a technique called *most recently executed tail* (MRET). Algorithms based

on this approach identify a number of *potential trace heads*, program points from where a trace can be started such as the target addresses of backwards branches or side-exits in existing traces, and subsequently observe how many times these heads are executed. If this execution count rises above a certain threshold, the compiler starts tracing. Tracing continues until an end-of-trace condition is reached (Bala et al., 2000). What constitutes an end-of-trace condition depends on the exact strategy under evaluation, but these conditions commonly include *repetition detection*, *buffer overflows*, which arise when the trace becomes too long and the recording buffer overflows, or the formation of irregular events, such as throwing an exception or invoking a native call (Wu et al., 2011).

Repetition detection is used to detect cyclic repetition paths in the recorded trace. Several strategies are known for implementing this detection, such as *stop-at-backwards-branch*, which stops tracing at *every* backwards branch, *cyclic-path-based* which works by checking whether the current value of the operation counter has already been encountered during trace recording, and *static-scope-based* strategies which leverage static information about the program structure (Hayashizaki et al., 2011).

Hayashizaki et al. (2011) have warned against a too naive implementation for checking this repetition detection condition. They coined the term *false loop* to signify cyclic paths, i.e., "paths that start and end at the same instruction address" (Hayashizaki et al., 2011), but which do not form a cyclic execution path, i.e., a loop. A common source of this issue consists of functions that have multiple invocation sites, as shown in Listings 3.1, which was adapted from another example used in Baumann et al. (2015).

```
def g(x, y):
    return x + y

def f():
    while True:
        g()
        g()
```

Listing 3.1: Example of a false loop in Python

In this case, tracing might start at the first invocation of `g`. This continues through its body and returns to the `while`-loop of `f` until it finally stops at the second invocation of `g`. Although the JIT has now concluded that it has traced a full loop, it has actually only traced the execution of an invocation to `g`. These kinds of short, non-looping traces are usually not the ideal targets for tracing, because optimizing them has only a smaller effect, and exits from these traces are more frequent, leading to performance degradation. Hayashizaki et al. proposed an approach they named *false loop filtering* to detect these false loops. They also investigated a number of potential techniques to implement this approach, with varying levels of accuracy and runtime overhead.

Wu et al. (2011) have proposed a number of techniques targeted at trace selection in order to improve the memory consumption of the JIT compiler. They

evaluated a specific MRET-based selection strategy, taken from the work of (Inoue et al., 2011), and evaluated this strategy in function of its *space efficiency*: its ability to maximize steady-state performance while minimizing memory consumption. They identified two types of sources of space inefficiency: the formation of short-lived traces and non-profitable trace duplication. Furthermore, they proposed six techniques to improve space efficiency. Although their work was mainly intended to improve memory usage of the JIT compiler, they noted that an increased space efficiency should also lead to less frequent instruction cache misses and hence an improved steady-state performance.

3.4 Meta-tracing

The idea of meta-tracing is quite recent and as such has only received minimal attention in research. Conceptually, this thesis is very related to the PyPy project, one of the first projects to introduce the concept of meta-tracing (Bolz, 2012). PyPy was started around 2003 as an attempt to create a minimalistic meta-circular Python interpreter, for educational purposes, but it has since moved on to become a subject for research towards high-performance VM's (Fijalkowski, 2013), specifically through a focus on JIT compilation.

3.4.1 The RPython project

RPython is a proper subset of Python, developed to combine the flexibility offered by Python, with the high performance program execution generally achieved in statically typed languages. To this end, RPython has eliminated some features found in regular Python, such as multiple inheritance and reflection. It also restricts the typing rules found in regular Python: in contrast with regular Python, RPython is statically typed, although types can be inferred and must not be explicitly stated by the programmer. This allows for a generous speed-up, making RPython competitive with languages such as C# and Java (Ancona et al., 2007).

The PyPy framework has grown into a whole development environment, based around RPython. The framework is aimed to create high-performing, but easy-to-write, interpreters. The original Python interpreter, PyPy, is just one of these interpreters that were created. The PyPy framework is also often called the *RPython framework*, because interpreters developed in this framework must be written in RPython.

The toolchain of this framework, called the *translation toolchain*, could be used to translate any interpreter written in RPython into the same interpreter but built on another platform, such as C, CLI or the JVM, by following a series of transformation steps. Figure 3.1, taken from , shows how a Python interpreter in the PyPy framework is translated into a C executable. During this translation, the toolchain inserts a number of low-level services into the interpreter, such as the components responsible for the memory- or threading-model. This removes the burden of implementing these details from the lan-

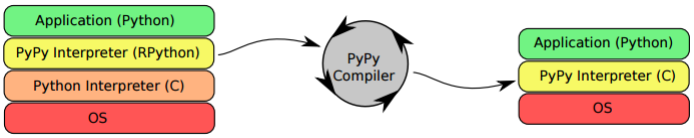


Figure 3.1: Translating a language interpreter to C, retrieved from Paška (2012)

guage developers, allowing them to only focus on the high-level implementation details.

To a certain extent, language developers have the option to select which components they wish to include. One of these components that can be included is a tracing JIT compiler, called MetaJIT (Bolz et al., 2009). This JIT compiler does not trace the execution of the user-program being run however, but instead the execution of the interpreter while it runs this program, so it is indeed a *meta-tracing* compiler. MetaJIT follows a trace selection strategy based on the MRET algorithm, where the execution of a number of potential trace heads are monitored. In the case of RPython's meta-tracing compiler, only the target addresses of backward branches, i.e., the program counters, are monitored.

A meta-tracing compiler ideally records the execution of a loop in the *user-program*, instead of the execution of a loop in the *language interpreter*. In the common case where the language interpreter is a bytecode interpreter, loops can be detected by checking the value of the program counter. This program counter keeps increasing unless the language interpreter executes a backwards branch. However, because the tracing compiler has no knowledge on the semantics of the target language for the language interpreter, it cannot know when loops occur. In other words, the tracing compiler does not know when a backjump is executed by the language interpreter and hence cannot know when a possible starting point for the trace is reached.

To solve this issue, language developers are required to annotate their interpreter with several hints, designed to transfer specific knowledge on the execution of the user-program from the language interpreter to the tracing compiler.

In Listings 3.2, retrieved from Bolz et al. (2009), an example is shown of how annotations can be placed in a generic bytecode interpreter created in the PyPy framework.

```
tlrjitdriver = JitDriver(greens = ['pc', 'bytecode'],
                         reds = ['a', 'regs'])


def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        tlrjitdriver.jit_merge_point()
```

```
opcode = ord(bytecode[pc])
pc += 1
if opcode == JUMP_IF_A:
    target = ord(bytecode[pc])
    pc += 1
    if a:
        if target < pc:
            tlrjitdriver.can_enter_jit()
        pc = target
elif opcode == ...
```

Listing 3.2: The annotations used in PyPy

The code shows the implementation of the interpreter's bytecode dispatch loop. This loop is responsible for the execution of the bytecode: each iteration of this loop corresponds with the execution of one instruction. Two annotations have been placed inside this loop: `tlrjitdriver.jit_merge_point()` and `tlrjitdriver.can_enter_jit()`. The first annotation should always be placed at the beginning of the dispatch loop and is used to signal to the tracing interpreter where to bail to when execution of the trace must be aborted. The `tlrjitdriver.can_enter_jit()` annotation should be placed around each location where a user loop can be closed. As mentioned before, in the case of a bytecode interpreter, these are the places where a backjump is executed. These two annotations provide enough information to the tracing interpreter to enable it to record traces matching the user loops instead of the interpreter loops.

Other than the PyPy interpreter, several other interpreters have been built on top of the RPython framework by now, including interpreters for Haskell, Prolog and PHP (Thomassen, 2013; Bolz et al., 2010; Homescu & Şuhan, 2011). Although seemingly less common, non-bytecode interpreters have been created in this framework as, well, including Pycket, an interpreter for Racket, which was based on the design of a CEK-machine (Bolz et al., 2014).

This last interpreter is noteworthy, because developing an interpreter for Racket raises a number of inherent challenges, such as the need to transform most special forms of the language to a small set of core forms. Furthermore, because the Pycket interpreter is not a bytecode interpreter, it does not store an operation counter, making it significantly more difficult to find loops in the user-program.

This problem is further aggravated by the fact that the only core Racket form that can cause the program to loop is the function application form, through function recursion. However, treating the body of every lambda form as a trace header would lead to redundant tracing and trace duplication. To solve this issue, Bolz et al. relied on the work mentioned earlier here by Hayashizaki et al. (2011) and compared two techniques to improve trace selection.

Allocation removal

Bolz, Cuni, FijaBkowski, et al. (2011) discuss a strategy for *allocation removal* in the traces that are generated by MetaJIT. This optimization focuses on re-

moving redundant object allocations in traces. Object allocations are especially costly in dynamic programming languages, because the absence of static type information forces the interpreter to *box* all primitive objects. That is, primitive objects are wrapped in a small structure containing e.g., a tag which specifies the type of the object. Because these boxes are allocated on the heap, memory consumption increases, and with it, pressure on the garbage collector. A large performance boost can therefore be achieved by avoiding object allocation wherever possible.

Once a trace has been recorded, the allocation removal optimization is run on this trace. Through a process called *escape analysis*, the optimization places all objects that are created in this trace in one of four categories, depending on whether and how they escape the trace. The escape analysis algorithm traverses the trace in a single pass and optimistically assumes that every object it encounters will escape the trace. It then replaces these objects by *static objects*, which contain their value as it was observed at runtime, and removes their allocation. Since the value of the object is now stored in a static object, subsequent operations that depend on the value of this object can be resolved statically by the algorithm and can hence be removed, assuming that the operation does not produce any side-effects on non-static objects. If the escape analysis algorithm later determines that a static object does escape the trace, it replaces this static object again by a dynamic, boxed, object in a process called *lifting*. Note that the use of these static objects also automatically enables a *constant folding* optimization, where operations that depend on objects whose value is known, either at compile time in the case of ahead-of-time compilers or at runtime in the case of JIT compilers, can be removed, since the result of these operations can be statically determined.

Bolz, Cuni, FijaBkowski, et al. have noted that this optimization successfully removes between 4% and 90% of all allocations, resulting in a speedup between 20% and 695%. They have also presented a formal description of this optimization.

Constant folding

This optimization has already been discussed in the context of the allocation removal optimization. Bolz et al. have argued that this optimization is especially useful when meta-tracing, because a regular, unoptimized, trace would be dominated by instructions manipulating interpreter structures, such as the bytecode string and the operation counter. Remember however that these are 'green' variables. The tracer therefore depends on the values of these variables to decide which, if any, trace to execute. It is therefore safe to assume that the value of these variables is fixed at the beginning of the trace, so for all intents and purposes it can be argued that their value is constant. This makes it possible to constant fold away all operations that depend on their values, assuming these operations are side-effect free. When applying this optimization on the PyPy meta-tracing compiler, a 4.7 speedup is reported on a simple benchmark.

Bolz, Cuni, Fijalkowski, et al. have later presented a more powerful tool for applying constant folding in a meta-traced interpreter. They presented two additional annotations which can be used by the language developers to offer extra information about the runtime profile of the language interpreter to the tracing compiler, making it possible to apply not only constant folding on some small set of variables, such as the aforementioned 'green' variables or variables only allocated in the trace, but on effectively any variable. In essence, to constant fold away an operation on a variable, two conditions need to be ensured once the trace has been recorded:

- The value can be statically determined by the optimizer
- The operation is free of any side-effects

Note that, even though we are dealing with a JIT compiler and hence we know the value of any variable encountered while tracing, the first condition is not automatically satisfied, because the value of a variable can change between subsequent runs of the trace. It is therefore not guaranteed that the optimizer can know the value of a variable at an arbitrary point in time.

These two extra annotations were introduced to solve both conditions. The first annotation, *promote*, can be applied on any variable used in the interpreter. It signals back to the compiler that the value of this variable rarely changes and is thus, in practice, constant throughout the execution of the program. When encountering this annotation during tracing, the tracing compiler inserts a guard for this variable in the trace to check whether the value of the variable is equal to the value that was encountered during tracing. This then allows the compiler to treat this variable as a constant for the remainder of the trace. This annotation therefore solves the first condition.

The second condition is solved by introducing the *elidable* annotation. By marking functions in the interpreter as *elidable*, language developers state to the tracing compiler that the function is pure, i.e., free of side-effects, and hence also referentially transparent: any call of this function in a program on a certain input can be replaced by the result of this function call on that input. This means that a call to a elidable function with a constant (set of) arguments can be replaced by just the result of this function that was observed at runtime, during the recording of the trace.

Listings 3.2 shows an example of how these two annotations could be used in an interpreter. The resulting, optimized, trace is shown to the right.

Note that the very first instruction in this trace is a guard which checks the value of the variable `a`. This guard is a result of the `promote` annotation in the function `f`. Because this guard is placed in the trace, the compiler is automatically saved from incorrect usage of this annotation. If a variable which frequently changes value is promoted, a guard failure is triggered when running the optimized trace. Although the frequent side-exits lead to performance degradation, they do not cause any incorrect behavior while running the interpreter. The *elidable* annotation on the other hand does introduce bugs when

```
class A(object):

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def f(self, val):
        promote(self)
        self.y = self.c() + val

    @elidable
    def c(self):
        return self.x * 2 + 1

a1 = A(4, 0)
# address of a1 = 0xb73984a8
a1.f(10)
```

```
guard(a1 == 0xb73984a8)
# a1.c() = 9 iff a1.x = 4
v2 = 9 + val
a1.y = v2
```

Figure 3.2: The *promote* and *elidable* annotations in RPython

used incorrectly, although Bolz, Cuni, Fijalkowski, et al. do discuss some future work where a debugging mode in the compiler would check the validity of this annotation.

Type specialization

Because in the context of meta-tracing the program being traced is an interpreter, this constant folding technique can automatically be used as a type specialization optimization. It suffices for language developers to mark the types of values as constant and make sure that the corresponding functions are elidable, and all instructions that depend on the type of a value are folded away, effectively enabling type specialization in the trace.

3.4.2 Hierarchical VM layering

Simultaneously with the creation of RPython's meta-tracing compiler framework, Yermolovich et al. (2009) have explored the feasibility of meta-tracing. They investigated whether it would be possible to run a VM for a dynamic language, which they named the *guest VM* on top of another, more mature, high-performing VM, named the *host VM*. In their set-up, the guest VM was LuaVM while the host VM was the Tamarin-Tracing VM. Because the LuaVM is written in C while Tamarin-Tracing executes ActionScript code, the code for the LuaVM was first converted to ActionScript using the Adobe Alchemy extension of the LLVM compiler framework. Yermolovich et al. also identified the fundamental challenge behind the meta-tracing approach, namely that the host VM does not realize it is executing another VM, causing it to optimize this guest VM just like it would optimize any other program. They initially used

the program counter of the Tamarin-tracing VM to detect loops, but just like the naive implementation of RPython's tracing compiler, this caused the host VM to trace the main interpreter dispatch loop, leading to very frequent guard failures. To solve this, they introduced hints into the guest VM which were used to signal back the value of the guest VM's program counter. The value of this program counter was then combined with the value of the host VM's program counter into a *virtual program counter* and this virtual program counter was subsequently used to detect loops. This approach is similar to the 'green variables' used in RPython.

Chapter 4

The SLIPT language

We introduce our input language, SLIPT, by presenting the formal semantics that specify the semantics of this language. SLIPT is a variant of Scheme, but lacks features such as macros or call/cc-style continuations. The formal semantics are defined as a set of rules operating on a CESKθ-machine, which is itself a novel adaptation of the more common CESK-machine (Felleisen & Friedman, 1987). In this chapter, we start by explaining the meaning and purpose behind CESK-machines in general to lay a solid foundation for further sections. From Section 4.2 onwards, we present SLIPT itself. We first specify the syntax of this language. Afterwards we express the semantics of this language as a set of transition rules operating on a CESKθ-machine. We then continue by introducing a low-level instruction set for the CESKθ-machine. These instructions are just a set of rules that operate directly on the CESKθ-machine, instead of on SLIPT. Introducing these instructions then allows us to redefine our CESKθ-machine in a new setting. We have provided an implementation for the CESKθ-machine, written in Racket. In the final section, we describe how this implementation works.

4.1 CESK-machines

4.1.1 Overview

CESK-machines are deterministic state-transition systems that can operate as formal specification models for the semantics of any language, i.e., the different features and aspects of that language (Felleisen & Friedman, 1987; Felleisen, 1988). They can be used to formalize the semantics of this input language: the meaning behind the various features and aspects of the language. Their state is composed of four components: a control (C), an environment (E), a store (S) and a continuation component (K). The analogy has been made between these four components and the instruction pointer, the local variables, the heap and the stack respectively (Might, 2015). The control component refers to the

location in the program that is currently being evaluated by the interpreter. Depending on the language that is used, as well as the implementation of the interpreter, the control can be an expression, a program counter referring to a set of bytecode instructions etc. The environment is a mapping from variable names to addresses, while the store is mapping from addresses to values. Together, these components are used to lookup the value of variables in the program.

An alternative approach would be to combine the environment and the store into a single component that maps names directly to values. This is exactly what is done in CEK-machines. A CEK-machine is identical to a CESK-machine except that it lacks a store component and uses the environment to bind variables to values directly. However, CESK-machines have some advantages over CEK-machines that make it easier to perform abstract reasoning over the workings of the machine. For example, when creating a closure, the closure might need to refer to its own lexical environment. When we bind this closure to a variable, we effectively place the closure itself in the environment. If we were to only use an environment and not a store, then from an abstract point of view, we have an object that points to the same location in which it is currently residing (Van Horn & Might, 2010). Using a store solves this issue, because the closure itself resides in the store instead of the environment. In order to simplify potential future research on this thesis, we have therefore opted to base our research on a CESK- instead of a CEK-machine.

The continuation component represents the future of the computation. The continuation is often modelled as a stack of frames where each frame represents the actions the machine must take when evaluation of the current control component is completed.

The transition rules between the possible states of the CESK-machine then define how a program is executed by the machine In other words, it is these rules that define the actual semantics of the language under consideration.

4.1.2 Example

As an example of how CESK-machines are used, suppose a language that has assignments such as the following:

$$x := 1 + 1$$

The relevant transition rules could then be expressed as in Figure 4.1:

$$< var := e,\ \rho,\ \sigma,\ \kappa > \rightarrow < e,\ \rho,\ \sigma,\ \mathbf{setk}(var) : \kappa > \tag{4.1}$$
$$< v_1 + v_2,\ \rho,\ \sigma,\ \kappa > \rightarrow < n_1 \oplus n_2,\ \rho,\ \sigma,\ \kappa > \tag{4.2}$$
$$< v,\ \rho,\ \sigma,\ \mathbf{setk}(var) : \kappa > \rightarrow < v,\ \rho,\ \sigma[\rho(var)/v],\ \kappa > \tag{4.3}$$

Figure 4.1: A set of transition rules for a CESK-machine

In these rules, each tuple corresponds to a particular state of a CESK-machine. The first field in the tuple is the control component, c, the second the environment, ρ, the third the store, σ, and the fourth field is the continuation stack, κ. We use the following terminology in these semantics: var refers to a variable, e to a general expression and v is always a value: i.e., an expression that cannot be evaluated any further, such as a number or a boolean.

In rule 4.1, we describe how to evaluate general expressions of the form $var := e$: we push a certain continuation, the $\mathbf{setk}$ continuation on the stack, and continue with the evaluation of the right-hand expression of the assignment. The $\mathbf{setk}$ continuation stores a reference to the name of the variable being assigned to so that, when the evaluation of the right-hand expression completes, we can pop this continuation from the stack, extract this variable's name and update its value. Rule 4.2 expresses how to evaluate the sum of two values in a program. Assuming these values are indeed numbers, we only have to actually add these numbers together. In rule 4.3, we state that if the control component currently contains a value, which per definition cannot be evaluated any further, and the top of the continuation stack holds a $\mathbf{setk}$ continuation, we pop this continuation, extract the variable's name from it, and then perform the actual update of this variable. To modify a variable's value, we first have to fetch the address of this value from the environment, which is expressed here as $\rho(var)$, and then update the location to which this address is pointing.

By applying these general transition rules on the example assignment shown previously, we obtain the series of CESK-states shown in Figure 4.2.

$$
\begin{aligned}
&< x := 1 + 1,\ \rho,\ \sigma,\ \kappa > \rightarrow \\
&< 1 + 1,\ \rho,\ \sigma,\ \mathbf{setk}(x) : \kappa > \\
&< 2,\ \rho,\ \sigma,\ \mathbf{setk}(x) : \kappa > \\
&< 2,\ \rho,\ \sigma[\rho(x)/2],\ \kappa >
\end{aligned}
$$

Figure 4.2: A concrete example of evaluating an assignment

Evaluating this assignment, and by extension, evaluating programs in general, is a combination of evaluating the expression stored in the control, pushing and popping the correct continuations from the continuation stack and then acting on these continuations.

4.2 Syntax

The syntax of our input language, SLIPT, is based on SLIP, a derivative of Scheme designed by Theo D'Hondt for educational purposes (D'Hondt, 2015). SLIP has Scheme syntax, but omits certain aspects, such as macros and forward declaration of variables. SLIPT also lacks these features, as well as a native `eval` function and call/cc-style continuations.

The formal semantics below specify the syntax of the input language.

$$
\begin{aligned}
e \in \mathit{Exp} &= \mathit{Application} \\
&\mid (\texttt{apply}\ \mathit{Exp}\ \mathit{Exp}) \\
&\mid (\texttt{begin}\ \mathit{Exp}*) \\
&\mid (\texttt{define}\ \mathit{Variable}\ \mathit{Exp}) \\
&\mid (\texttt{define}\ \mathit{DefineFunctionPattern}\ \mathit{Exp}*) \\
&\mid (\texttt{if}\ \mathit{Exp}\ \mathit{Exp}) \\
&\mid (\texttt{if}\ \mathit{Exp}\ \mathit{Exp}\ \mathit{Exp}) \\
&\mid (\texttt{lambda}\ \mathit{LambdaParameters}\ \mathit{Exp}*) \\
&\mid \mathit{Literal} \\
&\mid \mathit{Native\ function} \\
&\mid (\texttt{set!}\ \mathit{Variable}\ \mathit{Exp}) \\
&\mid \mathit{Variable} \\
\mathit{Application} &= (\mathit{Exp}+) \\
\mathit{DefineFunctionPattern} &= (\mathit{Variable}+) \\
&\mid (\mathit{Variable}+\ .\ \mathit{Variable}) \\
\mathit{LambdaParameters} &= (\mathit{Variable}+) \\
&\mid (\mathit{Variable}+\ .\ \mathit{Variable}) \\
&\mid \mathit{Variable} \\
x \in \mathit{Variable} &= \mathit{Identifier} \\
\mathit{Literal} &= \mathit{Boolean} \\
&\mid \mathit{Number} \\
&\mid \mathit{String} \\
&\mid \mathit{Symbol}
\end{aligned}
$$

Figure 4.3: The syntax of SLIPT

SLIPT contains both native and user-defined functions. A user-defined function is a function created by the programmer either through a `lambda`expression, or by defining a function directly using the (`define` *DefineFunctionPattern Exp*$*$) syntax.

4.3 CESKθ-machine

4.3.1 Overview

We express the semantics of SLIPT as a CESKθ-machine, which is a variation on the CESK-machine. A CESKθ-machine is a six-tuple which, in addition to the four components used in regular CESK-machines, i.e., control, environment, store and continuation, also contains a value register (v) and a value stack (θ). The value register stores the result of evaluating an intermediate expression. The value stack is mainly used for evaluating function applications and, to a lesser extent, the evaluation of sequences of expressions. For example, when evaluating a function application, we push all evaluated arguments onto this stack, and we pop them from that stack when binding the parameters of the function. This stack also saves the environment before evaluating certain expressions, so that the environment of the program can be restored at a later point in time. These two additional components are not strictly required, but their inclusion offers some advantages.

In the small CESK-machine that was shown in Section 4.1, evaluated expressions, i.e., *values*, were stored in the control component. Suppose however that we are evaluating a large expression, consisting of many different subexpressions that also need to be evaluated, and whose values need to be stored until evaluation of the compound expression is complete. By including a value register and a value stack, we can easily store these intermediate values for as long as is required.

In contrast to the previously presented CESK-machine, the control component of a CESKθ-machine not only stores expressions, but also continuations that were popped from the continuation stack. This has the advantage that it is now always clear when we are actually evaluating an expression, and when we are following a continuation: if the control contains an expression we are evaluating, else we are following a continuation.

4.3.2 CESKθ definition

Figure 4.4 shows the formal definition of the CESKθ-machine.

$$
\begin{aligned}
\varsigma \in \mathit{ProgramState} &= \mathbf{ps}(\mathit{Control},\ \mathit{Env},\ \mathit{Store},\ \mathit{KStack},\ \mathit{Val},\ \mathit{VStack}) \\
\alpha \in \mathit{Address} &= \mathit{Identifier} \\
\mathit{clo} \in \mathit{Closure} &= \mathbf{clo}(\mathbf{lam}(x,\ e),\ \rho) \\
\mathit{Control} &= \mathit{Exp} \\
&\mid \mathit{Kont} \\
\rho \in \mathit{Env} &= \mathit{Variable} \to \mathit{Address} \\
\phi \in \mathit{Kont} &= \mathbf{aplck}() \\
&\mid \mathbf{applyk}(\mathit{rator}) \\
&\mid \mathbf{definek}(x) \\
&\mid \mathbf{haltk}() \\
&\mid \mathbf{ifk}(e_1,\ e_2) \\
&\mid \mathbf{randk}(e,\ es,\ i) \\
&\mid \mathbf{ratork}(i) \\
&\mid \mathbf{seqk}(es) \\
&\mid \mathbf{setk}(x) \\
\kappa \in \mathit{KStack} &= \mathit{Kont}\ :\ \mathit{KStack} \\
&\mid \epsilon \\
\sigma \in \mathit{Store} &= \mathit{Address} \to \mathit{Val} \\
\mathit{val} \in \mathit{Val} &= \mathit{Literal} \\
&\mid \mathit{Closure} \\
&\mid \mathit{Native\ function} \\
\theta \in \mathit{VStack} &= (\mathit{Val}\ +\ \mathit{Env})\ :\ \mathit{VStack} \\
&\mid \epsilon
\end{aligned}
$$

Figure 4.4: The CESKθ-machine

In our definition, CESKθ-states are also called *program-states*. We define continuations (Kont) for evaluating `apply`, `set!`, `define`, `if` and `begin`-expressions. We use respectively the continuations **applyk**, **setk**, **definek**, **ifk** and **seqk** for these expressions. For function applications, we use the continuations **randk**, **ratork** and **aplck**. Additionally, we use a special halting continuation that signals the end of program execution. This halting continuation (**haltk**) is pushed to the continuation stack at the very start of the program execution.

Like CESK-machines, the environment (Env) is a mapping from variables to addresses, while the store (Store) is a mapping from these addresses to values. A continuation stack (KStack) is a list of continuations, and the value stack (VStack) is a list of values or environments. A closure is a combination

of a lambda and its lexical environment. A function is represented as a list of variables, its parameters, and its body. Addresses are unique identifiers. They could be defined as e.g., natural numbers, but their exact definition is irrelevant in the context of this thesis. Values, i.e., expressions that cannot be evaluated any further, are either the literals we defined in the previous section, or closures.

4.3.3 Evaluation rules

We now present the evaluation rules for SLIPT programs, which define the actual semantics of SLIPT. These rules are expressed as transition rules between various program-states. We explain how each rule is constructed and we provide mnemonic labels to refer to each rule.

Variables and values

Evaluating a variable lookup:

$$\mathbf{ps}(x,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{evar}} \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ \sigma(\rho(x)),\ \theta)$$

Evaluating a value:

$$\mathbf{ps}(val,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{evalu}} \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ val,\ \theta)$$

Evaluating a call to `lambda`:

$$\mathbf{ps}((\texttt{lambda}\ x\ .\ es),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{elmbd}} \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ \mathbf{clo}(\mathbf{lam}(x, es), \rho),\ \theta)$$

Figure 4.5: Handling variables and values

Evaluating a variable is accomplished by looking in the environment for the address associated with this variable, and subsequently using this address to look up the variable's value in the store. This value is then moved to the value register. To handle values, such as numbers, strings, booleans, native functions or closures, we only have to copy the value to value register, since no actual evaluation has to be performed. `elmbd` states how calls to `lambda` are handled, and subsequently, how user-defined functions or closures are constructed. `lambda` takes two arguments: a list of variables, its parameter list, and a list of expressions, the body of the function. From these two elements, we create a **lam** structure. We wrap this structure in a **clo** and add the current environment, which serves as the lexical environment of the closure.

Definitions and assignments

Evaluating the definition of a variable:

$$\mathbf{ps}((\texttt{define}\ x\ e),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{edfvr}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{definek}(x) : \kappa,\ v,\ \rho : \theta)$$

Syntactic sugar for defining a function:

$$\mathbf{ps}((\texttt{define}\ (f\ .\ pars)\ .\ body),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{edffn}} \mathbf{ps}(\phi,\ \rho[name \to \alpha],\ \sigma[\alpha \to c],\ \kappa,\ c,\ \theta)$$

where c equals $\mathbf{clo}(\mathbf{lam}(pars, body), \rho)$

Completing the definition of a variable after having evaluated its expression:

$$\mathbf{ps}(\mathbf{definek}(x),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{kdefv}} \mathbf{ps}(\phi,\ \rho'[x \to \alpha],\ \sigma[\alpha \to v],\ \kappa,\ v,\ \theta)$$

where α is a new, unique adress

Evaluating an assignment:

$$\mathbf{ps}((\texttt{set!}\ x\ e),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{eset!}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{setk}(x) : \kappa,\ v,\ \rho : \theta)$$

Completing the evaluation of an assignment to a variable:

$$\mathbf{ps}(\mathbf{setk}(x),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{kset}} \mathbf{ps}(\phi,\ \rho',\ \sigma[\rho'(x) \to v],\ \kappa,\ v,\ \theta)$$

Figure 4.6: Evaluating definitions and assignments

To define a variable x, we evaluate the expression whose value will be bound to x and we push $\mathbf{definek}(x)$ to the continuation stack. When evaluation of e is completed, we pop this continuation, extract the variable name x and update the environment and the store so that x is bound to the value v that has just been evaluated.

Similar to Scheme, we provide the programmer with syntactic sugar for defining functions. Defining a function is similar to directly creating this function by using `lambda` and immediately binding this closure to a variable.

Assigning a variable is similar to defining a new variable: we first evaluate the expression e and we push a $\mathbf{setk}(x)$ continuation. When evaluation of e is completed, we pop this continuation from the stack, retrieve the variable name x, use the environment to locate the address of this variable in the store, written here as $\rho'(x)$, and assign the new value to this address.

If-expressions

Like Scheme, an if-expression consists of two or three subexpressions: the if-condition, the true-branch and possibly a false-branch.

Evaluating an if-expression, with a potential else-branch:

$\mathbf{ps}((\texttt{if}\ e\ e1\ .\ e2),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{eif}}$
$\quad \mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{ifk}(e1,\ e2)) : \kappa,\ v,\ \rho : \theta)$

Completing an if-expression without an else-branch whose condition evaluated to $\#\texttt{f}$:
$\mathbf{ps}(\mathbf{ifk}(e1,\ `()),\ \rho,\ \sigma,\ \phi : \kappa,\ \#\texttt{f},\ \rho' : \theta) \xrightarrow{\texttt{kifsf}}$
$\quad \mathbf{ps}(\phi,\ \rho',\ \sigma,\ \kappa,\ `(),\ \theta)$

Completing an if-expression without an else-branch whose condition did not evaluate to $\#\texttt{f}$:
$\mathbf{ps}(\mathbf{ifk}(e1,\ `()),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{kifst}}$
$\quad \mathbf{ps}(e1,\ \rho',\ \sigma,\ \kappa,\ v,\ \theta)$

Completing an if-expression with an else-branch whose condition evaluated to $\#\texttt{f}$:
$\mathbf{ps}(\mathbf{ifk}(e1,\ e2),\ \rho,\ \sigma,\ \kappa,\ \#\texttt{f},\ \rho' : \theta) \xrightarrow{\texttt{kifdf}}$
$\quad \mathbf{ps}(e2,\ \rho',\ \sigma,\ \kappa,\ v,\ \theta)$

Completing an if-expression with an else-branch whose condition did not evaluate to $\#\texttt{f}$:
$\mathbf{ps}(\mathbf{ifk}(e1,\ e2),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{kifdt}}$
$\quad \mathbf{ps}(e1,\ \rho',\ \sigma,\ \kappa,\ v,\ \theta)$

Figure 4.7: Evaluating if-expressions

When evaluating an if-expression, we first evaluate its condition and we push $\mathbf{ifk}(e_1,\ e_2)$ to the continuation stack. Once the condition has been evaluated, we pop the **ifk** continuation. How evaluation proceeds depends on the condition's value and on whether or not a false-branch was provided. If the condition evaluated to `#f` but no false-branch was given, the if-expression evaluates to `'()`, as stated by the `kifsf` rule. In other words, expressions such as `(if #f 99)` evaluate to `'()`. In any other case, we continue evaluation through the branch that corresponds with the value of the condition.

Sequences

Evaluating a `begin`-expression consisting of zero subexpressions:
$\mathbf{ps}((\texttt{begin}),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{ebgn0}}$
$\quad \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ `(),\ \theta)$

Evaluating a `begin`-expression consisting of exactly one subexpression:
$\mathbf{ps}((\texttt{begin}\ e),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{ebgn1}}$
$\quad \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta)$

Evaluating a `begin`-expression consisting of more than one subexpression:
$\mathbf{ps}((\texttt{begin}\ e\ .\ es),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{ebgn2}}$
$\quad \mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{seqk}(es) : \kappa,\ v,\ \rho : \theta)$

Completing the evaluation of the last expression in a sequence:

$$\mathbf{ps}(\mathbf{seqk}(\text{`}()),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{kseq0}} \mathbf{ps}(\phi,\ \rho',\ \sigma,\ \kappa,\ v,\ \theta)$$

Completing the evaluation of an expression in a sequence:

$$\mathbf{ps}(\mathbf{seqk}(e : es),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{kseq1}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{seqk}(es) : \kappa,\ v,\ \theta)$$

Figure 4.8: Evaluating sequences of expressions

Sequences of expressions are expressed either explicitly through a `begin` expression or implicitly as the body of a function. These sequences are modelled as thunks, so they have their own contained environment: variables that are created inside this sequence are not visible outside the sequence. Hence, before starting the evaluation of a sequence, we first save the current environment on the value stack so that it can be restored later on. Evaluating sequences of expressions is done by pushing and popping the $\mathbf{seqk}(es)$ continuation. This continuation holds a list of all expressions that still need to be evaluated. After finishing the evaluation of one item, we pop the continuation, retrieve the next item from the list, push a new **seqk** continuation and then start evaluating the item we just extracted. Once all items have been evaluated, we restore the environment that was initially saved to the value stack.

Function application

Evaluating a function call without any arguments:

$$\mathbf{ps}((rator),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{eapl0}} \mathbf{ps}(rator,\ \rho,\ \sigma,\ \mathbf{ratork}(\theta) : \kappa,\ v,\ \rho : \theta)$$

Evaluating a function call with at least one argument:

$$\mathbf{ps}((rator\ .\ e_1 : ... : e_n),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{eapl1}} \mathbf{ps}(e_n,\ \rho,\ \sigma,\ \mathbf{randk}(rator,\ e_{n-1} : ... : e_1,\ 1) : \kappa,\ v,\ \rho : \theta)$$

Completing the evaluation of the last argument in a function application

$$\mathbf{ps}(\mathbf{randk}(rator,\ \text{`}(),\ i),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{krnd0}} \mathbf{ps}(rator,\ \rho',\ \sigma,\ \mathbf{ratork}(i) : \kappa,\ v,\ \rho : v : \theta)$$

Completing the evaluation of an argument with more arguments still left to evaluate:

$$\mathbf{ps}(\mathbf{randk}(rator,\ e_j : e_{j+1} : ... : e_k,\ i),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{krnd1}} \mathbf{ps}(e_j,\ \rho',\ \sigma, \mathbf{randk}(rator,\ e_{j+1} : ... : e_k,\ i+1) : \kappa,\ v,\ \rho : v : \theta)$$

Completing the evaluation of a user-defined operator in a function application:

$$\mathbf{ps}(\mathbf{ratork}(i),\ \rho,\ \sigma,\ \kappa, \mathbf{clo}(\mathbf{lam}(pars, body), \rho*),\ \rho' : v_1 : ... : v_i : \theta) \xrightarrow{\texttt{krtrc}} \mathbf{ps}((\texttt{begin}\ body),\ \rho*',\ \sigma', \mathbf{aplck}() : \kappa,\ v,\ \rho' : \theta)$$

where $< \rho*', \sigma' >$ equals $\texttt{bindParams}(\texttt{pars},\ \texttt{i}_1 : ... : \texttt{i}_\texttt{i}, \rho*, \sigma)$

Completing the evaluation of a native function in an application:

$\mathbf{ps}(\mathbf{ratork}(i),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : v_1 : ... : v_n : \theta) \xrightarrow{\texttt{krtrn}}$
$\mathbf{ps}(\phi,\ \rho',\ \sigma, \kappa,\ \mathcal{A}(v, v_1 : ... : v_n),\ \theta)$
where $\mathcal{A}(rator,\ rands)$ applies the native function $rator$ on the arguments $rands$

Completing the evaluation of a function's body:

$\mathbf{ps}(\mathbf{aplck}(),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{kapli}}$
$\mathbf{ps}(\phi,\ \rho',\ \sigma,\ \kappa,\ v,\ \theta)$

Figure 4.9: Evaluating function applications

When evaluating function application with one or more arguments, we first reverse the list of arguments, evaluate the first argument of the reversed list, i.e., the last argument of the application, and then push the **randk**, short for *operand*, continuation on the stack. If an argument has been evaluated, the **randk** continuation has been popped and there are still arguments left to evaluate, we save the value of the argument onto the value stack, push a new **randk** continuation and we evaluate the next argument. If all arguments have been evaluated, or there were no arguments to begin with, we push a **ratork**, short for *operator*, continuation on the stack and start evaluating the operator itself.

Recall that we allow users to employ native functions and not only user-defined SLIPT functions. All functions that are not user-defined functions are automatically considered to be native functions by the CESKθ-machine.

When evaluating the application of a user-defined function, we first save the current environment on the value stack and then switch to the lexical environment of the closure. We bind all parameters to their corresponding arguments, located on the stack, via the `bindParams` function. We then push the **aplck** continuation on the stack and evaluate the body of the function. The number of arguments that are to be bound is expressed through the i component of the **ratork** continuation, and was created via the successive pushes and pops of the **randk** continuation. If the number of parameters does not match this i, function application fails. Once the function's body has been executed, the **aplck** continuation is popped. At this point, the environment that was saved to the stack before executing the function call is restored.

The `krtrn` rule determines how applications that use a native function as operator are handled in SLIPT. For brevity, the precise definition of these native functions and how they can be applied is left out of this thesis. We assume that we have some way of calling a native function f on a list of arguments es, expressed as $\mathcal{A}(f, es)$. The result of this application is then moved to the value register.

The auxiliary function $\texttt{bindParams}(\texttt{pars},\ \texttt{args},\ \rho,\ \sigma)$ is defined as:

$\texttt{bindParams}(\text{`}(),\ \text{`}(),\ \rho,\ \sigma)\ =$

$$< \rho,\ \sigma >$$

$$\texttt{bindParams}((\texttt{par} : \texttt{pars}),\ (\texttt{arg} : \texttt{args}),\ \rho,\ \sigma) =$$
$$\texttt{bindParams}(\texttt{pars},\ \texttt{args},\ \rho[\texttt{par} \to \alpha],\ \sigma[\alpha \to \texttt{arg}])$$

$$\texttt{bindParams}(\texttt{par},\ \texttt{args},\ \rho,\ \sigma) =$$
$$< \rho[par \to \alpha],\ \sigma[\alpha \to args] >$$

Figure 4.10: The definition of `bindParams`

We use the symbol α to refer to a new, unique address.

This function takes a list of parameters, another list of arguments, an environment and a store, and returns the new environment and the store in which all arguments have been bound to their corresponding parameter.

If the list of parameters and arguments are both empty, the environment and the store are returned. If the arguments and parameters are both non-empty lists, we bind the first element of the arguments list, arg, to the first element of the parameters list, par. `bindParams` then reduces to itself with the rest of the arguments and parameters. As in Scheme, functions in SLIPT can have a variable number of arguments. These kinds of variable-arity functions have the same semantics as they do in Scheme: all additional arguments are grouped together in the form of a list to the very last parameter of the function. This case is handled through the third rule of `bindParams`: if the set of parameters is a symbol instead of a list, $args$ is bound to this symbol. Since no more arguments are left to bind, the new environment and store are returned.

Apply

Evaluating a call to `apply`:

$$\mathbf{ps}((\texttt{apply}\ rator\ rands),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{eaply}}$$
$$\mathbf{ps}(rands,\ \rho,\ \sigma,\ \mathbf{applyk}(rator) : \kappa,\ v,\ \theta)$$

Completing the evaluation of the arguments:

$$\mathbf{ps}(\mathbf{applyk}(rator),\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_n,\ \theta) \xrightarrow{\texttt{kaply}}$$
$$\mathbf{ps}(rator,\ \rho,\ \sigma,\ \mathbf{ratork}(i) : \kappa,\ v_1 : ... : v_n,\ \rho : v_1 : ... : v_n : \theta)$$

Figure 4.11: Evaluating `apply`-expressions

To evaluate `apply`-expressions, we first push the **applyk** continuation and we then continue with the evaluation of the operands. Once evaluation of $rands$ is completed, we pop the **applyk** continuation, extract the $rator$ expression and start evaluating it. To handle the actual application, we push a **ratork** continuation with the correct number of arguments, i.e., the length of the operands list, onto the stack. From this point on, the function application continues as it was defined previously.

4.4 Low-level instruction set

In this section we present a small instruction set for the CESKθ-machine. The goal of this instruction set is to reduce each of the evaluation rules that were just presented to a set of traceable basic instructions. This set consists of a total of eighteen low-level instructions used for manipulating the registers of the machine. Like the evaluation rules from Section 4.3, these instructions express a transition from one program-state to another, but on a lower level: each instruction generally only updates one component of the program-state. Because tracing the different evaluation steps of SLIPT results in a linear trace of basic instructions, i.e., control-flow is completely removed, we do not define any low-level instructions for manipulating the control component of the machine, i.e., the expression or continuation register.

It is important to understand that these instructions are completely orthogonal to the semantics of SLIPT. They are not used directly to execute a language, but to model the execution of the CESKθ-machine. Conceptually, these instructions can be viewed as a sort of assembler instructions for the CESKθ-machine. By defining these instructions, we can study the workings of the CESKθ-machine on an even lower level of granularity than through the semantics described in the previous section. Whereas the previous section describes the small-step semantics for SLIPT, these instructions express "smaller-step" semantics for the CESKθ-machine.

By carefully selecting the low-level instructions we now introduce, we can redefine the formal semantics of SLIPT that were specified in the previous section. This is exactly what we will do in Section 4.5.

The instruction set is defined as follows:

Allocating a variable from the value register:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{avar(x)}} \mathbf{ps}(e,\ \ \rho[x \to \alpha],\ \ \sigma[\alpha \to v],\ \kappa,\ v,\ \theta)$$

Applying a native function:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ v_1 : \ldots : v_i : \theta) \xrightarrow{\texttt{anat(i)}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \mathcal{A}(v,\ v_1 : \ldots : v_i),\ \theta)$$

Creating a new closure:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{ccls(x, es)}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \mathbf{clo}(\mathbf{lam}(x, es), \rho),\ \theta)$$

Moving a literal to the value register:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{litv(x)}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ x,\ \theta)$$

Looking up a variable:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{lvar(x)}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \sigma(\rho(x)),\ \theta)$$

Popping a continuation from the continuation stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{popk()}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta)$$

Save the environment and move all arguments to the top of the stack to prepare the machine for a function call:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \mathbf{clo}(\mathbf{lam}(x, es), \rho*),\ v_1 : ... : v_i : \theta) \xrightarrow{\texttt{prfc(i)}} \mathbf{ps}(e,\ \rho*,\ \sigma,\ \kappa,\ \mathbf{clo}(\mathbf{lam}(x, es), \rho*),\ v_1 : ... : v_i : \rho : \theta)$$

Pushing a continuation on the continuation stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{pshk}(\phi)} \mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta)$$

Restoring the environment from the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta) \xrightarrow{\texttt{renv()}} \mathbf{ps}(e,\ \rho',\ \sigma,\ \kappa,\ v,\ \theta)$$

Restoring a value from the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ val' : \theta) \xrightarrow{\texttt{rval()}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ val',\ \theta)$$

Restoring the first i values from the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ v_1 : ... : v_i : \theta) \xrightarrow{\texttt{rvls(i)}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_i,\ \theta)$$

Saving all values in the value register one by one to the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_n,\ \theta) \xrightarrow{\texttt{svav()}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_n,\ v_1 : ... : v_n : \theta)$$

Saving the environment to the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{svev()}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \rho : \theta)$$

Saving a single value to the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{svvl()}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ v : \theta)$$

Saving the first i values from the value register one by one to the value stack:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_i : ... : v_n,\ \theta) \xrightarrow{\texttt{svvs(i)}} \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v_{i+1} : ... : v_n,\ \ v_1 : ... : v_i : \theta)$$

Setting the environment:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{stev}(\rho*)} \mathbf{ps}(e,\ \ \rho*,\ \sigma,\ \kappa,\ v,\ \theta)$$

Setting the store:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{stst}(\sigma*)} \mathbf{ps}(e,\ \rho,\ \ \sigma*,\ \kappa,\ v,\ \theta)$$

Assigning a new value to an existing variable:

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{stvr(x)}} \mathbf{ps}(e,\ \rho,\ \ \sigma[\rho(x) \rightarrow v],\ \kappa,\ v,\ \theta)$$

Figure 4.12: The low-level instruction set

Note that every instruction is labelled by a mnemonic that may or may not take a list of parameters. These parameters bind any free variables that occur in the rule. This implies that every low-level instruction can be completely defined by using its label along with the appropriate argument. For example, moving the value `0` to the value register of the program-state is expressed as `litv(0)`.

4.5 Redefining SLIPT's semantics

We can redefine the semantics of SLIPT by using this low-level instruction set. In this section we redefine the semantics for SLIPT that were provided in Section 4.3. Since our instruction set lacks operations for altering the control component, we ignore this component.

Ignoring the fact that the control component remains unchanged by each of these new transition rules, the semantics below are identical to the semantics defined in Section 4.3. By reimplementing the previous evaluation rules with these new instructions, we prove that the instruction set is powerful enough to express non-trivial semantics.

We name the CESKθ-machine that we now define the LLI CESKθ-machine: the low-level instructions CESKθ-machine, to make the difference with the ordinary CESKθ that was presented in Section 4.3.

$$\mathbf{ps}((\texttt{apply}\ rator\ rands),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta)\ :\ pshk(\mathbf{applyk}(rator))$$

$$\mathbf{ps}((\texttt{begin}),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta)\ :$$

$litv(`())$
$popk()$

ps((`begin` e), ρ, σ, κ, v, θ) :
ϵ

ps((`begin` e . es), ρ, σ, κ, v, θ) :
$svev()$
$pshk(\mathbf{seqk}(es))$

ps((`define` ($name$. $pars$) . $body$), ρ, σ, $\phi : \kappa$, v, θ) :
$avar(name)$
$ccls(pars,\ body)$
$stvr(name)$
$popk()$

ps((`define` x e), ρ, σ, κ, v, θ) :
$svev()$
$pshk(\mathbf{definek}(x))$

ps((`if` e $e1$. $e2$), ρ, σ, κ, v, θ) :
$svev()$
$pshk(\mathbf{ifk}(e1,\ e2))$

ps((`lambda` x . es), ρ, σ, $\phi : \kappa$, v, θ) :
$ccls(x,\ es)$
$popk()$

ps((`set!` x e), ρ, σ, κ, v, θ) :
$svev()$
$pshk(\mathbf{setk}(x))$

ps(($rator$), ρ, σ, κ, v, θ) :
$svev()$
$pshk(\mathbf{ratork}(0))$

ps(($rator$. $e_1 : ... : e_n$), ρ, σ, κ, v, θ) :
$svev()$
$pshk(\mathbf{randk}(rator,\ e_2 : ... : e_n,\ 1))$

ps(x, ρ, σ, $\phi : \kappa$, v, θ) :

$lvar(x)$
$popk()$

ps$(val,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta)$:
$litv(e)$
$popk()$

ps$(\mathbf{aplck}(),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta)$:
$renv()$
$popk()$

ps$(\mathbf{applyk}(rator),\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_n,\ \theta)$:
$svav()$
$svev()$
$pshk(\mathbf{ratork}(n))$

ps$(\mathbf{definek}(x),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta)$:
$renv()$
$avar(x)$
$popk()$

ps$(\mathbf{ifk}(e1,\ `()),\ \rho,\ \sigma,\ \phi : \kappa,\ \#\mathtt{f},\ \rho' : \theta)$:
$renv()$
$popk()$
$litv(`())$

ps$(\mathbf{ifk}(e1,\ `()),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta)$:
$renv()$

ps$(\mathbf{ifk}(e1,\ e2),\ \rho,\ \sigma,\ \kappa,\ \#\mathtt{f},\ \rho' : \theta)$:
$renv()$

ps$(\mathbf{ifk}(e1,\ e2),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta)$:
$renv()$

ps$(\mathbf{randk}(rator,\ `(),\ i),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta)$:
$renv()$
$svvl()$
$svev()$
$pshk(\mathbf{ratork}(i))$

$\mathbf{ps}(\mathbf{randk}(rator,\ e_j : e_{j+1} : ... : e_k,\ i),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta)\ :$
$renv()$
$svvl()$
$svev()$
$pshk(\mathbf{randk}(rator,\ (\texttt{cdr}\ rands),\ i+1))$

$\mathbf{ps}(\mathbf{ratork}(i),\ \rho,\ \sigma,\ \kappa, \mathbf{clo}(\mathbf{lam}(pars, body), \rho*),\ \rho' : v_1 : ... : v_i : \theta)\ :$
$renv()$
$prfc(i)$
$stev(\rho*')$
$stst(\sigma')$
$pshk(\mathbf{aplck}())$
where $< \rho*', \sigma' >$ equals $\texttt{bindParams}(pars,\ v_1 : ... : v_i, \rho*, \sigma)$

$\mathbf{ps}(\mathbf{ratork}(i),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : v_1 : ... : v_n : \theta)\ :$
$renv()$
$anat(i)$
$popk()$

$\mathbf{ps}(\mathbf{seqk}(`()),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta)\ :$
$renv()$
$popk()$

$\mathbf{ps}(\mathbf{seqk}(e : es),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta)\ :$
$pshk(\mathbf{seqk}(es))$

$\mathbf{ps}(\mathbf{setk}(x),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : \theta)\ :$
$renv()$
$stvr(x)$
$popk()$

Figure 4.13: The LLI CESKθ-machine

4.6 Implementation

We have implemented both the regular CESKθ-machine as well as LLI CESKθ in Racket. This serves as an experimental validation and as a basis for further research in this thesis. There are some small deviations between the formal semantics and the actual implementation.

To ease further development in SLIPT, the syntax of the implemented inter-

preter has been extended to include some of the various `let`, `let*` and `letrec` constructs, as well as `and`, `or` and `cond`-expressions. Because there is a large semantic overlap between the interpreter and Racket, we did not redefine any of the SLIPT literals, i.e., strings, booleans, numbers and symbols, but we used reflective overlap: SLIPT literals are implemented as their equivalent Racket literal. We also extended the domain of SLIPT literals to include all additional Racket literals, such as characters and regex strings.

All native functions that are defined for SLIPT are implemented as Racket functions. The interpreter automatically makes sure that these Racket functions are integrated into SLIPT. Applying a native function in SLIPT can then be accomplished by applying the underlying Racket function on the corresponding arguments.

Although the interpreter differentiates between user-defined SLIPT functions and native functions, this difference is not visible to the programmer developing SLIPT programs. Native functions can thus safely be used in combination with user-defined functions inside SLIPT programs.

Chapter 5

Tracing semantics

5.1 Introduction

In this chapter, we present a tracing compiler for SLIPT, based on the CESKθ-machine from the previous chapter. We only develop a minimalistic meta-tracing compiler here, which does not contain any non-essential features that are commonly found in other, state-of-the-art tracing compilers. By ignoring these non-essentials, we concentrate on creating a solid foundation for our compiler. In Chapter 6, we build on this foundation and expand our compiler with additional features.

We present our tracing compiler as a formal semantics, similar to the previous chapter, and we express the workings of our compiler as a series of transition rules between *tracer-states*.

The key idea behind our tracing compiler is that we divide the execution of a SLIPT program into three distinct phases: *normal interpretation*, *trace recording* and *trace execution*. The first phase corresponds with an interpreter executing the program and ignoring everything related to tracing entirely. In the second phase, the execution of an interpreter is recorded into a trace. In the third phase, we execute a trace that was previously recorded. We further assign responsibility for each of these phases to two separate entities: a tracer and an interpreter. The interpreter is nothing more than the LLI CESKθ-machine presented in Section 4.5, i.e., the CESKθ-machine that operates on the low-level instruction set. Throughout this chapter the terms interpreter and CESKθ-machine are used interchangeably: they always refer to the same concept. The tracer is a new entity that is defined in this chapter. Together, these two entities compose the tracing compiler. The trace execution phase is the responsibility of the tracer, while normal interpretation is handled by the interpreter. Trace recording is the joint responsibility of both tracer and interpreter.

The tracer is master over the interpreter: it monitors and controls the execution of the interpreter and decides when to transition from one state to another. The interpreter is able to communicate with the tracer through a well-defined,

but flexible, interface. Figure 5.1 shows how the tracer and the interpreter are related.

This interface allows the CESKθ-machine to transfer information about its execution, along with several other important signals, back to the tracer, which then uses this information to capture the execution of the machine at trace-recording time. Note that the tracer does not have any way of checking on the state of the program currently being executed, as this is completely the responsibility of the interpreter. However, the tracer can gather some information on the execution of the program by interacting with the interpreter.

By creating an explicit interface between the two, it becomes possible to switch one CESKθ-machine for another without impacting the tracer. For the purpose of this thesis, we stick to tracing SLIPT programs, but our framework is entirely configurable in the sense that we can plug in whatever interpreter we want, as long as it conforms to our interface. We can even plug in machines that interpret a language completely different from SLIPT. Only two requirements must be satisfied by the interpreter in order for it to be traced: the interpreter must satisfy this interface, and it must be possible to trace its actions, i.e., it must be possible to reify its actions so that they can be inserted into a trace. In this thesis, we only describe the tracing machine in combination with the LLI CESKθ-machine. In this configuration, the second requirement is automatically satisfied because the LLI CESKθ-machine operates using the low-level instruction set. Every action that is performed by the interpreter can therefore be represented as a sequence of low-level instructions.

An introduction to tracing compilation was given in Chapter 2. In Section 5.2, we give an informal overview of how SLIPT programs are traced and what kinds of traces can be produced, using some concrete examples. We slightly extend the syntax of SLIPT in Section 5.3 so it becomes possible to trace the CESKθ-machine. In Section 5.4, we give a formal definition of our tracer. In Section 5.5 we go into more detail on the interface between the tracer and the interpreter, and we explain exactly how they cooperate. Section 5.6 contains some information on how we use the guard instructions (introduced in Chapter 2) in our framework. From Section 5.7 through Section 5.9, we explain the three phases in the execution of a SLIPT program: normal interpretation, trace recording and trace execution respectively.

5.2 Tracing overview

To trace a SLIPT program, the traced program must contain a number of annotations that specify where to start and stop tracing. We specify two annotations in our tracing compiler: `can-start-loop` and `can-close-loop` annotations. `can-start-loop` annotations are used to signal the starting point of traces, `can-close-loop` annotations form the end point. Both annotations are spread throughout the program at locations relevant for tracing. An annotation is always parametrized by a label that identifies the trace that starts or ends at this location. Such a label takes the form of a SLIPT value, such as a

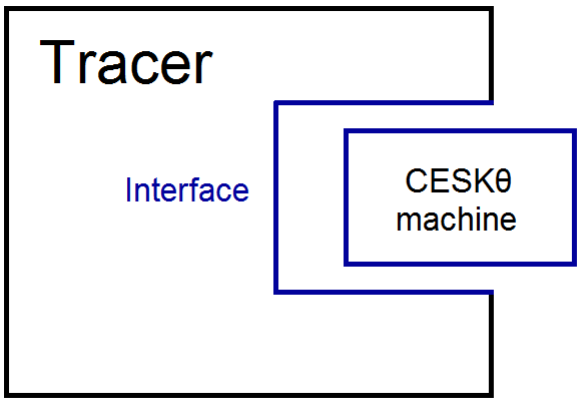


Figure 5.1: The relation between the interpreter and the tracer

boolean, a symbol or even a function. Listing 5.1 shows some example SLIPT code containing both `can-start-loop` and `can-close-loop` annotations.

```
(define (complex-calculation n)
  (can-start-loop 'complex-trace)
  (define a (sqr n))
  (define b (* 2 a))
  (define result (sqrt b))
  (can-close-loop 'complex-trace)
  result)
```

Listing 5.1: A function in SLIPT

The execution of a SLIPT program is divided in three distinct phases: normal interpretation, where the interpreter executes the program without any tracing going on whatsoever, trace recording, where the actions of the interpreter are recorded, and trace execution, where a previously recorded trace is executed. These tracing annotations serve as transition points where the program's execution may shift from one phase to another. Figure 5.2 shows a full state diagram expressing how the three execution phases are linked. Throughout the rest of this section, we explain this diagram by presenting concrete examples of SLIPT programs that use tracing annotations and by explaining how these programs are executed exactly.

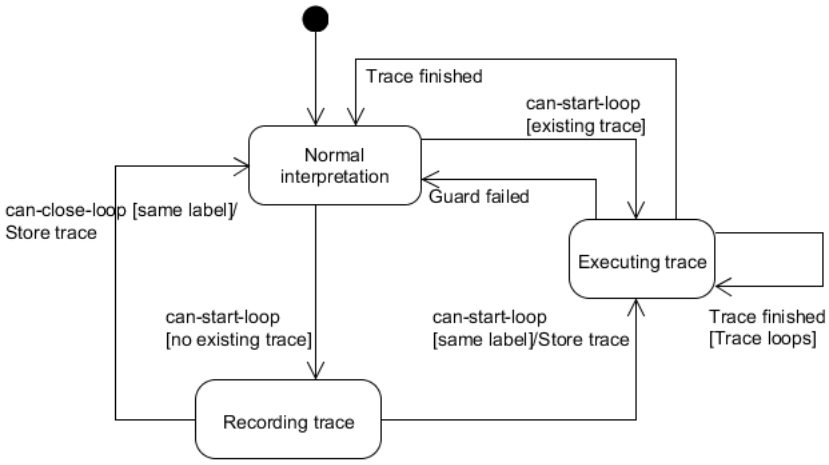


Figure 5.2: Transitioning between the three phases in a SLIPT program's execution

Bounded traces

We first consider the example program shown in Listing 5.1. Since the execution of a program always begins in the normal interpretation phase, the interpreter starts off with the evaluation of this program. When it evaluates the `(can-start-loop 'complex-trace)` expression, it first evaluates the label `'complex-trace` and then checks whether a trace with this label has already been recorded. Since there is no trace yet with the label `(can-start-loop 'complex-trace)`, we transition to the trace recording phase and the tracer starts recording a trace with this label from this point on. The interpreter proceeds as before, but all of the actions it executes during interpretation are now recorded into a trace by the tracer. Once the tracer has started tracing, it continues to do so across all function calls until the interpreter reaches the `(can-close-loop 'complex-trace)` expression. Because the interpreter has now reached a trace end point with a label identical to the label of the trace it is currently recording, tracing stops, we shift back to the normal interpretation phase and the recorded trace is stored under the label `'complex-trace`. The trace is therefore bound by the `can-start-loop` and `can-close-loop` annotations.

The next time the interpreter calls the `complex-calculation` function and encounters the `(can-start-loop 'complex-trace)` annotation, it discovers that a trace for this label already exists and the framework starts executing this trace instead of continuing with normal interpretation, i.e., we move on to the trace execution phase. The body of this function, and hence the contents of this trace, does not contain any apparent potential locations for side-exits.

Recall from Chapter 2 that a trace performs a side-exit when it aborts the execution of a trace before it has reached the end of the trace. This usually occurs when a guard fails. Because there are no apparent locations where a guard could fail in this example, the interpreter is guaranteed to continue executing the trace until the trace has come to an end, at which point we transition back to the normal interpretation phase. The interpreter restarts interpretation from the point in the program corresponding with the end of the trace, namely right after the location of the `(can-close-loop 'complex-trace)` annotation and before the `result` expression.

Looping traces

Listing 5.2 shows an example of a recursive function that contains only one annotation: a `can-start-loop` annotation with the label `'print-number`.

```
(define (do-something)
  (can-start-loop 'do-something)
  (if (= (random 2) 0)
      (displayln 0)
      (displayln 1))
  (do-something))
```

Listing 5.2: A looping function in SLIPT

Once the interpreter evaluates this annotation, it checks whether a trace with the label `'print-number` has already been recorded. If not, we transition to the trace recording phase and the tracer starts tracing from the point of this annotation. Since the body of this function does not contain any `can-close-loop` annotations, tracing continues across the recursive function call until the `can-start-loop` annotation is reached again in the next iteration. Encountering a `can-start-loop` again with the label `'print-number` completes the circle and causes tracing to stop. `can-start-loop` annotations have the implicit condition that they are placed at the start of a loop since we are usually only interested in tracing loops. In this case, the loop takes the form of the recursive function `print-number-loop`. If tracing starts from a `can-start-loop` annotation with some label and later on the interpreter sees another `can-start-loop` annotation with the same label, the interpreter assumes it has completed one full iteration of some loop and that it has encountered the *same* annotation. Since the underlying control-flow loops, the trace itself should also loop. Traces that therefore end at a `can-start-loop` annotation should be restarted once execution has reached their end, in order to accommodate to the implicit looping semantics of the trace. Note that the interpreter cannot make the distinction between two annotations that have the same type and carry the same label, so it always considers them identical, even though they are possibly different.

Side-exits

Listing 5.3 shows an example of a function that does contain possible locations for side-exits in the trace.

```
(define (do-something)
  (can-start-loop 'do-something)
  (if (= (random 2) 0)
      (displayln 0)
      (displayln 1))
  (do-something))
```

Listing 5.3: A function whose trace can contain a side-exit

As mentioned in Chapter 2, the interpreter uses special instructions called *guards* to make sure that the conditions that were present while tracing a program are still valid when executing the trace. Concretely, this means that when tracing `do-something`, the tracer places a guard in the trace at the location corresponding to the if-test. If during tracing of this function `(random 2)` evaluates to `0`, then the tracer inserts a guard into the trace to check that, if the trace is executed, `(random 2)` again evaluates to `0`. If this trace is now re-executed and `(random 2)` evaluates to `1` instead of `0`, a *guard failure* is triggered. This causes the tracer to abandon the execution of the trace, which is called a *side-exit*. Execution of the program then transitions back to normal interpretation, so the interpreter resumes interpretation of the program starting from the point of the guard failure. In this case, the guard failure occurred at the location of the if-test, so interpretation is restarted from just after the if-condition and right before the false-branch is evaluated.

Trace example

In our framework, since we are tracing the actions of the LLI CESKθ-machine from Section 4.5, the trace of a SLIPT program consists of the low-level program-state transition instructions that were introduced in Section 4.4. Execution of a trace is then nothing more than consistently applying each instruction to a certain program-state, and using the resulting output state as the input for the next instruction. In essence, executing a trace is playing back a series of low-level state transitions that have been recorded earlier. As an example, Figure 5.3 shows the program already presented in Listing 5.1 and part of its trace side by side. Concretely, the trace handles the evaluation of the expression `(define a (sqr n))`.

```
(define (complex-calculation n)
  (can-start-loop 'complex-trace)
  (define a (sqr n))
  (define b (* 2 a))
  (define result (sqrt b))
  (can-close-loop 'complex-trace)
  result)
```

Listing (5.4) A function in SLIPT

...
$svev()$
$pshk(\mathbf{definek}(\texttt{a}))$
$svev()$
$pshk(\mathbf{randk}(\texttt{sqr}, \texttt{'()}, 1))$
$lvar(\texttt{n})$
$popk()$
$renv()$
$svvl()$
$svev()$
$pshk(\mathbf{ratork}(1))$
$lvar(\texttt{sqr})$
$popk()$
$renv()$
$anat(1)$
$popk()$
$renv()$
$avar(x)$
$popk()$
...

(a) Part of the trace corresponding with the code from Listing 5.4

Figure 5.3: A SLIPT program that can be traced along with its resulting trace

This trace matches the sequence of instructions that the LLI CESKθ-machine executes if it evaluates the expression `(define a (sqr n))`.

Annotations

It should be noted that although it seems extremely cumbersome for users to have to place annotations at all locations where programmers wish to start or stop tracing, this issue is not so severe in practice. This compiler is developed primarily to serve as a meta-tracing compiler, so the input to the CESKθ-machine are other interpreters. This also means that the people who will be generally writing programs for this framework are the language developers that create these interpreters. These people are therefore the only ones who have to understand how tracing works in our framework and where these annotations have to be placed: the end-users who develop programs on top of the

language interpreters built by the language developers do not need to have this knowledge at all. Furthermore, although we had to use one or two annotations per example program that we have shown in this section, language interpreters built on top of this framework generally only require a handful of annotations at most to function correctly.

5.3 Extended Syntax

To enable the tracing of the LLI CESKθ-machine, we must include `can-close-loop` and `can-start-loop` annotations in our SLIPT programs. We therefore extend the syntax of SLIPT with these tracing annotations. Since these are the only expressions we have to add to our syntax, extending the syntax is straightforward. We extend our definition for *Exp* to include these new annotations. Figure 5.4 shows the updated definition of *Exp*.

$$
\begin{aligned}
e \in \mathit{Exp} = \ & \ldots \\
& \mid (\texttt{can-close-loop}\ \mathit{Val}) \\
& \mid (\texttt{can-start-loop}\ \mathit{Val})
\end{aligned}
$$

Figure 5.4: The new definition for *Exp*

5.4 Tracing machine

In this section, we formally define the tracing component of our framework. We also call this the *tracing machine*, or simply *tracer*, in contrast with the LLI CESKθ-machine which serves as the interpreter in our framework.

$$
\begin{aligned}
TracerState &= \mathbf{ts}(ExecutionPhase,\ TracerContext,\ ProgramState,\ False + TraceNode) \\
ExecutionPhase &= TE \\
&\mid NI \\
&\mid TR \\
\tau \in Trace &= Instruction : Trace \\
&\mid \epsilon \\
tc \in TracerContext &= \mathbf{tc}(False + TraceNode,\ TNs) \\
val \in TraceKey &= Val \\
tn \in TraceNode &= \mathbf{tn}(TraceKey,\ Trace) \\
TNs &= TraceNode : TNs \\
&\mid \epsilon
\end{aligned}
$$

Figure 5.5: The tracing machine

Execution of a SLIPT program is defined as transitioning between the phases of normal interpretation, trace recording and trace execution. Execution can therefore be represented as a state machine.

TracerState Because a program's execution is controlled by the tracing machine, we define tracer-states that capture the state of the tracing machine. Such a state is a four-tuple consisting of an evaluator-state keyword, a tracer context, a program-state, and a trace node, if any trace is being executed.

ExecutionPhase The $ExecutionPhase$ is a keyword that indicates in which phase the tracing machine is working. It always takes the form of one of three values: NI when the tracing machine is in the **n**ormal **i**nterpretation phase, TR when the tracer is in the **t**race **r**ecording phase and TE when the tracer is in the **t**race **e**xecuting phase.

TracerContext The tracer context is a structure used to store various information relevant to the tracer, such as the trace that is currently being captured or recorded, as well as a list of all traces that have already been recorded. To implement the tracer-context, we must first define a trace-node auxiliary construct. A trace-node is a structure that associates a trace to a trace-key. The trace-key is the label that is given to the trace, as explained in Section 5.2. The tracer context consists of a list of trace-nodes, the traces that have already been recorded previously, as well as a single trace-node that represents the trace that is currently being captured. If no trace is being recorded, this field of the tracer context is set to $False$.

ProgramState The third component of a tracer-state is its program-state. This is the state on which both the CESKθ-machine and the instructions in the trace operate. So if the the tracing machine is either performing normal interpretation of its input-program or is recording a trace, it constantly feeds this program-state as input to the CESKθ-machine, captures the resulting output-state of the interpreter and swaps its old program-state for this new state. This program-state is also required when the tracer is executing a trace, since it serves as the input to each instruction that was recorded into the trace.

TraceNode The final field of the tracer-state is either the trace-node that is currently being executed, i.e., a structure containing the trace that is being executed along with the label of this trace, or it is False if no trace is being executed at the moment.

A trace consists of a series of the low-level instructions that were presented in Section 4.4. Figure 5.3 of Section 5.2 shows an example trace.

5.5 Interface

We now define the interface between the LLI CESKθ-machine and the tracer. This interface is extremely important, because it is the bridge between the tracer and the interpreter. Before we formally define the workings of the tracing machine, we first define the interface through which the CESKθ-machine interacts with the tracer, which allows us to express more clearly the actual mechanics employed by the tracing framework. The interface is expressed in Figure 5.6.

$$
\begin{aligned}
\mathit{CesktReturn} &= \mathbf{cesktStep}(\mathit{ProgramState},\ \mathit{Trace},\ \mathit{AnnotationSignal}) \\
&\mid \mathbf{cesktEvent}(\mathit{EventSignal}) \\
\mathit{EventSignal} &= \mathit{CesktStopped} \\
\mathit{AnnotationSignal} &= \mathbf{CCL}(\mathit{Val}) \\
&\mid \mathbf{CSL}(\mathit{Val}) \\
&\mid \mathit{False}
\end{aligned}
$$

Figure 5.6: The interface between the CESKθ-machine and the tracer

The relationship between the tracer and the interpreter is defined as a master-slave relation, where the tracer is the master and controls the execution of the interpreter. Whenever the tracer asks it to, the interpreter performs a single transition rule, i.e., a single *step*, on a given input-state.

CesktReturn Until now, a transition would take a certain program-state and return a new program-state. This is no longer satisfactory because we must now trace the actions of the interpreter, so we require the interpreter to piggy-back certain signals about its evaluation of the program to the tracer. We therefore wrap the return values of each transition of the LLI CESKθ-machine in an extended structure, allowing us to return not only the new program-state that resulted from this transition but also additional information that may or may not be relevant to the tracer. For example, if the CESKθ-machine has encountered a tracing annotation, it should have a way to signal this back to the tracer, so that the tracing machine may decide to start tracing, or to start executing a previously recorded trace. Each CESKθ transition therefore returns a $CesktReturn$ structure, which contains a program-state, an annotation signal, and a series of instructions, i.e., a *trace*.

ProgramState The $ProgramState$ is the program-state that resulted from applying this transition on the given program-state.

AnnotationSignal The CESKθ-machine express to the tracer that a tracing annotation has been encountered by using the correct $AnnotationSignal$. Since we have included two annotations in SLIPT, we must define two annotation signals: **CCL** for `can-close-loop` annotations and **CSL** for `can-start-loop` annotations. These signals carry the label used by the annotation. Recall that this label is a SLIPT value, so we define the label used in the signal to be a Val. If the CESKθ has not encountered any tracing annotation, it returns $False$ instead of an annotation signal.

Trace The CESKθ-machine must return the sequence of actions it has performed in that transition. These actions are the low-level instructions used in each transition of the LLI CESKθ-machine. Because each transition can be described in terms of a handful of low-level instructions that are applied on a program-state, we can trace the execution of the interpreter by recording the sequence of low-level instructions that correspond with the transition rules that the interpreter applies.

Events To add flexibility to this interface, we define *event returns*. These are special return values that can be used by the CESKθ-machine in case of special events, such as when the evaluation of the input program has been completed.

Interface example

In order for the CESKθ-machine to comply to this new interface, we have to wrap the result of all transitions that it applies. We do not redefine the full semantics, but in Table 5.1 we show two examples of how the new transition rules look in comparison with the old rules.

Old transition rule	Wrapped transition
$\mathbf{ps}((\texttt{begin}),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{ebgn0}}$ $\mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ `(),\ \theta)$	$\mathbf{ps}((\texttt{begin}),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta) \xrightarrow{\texttt{ebgn0}'}$ $\mathbf{cesktStep}(\mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ `(),\ \theta),$ $\{litv(`()),\ popk()\},$ $False)$
$\mathbf{ps}(\mathbf{seqk}(e : es),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{kseq1}}$ $\mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{seqk}(es) : \kappa,\ v,\ \theta)$	$\mathbf{ps}(\mathbf{seqk}(e : es),\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{kseq1}'}$ $\mathbf{cesktStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \mathbf{seqk}(es) : \kappa,\ v,\ \theta)$ $\{pshk(\mathbf{seqk}(es))\},$ $False)$

Table 5.1: The old transition rules compared to the new, wrapped rules

5.6 Guard instructions

5.6.1 Introduction

In this section we explain how guards are handled by our framework. Guard instructions are used by the tracer to ensure that certain conditions that were valid while a trace was being recorded still hold when the trace is executed. If the guard fails when the trace is executed, execution of the trace must be aborted. We specify four kinds of guard instructions in our framework. We provide a formal definitions of these guards in this section.

5.6.2 Guard examples

A trace is essentially a recording of a linear control-flow: the path through the program that was taken while the trace was being recorded. As the following three examples demonstrate, it is however possible that this control-flow is no longer valid when the trace is executed.

Ordinary guards

Consider the following example, which we have already seen in Section 5.2.

```
(define (do-something)
  (can-start-loop 'do-something)
  (if (= (random 2) 0)
      (displayln 0)
      (displayln 1))
  (do-something))
```

Listing 5.5: A function where control-flow may diverge

Suppose that when this function was traced, `(random 2)` evaluated to `0`. Then the trace contains code to execute the true-branch, and only the true-branch, of this if-expression, i.e., `(displayln 0)`. When executing the recorded

trace, the expression `(random 2)` must still be re-evaluated. However, it is possible that this expression evaluates to `1` this time, which means that the false-branch must be evaluated. We can say that the control-flow of the program diverges at trace-execution time with respect to the control-flow at trace-recording time. To protect ourselves from these divergences, we introduce *guard instructions* into the trace while we are recording it. When the trace is executed, these guard instructions take a program-state as input, check on the conditions that ensure identical control-flow and when these conditions differ from the ones that were present while recording the trace, the guard signals an error and execution of the trace is aborted. Concretely, in the previous example we would introduce a guard that checks whether the expression `(= (random 2) 0)` evaluates to `#t` when executing the trace. If it does, execution of the trace continues normally, else the guard fails and execution of the trace is aborted. The tracing machine then switches back to normal interpretation of the program, starting from the point of the guard failure.

Guards for higher-order functions

Listing 5.5 is not the only example of a program where guards must be introduced to safeguard the validity of the control-flow. Because SLIPT allows for the creation of higher-order functions, we must be careful when tracing function applications. Consider the program in Listing 5.6.

```
(define (g)
  (displayln "g was called"))

(define (h)
  (displayln "h was called"))

(define (do-something f)
  (can-start-loop 'do-something)
  (displayln (f))
  (do-something h))

(do-something g)
```

Listing 5.6: Control-flow diverging because of higher-order functions

In this program, we are using a higher-order function `do-something` that takes a parameter `f` referring a function. When we first call `do-something`, `f` refers to the function `g` but after the first iteration, `f` changes to the function `h`. If the first iteration is traced, we must prevent the tracing machine from executing the `g` function in every iteration of the trace. This is done by inserting a guard. The guard checks the identity of the function being called, before actually executing the function.

Guards for `apply`-expressions

A more subtle example of when guards are necessary is shown in Listing 5.7.

```
(define (g a b)
  (+ a b))

(define (do-something-else args)
  (can-start-loop 'do-something)
  (apply g args)
  (do-something-else '(1)))

(do-something-else '(1 2))
```

Listing 5.7: A more subtle example of code where guards must be introduced

In this program we have a function `do-something-else` in which we apply the function `g` to some list of arguments. In the first iteration, the arguments list is the list `'(1 2)` but in subsequent iterations this list changes to `'(1)`. The problem here is that we are using `apply` to call `g` on some arguments, instead of calling `g` directly on these arguments. As the example shows, it is possible that the number of arguments changes between iterations of the loop. In other words, the number of arguments used is dynamic instead of static.

If the first iteration of this function is traced, the tracing machine inserts instructions into the trace which correspond with evaluating the expression `args`, which evaluates to `'(1 2)`, pushing these values on the value stack *and then calling `g` on the first two values of the value stack*. In other words, the trace contains instructions for popping the first two values from the stack, even though in subsequent iterations, the list of arguments only contains one element. Applying `g` on a list with only one element must cause a runtime error but in practice, the tracing machine calls `g` with the value `1` *and whatever other value lies below* `1` *on the value stack*.

If we were to replace `(apply g args)` by `(g arg-1 arg-2)`, we would not encounter this issue, because it is statically determined that `g` is called on two arguments. Although the value of these two arguments may change between trace-recording and trace-execution time, the *number of arguments* does not change. We could solve this problem by adding a guard to the trace that checks the number of arguments used in an `apply`-expression.

5.6.3 Low-level instruction interface

Low-level instructions take a program-state and return another program-state. In principle, guard instructions are no different from any other instructions: they are inserted in a trace and take some program-state as input. Guards should therefore also share the signature $ProgramState \to ProgramState$. However, such a signature makes it impossible to introduce the guards that are necessary in our framework because guards must have some kind of way of signalling to the tracing machine that the condition they are guarding is invalidated. This problem is similar to a problem we encountered earlier. Similar to low-level instructions, the transition rules that are employed by the CESKθ-machines used to take a program-state as input and return a new program-state. This made it impossible for the interpreter to signal back additional

information on the program's evaluation, such as when it has encountered a tracing annotation. We solved this problem by defining an interface between the tracer and the interpreter: all transitions that are applied by the CESKθ-machine must be wrapped, so that the interpreter can piggy-back additional information to the tracing machine.

We can use the same solution now as we used then: we wrap the low-level instruction so that it can return more than just the resulting program-state. This defines an interface between the low-level instructions and the tracing machine. An extra requirement for for tracing an interpreter in our framework now becomes apparent: the interpreter itself must satisfy the interface between it and the tracing machine, but its actions that are placed into the trace during the recording of the trace must also satisfy some interface.

In Figure 5.7 we present this interface that is used between the low-level instructions and the tracing machine.

$$\begin{aligned} \mathit{InstructionReturn} &= \mathbf{lliStep}(\mathit{ProgramState}) \\ &\mid \mathbf{lliEvent}(\mathit{EventSignal}) \\ \mathit{EventSignal} &= \mathbf{guardFailed}(\mathit{Control}) \end{aligned}$$

Figure 5.7: The interface between the low-level instructions and the tracer

InstructionReturn An instruction always returns an $\mathit{InstructionReturn}$ when it is applied, which can be either a **lliStep** or an **lliEvent**.

lliEvent To provide flexibility to the interface, we use **lliEvent** structures. These structures can not only be used to signal guard failures, but can also be used in the future, if we want to make instructions signal back other kinds of information.

lliStep Normal, non-guard, instructions, always return a **lliStep** containing the program-state resulting from applying this instruction on some program-state.

Guards Guard instructions can return either of both $\mathit{InstructionReturns}$. If a guard does not fail, it returns a **lliStep** with the program-state it was given as input, i.e., it does nothing. If the guard fails, it returns an **lliEvent** with a **guardFailed** as signal.

Guard's control

A **guardFailed** not only signals the fact that a guard failed, but it also carries back additional information. Recall that guard failures are essentially associ-

ated to some location in the code, namely the location of where some condition needs to be checked. If this condition is invalidated when executing the trace, the tracing machine must abort trace execution and switch back to normal interpretation of the code. However, interpretation should restart from the point in the code where the condition is checked. In our CESKθ-machine, points in the program can be referred to through the control component of the program-state, which, as stated in Section 4.3, is either an expression or a continuation. We therefore make guards, if they fail, signal back not only the fact that they failed, but also the point in the program from where normal interpretation must restart.

5.6.4 Guard instructions

We now present which guard instructions we apply in our framework and how they are used. Afterwards, we redefine some of the transition rules of Section 4.5, namely those rules where we have to insert one of these guards. We define four kinds of guards: `guard-false`, `guard-true`, `guard-same-closure` and `guard-same-nr-of-args`. We also label each guard with a mnemonic.

Guard-false

`guard-false` is defined as follows.

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \#\mathtt{f},\ \theta) \xrightarrow{\mathtt{gfls(c)}} \mathbf{lliStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \#\mathtt{f},\ \theta))$$

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\mathtt{gfls(c)}} \mathbf{lliEvent}(\mathbf{guardFailed}(c))$$

Figure 5.8: The definition of `guard-false`

This kind of guard checks that some condition evaluates to `#f`. It is parametrized with the argument c, which represents the location in the code to which the condition that is checked by this guard corresponds. The guard checks the contents of the value register. If this register contains the value `#f`, it does nothing. If it does not contain this value, the guard fails. It signals back this fact by returning a **guardFailed** containing the point in the program from which interpretation should restart. This point can either be an expression or a continuation.

Guard-True

`guard-true` is identical to `guard-false`, except that we now return an **lliEvent** if the value register *does* contain the value `#f`.

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \#\mathtt{f},\ \theta) \xrightarrow{\mathtt{gtru(c)}} \mathbf{lliEvent}(\mathbf{guardFailed}(c))$$

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\mathtt{gtru(c)}} \mathbf{lliStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta))$$

Figure 5.9: The definition of `guard-true`

Guard-same-closure

`guard-same-closure` checks whether the same function is applied when executing the trace as when recording it. Recall that there are two kinds of functions in SLIPT. The first kind of functions are user-defined functions, which are represented as **clo** structures containing an environment, the lexical environment of the function, and a **lam** structure, which contains a list of parameters and a list of expressions, i.e., its body. The other kind of functions are native functions. For two functions to be the same they must first be of the same type. If they are both user-defined functions, their list of parameters and their bodies must be identical. Checking the identity of two native function lies outside the scope of this thesis. `guard-same-closure` is parametrized with two arguments v and c. v represents the function that was applied while recording the trace. This is then the function that should also be applied when executing the trace. As with the previous two guards, c is again the point in the program from where interpretation should restart if the guard fails. If the functions are the same, `guard-same-closure` does nothing, otherwise it signals this guard failure.

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\mathtt{gscl(v,\ c)}} \mathbf{lliStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta))$$

$$\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v',\ \theta) \xrightarrow{\mathtt{gscl(v,\ c)}} \mathbf{lliEvent}(\mathbf{guardFailed}(c))$$

Figure 5.10: The definition of `guard-same-closure`

Guard-same-nr-of-args

The final guard, `guard-same-nr-of-args`, checks whether the length of the argument list in a call to `apply` during the recording of the trace is equal the length of this list while executing the trace. To perform this check, the expected length, i.e., the length of the list used when recording the trace, is passed as an

argument to the guard. Again, the location in the code from where interpretation should restart if the guard fails is also passed to the guard. The list of arguments is located in the value register of the program-state.

$$\begin{array}{l}\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ val_1 : ... : val_n,\ \theta) \xrightarrow{\texttt{gsna(n, c)}} \\ \quad \mathbf{lliStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ val_1 : ... : val_n,\ \theta))\end{array}$$

$$\begin{array}{l}\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ val_1 : ... : val_k,\ \theta) \xrightarrow{\texttt{gsna(n, c)}} \\ \quad \mathbf{lliEvent}(\mathbf{guardFailed}(c)) \\ \quad \text{with } k \text{ not equal to } n\end{array}$$

Figure 5.11: The definition of `guard-same-nr-of-args`

5.6.5 Adding guards to existing transition rules

We update the definitions of the transition rules that are used by the LLI CESKθ-machine and which determine the control-flow of the program, to reflect the fact that a guard must be placed in the trace when executing these transitions. By placing a guard in these transitions, we make sure that the control-flow that was used when recording the trace is not changed when executing the trace.

If-expressions

The first transition rules we need to alter are the rules that express how an if-expression is evaluated. Concretely, we must change the rules that govern the execution of the if-expression *after* the if-condition has already been evaluated, because we need to know what the value of the if-condition is in order to know which control-flow branch will be taken. The evaluation of an if-expression whose condition has just been evaluated is divided over four rules, depending on the value of the condition and whether an alternative false-branch was provided or not.

$$\begin{array}{l}\mathbf{ps}(\mathbf{ifk}(e1,\ \text{`}()),\ \rho,\ \sigma,\ \phi : \kappa,\ \#\texttt{f},\ \rho' : \theta)\ : \\ \quad renv() \\ \quad gfls(e_1) \\ \quad popk() \\ \quad litv(\text{`}())\end{array}$$

$$\begin{array}{l}\mathbf{ps}(\mathbf{ifk}(e1,\ \text{`}()),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta)\ : \\ \quad renv() \\ \quad gtru(\text{`}())\end{array}$$

$$\mathbf{ps}(\mathbf{ifk}(e1,\ e2),\ \rho,\ \sigma,\ \kappa,\ \#\texttt{f},\ \rho' : \theta)\ :$$

$$
\begin{array}{l}
\quad renv() \\
\quad gfls(e_1) \\
\\
\mathbf{ps}(\mathbf{ifk}(e1,\ e2),\ \rho,\ \sigma,\ \kappa,\ v,\ \rho' : \theta)\ : \\
\quad renv() \\
\quad gtru(e_2)
\end{array}
$$

Figure 5.12: The new transition rules for evaluating an if-expression

For the first rule, where we consider the case that some condition was false but that no false-branch was provided, the if-expression just evaluates to `'()`, i.e., we move the value `'()` to the value register. In order to check that this condition still evaluated to `#f` during the execution of the trace, we insert a `guard-false` instruction into the trace, right before we pop the continuation from the continuation stack. The location of the code to where interpretation must jump should the guard fail is the expression that must be evaluated if the if-condition did not evaluate to `#f`, namely the true-branch of the if-expression. The three other rules are similar to the first: they each state that the if-condition evaluated to some value and that the corresponding branch was taken. For these rules, we only have to place a guard that corresponds with the evaluated condition, i.e., place a `guard-false` branch when the condition evaluated to `#f` and a `guard-true` branch when it did not evaluate to `#f`, and pass to this guard the branch of the if-expression other than the one that has to be evaluated. For the second rule, where the CESKθ-machine takes the true-branch but where no false-branch is provided, we pass the value `'()` to the guard. If that guard fails, the if-expression evaluates to `'()`, which is exactly in line with the expected semantics.

Function application

The second set of rules that need to be changed are the rules dealing with function application, in order to catch issues that arise when dealing with higher-order functions. We have defined two rules for function application: one rule that expresses how user-defined SLIPT functions are applied and another to states how native functions are applied. For both rules, we insert a `guard-same-closure` instruction. In both cases, we pass to this guard the function that is being applied and a $\mathbf{ratork}(i)$ continuation, where i is the length of the argument list. If the closure guard fails, it jumps back to this continuation and applies the correct function. Since this continuation needs to know the number of arguments used, we pass this number to the continuation.

$$
\begin{array}{l}
\mathbf{ps}(\mathbf{ratork}(i),\ \rho,\ \sigma,\ \kappa, \mathbf{clo}(\mathbf{lam}(pars, body), \rho*),\ \rho' : v_1 : ... : v_i : \theta)\ : \\
\quad gscl(\mathbf{clo}(\mathbf{lam}(pars, body), \rho*),\ \mathbf{ratork}(i)) \\
\quad renv()
\end{array}
$$

$prfc(i)$
$stev(\rho*')$
$stst(\sigma')$
$pshk(\mathbf{aplck}())$
where $< \rho*', \sigma' >$ equals `bindParams`$(pars,\ v_1 : ... : v_i, \rho*, \sigma)$

$\mathbf{ps}(\mathbf{ratork}(i),\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \rho' : v_1 : ... : v_n : \theta)\ :$
$gscl(v,\ \mathbf{ratork}(i))$
$renv()$
$anat(i)$
$popk()$

Figure 5.13: The new transition rules for evaluating a function application

`Apply`-expressions

The final rule that needs to be altered is the rule that handles expressions of the form `(apply rator rands)`. We need to insert a `guard-same-nr-of-args` guard here that checks the number of arguments. We pass to this guard two arguments: the expected number of arguments and the continuation from which interpretation should restart.

$\mathbf{ps}(\mathbf{applyk}(rator),\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_n,\ \theta)\ :$
$svav()$
$svev()$
$gsna(n,\ \mathbf{applyFailedk}(rator,\ n))$

Figure 5.14: The new transition rules for evaluating an `apply`-expression

The continuation that we pass, $\mathbf{applyFailedk}(rator,\ n)$ is a special continuation that we have not previously defined yet. Its definition is shown in Figure 5.15.

$\mathbf{ps}(\mathbf{applyFailedk}(rator,\ i),\ \rho,\ \sigma,\ \kappa,\ v_1 : ... : v_n,\ \theta) \xrightarrow{\texttt{kaplf}}$
$\mathbf{ps}(rator,\ \rho,\ \sigma,\ \mathbf{ratork}(i) : \kappa,\ v_1 : ... : v_n,\ \rho : v_1 : ... : v_n : \theta)$

Figure 5.15: Handling an $\mathbf{applyFailedk}(rator,\ i)$ continuation

This continuation takes an operator and the length of some argument list as input and sets everything up so that this operator can be applied to the ar-

gument list. It does this by making sure the value stack is properly set up, pushing the correct continuation onto the continuation stack and then evaluating the operator.

5.7 Normal interpretation

We now define the semantics of the tracer. It is these semantics that determine how the tracing compiler works: how traces are recorded and executed, how normal interpretation proceeds and how guard failures are handled. The execution of a program in our tracing framework can be divided into three distinct phases: normal interpretation, tracing and trace execution. The interpretation phase corresponds to the phase where the CESKθ-machine is interpreting the program and where tracing does not come into play at all. The second phase starts when the interpreter encounters a `can-start-loop` annotation, causing the tracer to start tracing the actions of the CESKθ-machine. In the final phase, the tracer is executing a previously recorded trace. The formal semantics of the tracer distinguish between these three phases. In the following three sections, we present the formal semantics of the tracer that correspond with these three phases. It is important to recall that the tracer itself never checks on the program-state it is using. In other words, it has no way of knowing when a specific annotation is encountered. It is the responsibility of the interpreter to check for these annotations and to signal the correct response, i.e., either one of the $AnnotationSignals$ or $False$ back to the tracer when necessary.

The interpretation phase in the program execution is the most straightforward one to describe, since it corresponds almost entirely to the execution of the LLI CESKθ-machine, which was already described in 4.5. We write $\texttt{step}(\varsigma)$ to indicate that the interpreter applies a single transition rule on the program-state ς.

$$\begin{aligned} &\mathbf{ts}(NI,\ tc,\ \varsigma,\ False) \rightarrow \\ &\quad \mathbf{ts}(NI,\ tc,\ \varsigma',\ False) \\ &\quad \text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ False) \end{aligned} \tag{5.1}$$

Figure 5.16: Normal interpretation if no annotation is encountered

Rule 5.1 represents the most common case: the CESKθ-machine does not encounter any tracing annotation. It therefore returns $False$ instead of a signal, along with the new program-state and the set of low-level instructions it has applied. Although the tracing machine is running in the normal interpretation phase and it hence has no use for these instructions, the LLI CESKθ-machine does not know this and it would indeed be undesirable for the interpreter to be aware of this fact, as it would break the single responsibility principle of programming. It is only the task of the tracer to understand that while execution is running in the interpretation phase, the instructions that are returned by the interpreter through the **cesktStep** struct are generally useless while in this

phase and should be discarded. 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 CESKθ-machine.

$$\begin{aligned}&\mathbf{ts}(NI,\ tc,\ \varsigma,\ False) \rightarrow \\ &\quad \mathbf{ts}(NI,\ tc,\ \varsigma',\ False) \\ &\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathbf{CCL}(val))\end{aligned} \tag{5.2}$$

Figure 5.17: Normal interpretation if a `can-close-loop` annotation is encountered

In rule 5.2, the CESKθ-machine signals that it has encountered a `can-close-loop` annotation. Since we are not tracing currently, we again do not have to take any significant actions. We only swap the old program-state for the new program-state.

$$\begin{aligned}&\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs),\ \varsigma,\ False) \rightarrow \\ &\quad \mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(val,\ \iota_1 : ... : \iota_n),\ TNs),\ \varsigma',\ False) \\ &\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CSL}(val)) \\ &\quad \text{and if no trace for } val \text{ has been recorded yet}\end{aligned} \tag{5.3}$$

$$\begin{aligned}&\mathbf{ts}(NI,\ tc,\ \varsigma,\ False) \rightarrow \\ &\quad \mathbf{ts}(TE,\ tc,\ \varsigma',\ \mathbf{tn}(val,\ \tau)) \\ &\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CSL}(val)) \\ &\quad \text{and where } \tau \text{ is the trace that has previously been recorded for } val\end{aligned} \tag{5.4}$$

Figure 5.18: Normal interpretation if a `can-start-loop` annotation is encountered

A more interesting case arises in rules 5.3 and 5.4, when the CESKθ-machine encounters a `can-start-loop` annotation with label val. In rule 5.3, we assume that no trace has been recorded yet for the label val, so the tracer decides to start tracing this label. It switches its execution-phase and updates its tracer-context to indicate that it is now tracing. The tracer-context is updated by replacing the field representing its current trace. This field now becomes a trace-node consisting of the label that is traced, as well as the first few instructions that have just been executed by the interpreter and that were carried back in the **cesktStep**. These instructions become the very first part of the trace. Note that the program-state of the tracer-state must also be updated as this program-state continues to be used by the interpreter as its actions are being traced.

In rule 5.4, we assume the same conditions as in the third rule, except that the tracer-context now does contain an already-recorded trace for the label val. In this case, the tracer must start executing this trace, so it switches its execution-phase to TE and updates the fourth field in the tracer-state. This field is switched to the tracer-node containing the previously recorded trace

for the label val. We again also have to update the program-state because this state now serves as the input to all instructions that were recorded in the trace τ.

5.8 Trace recording

We present the formal semantics that govern the execution of the tracing machine when it is in the trace recording phase.

$$\begin{aligned}&\mathbf{ts}(\mathit{TR},\ \mathbf{tc}(\mathbf{tn}(\mathit{val},\ \tau),\ \mathit{TNs}),\ \varsigma,\ \mathit{False}) \rightarrow \\ &\quad \mathbf{ts}(\mathit{TR},\ \mathbf{tc}(\mathbf{tn}(\mathit{val},\ \tau : \iota_1 : ... : \iota_n),\ \mathit{TNs}),\ \varsigma',\ \mathit{False}) \\ &\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathit{False})\end{aligned} \tag{5.5}$$

Figure 5.19: Trace recording if no annotation is encountered

Rule 5.5 of these semantics is similar to the rule 5.1 of the semantics presented in the previous chapter, except that the tracing machine is now recording a trace: the interpreter has not encountered any annotation signal. In this case, we again update the program-state of the tracer to feed it to the interpreter in the next step, but we also append the instructions that were executed by the interpreter to the back of the trace we are already recording. We append the instructions to the back of the trace because we are recording the instructions in chronological order: the first recorded instruction should come at the start of the trace.

$$\begin{aligned}&\mathbf{ts}(\mathit{TR},\ \mathbf{tc}(\mathbf{tn}(\mathit{val},\ \tau),\ \mathit{TNs}),\ \varsigma,\ \mathit{False}) \rightarrow \\ &\quad \mathbf{ts}(\mathit{TR},\ \mathbf{tc}(\mathbf{tn}(\mathit{val},\ \tau : \iota_1 : ... : \iota_n),\ \mathit{TNs}),\ \varsigma',\ \mathit{False}) \\ &\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CCL}(\mathit{val}'))\end{aligned} \tag{5.6}$$

$$\begin{aligned}&\mathbf{ts}(\mathit{TR},\ \mathbf{tc}(\mathbf{tn}(\mathit{val},\ \tau),\ \mathit{TNs}),\ \varsigma,\ \mathit{False}) \rightarrow \\ &\quad \mathbf{ts}(\mathit{TR},\ \mathbf{tc}(\mathbf{tn}(\mathit{val},\ \tau : \iota_1 : ... : \iota_n),\ \mathit{TNs}),\ \varsigma',\ \mathit{False}) \\ &\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CSL}(\mathit{val}'))\end{aligned} \tag{5.7}$$

Figure 5.20: Trace recording if an annotation with a different label is encountered

In rule 5.6, the interpreter has encountered a `can-close-loop` annotation but with a label different from the label we are currently tracing. Because the label is different, the annotation has no impact on the behavior of the tracer: it continues tracing. As with the first rule, we do have to update both the program-state and the trace currently being recorded. A tracing annotation that uses another label than the one being traced is handled no different than any other expression by the tracer.

Rule 5.7 is similar to rule 5.6, except that the interpreter has now come across a `can-start-loop` annotation.

$$\begin{aligned}&\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(val,\ \tau),\ TNs),\ \varsigma,\ False) \rightarrow \\ &\quad \mathbf{ts}(NI,\ \mathbf{tc}(False,\ \mathbf{tn}(val,\ \tau) : TNs),\ \varsigma',\ False) \\ &\quad \text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CCL}(val))\end{aligned} \tag{5.8}$$

$$\begin{aligned}&\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(val,\ \tau),\ TNs),\ \varsigma,\ False) \rightarrow \\ &\quad \mathbf{ts}(TE,\ \mathbf{tc}(False,\ tn : TNs),\ \varsigma',\ tn) \\ &\quad \text{where } tn \text{ equals } \mathbf{tn}(val,\ \tau : \iota_1 : ... : \iota_n : \lambda) \\ &\quad \text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CSL}(val))\end{aligned} \tag{5.9}$$

Figure 5.21: Trace recording if an annotation with the label that is being traced is encountered

In rule 5.8, the interpreter again sees a `can-close-loop` annotation, but this time, the annotation does use the same label as that of the trace already being recorded. Since `can-close-loop` annotations form the end points of traces, encountering such an annotation means that the recording of the trace should stop. This is exactly what is done in this rule: the tracer stops recording the trace, constructs a trace-node consisting of the trace and its label, and adds this node to the other trace-nodes so that it can be executed at a later point in time. Furthermore, the tracer must now start interpreting the program again, so the execution-phase is switched. As always, we also update the program-state.

Rule 5.9 is somewhat similar to rule 5.8 in that the interpreter again encounters an annotation carrying the same label as that of the trace currently being recorded. This time however, the annotation is a `can-start-loop` annotation. We again have to stop tracing and add the trace to the list of other trace-nodes. Before we store the trace away however, we first add a special looping-instruction λ to the end of the trace. This instruction serves to tell the tracer that this trace *loops*: it started from a certain start point and came full circle when it again reached a trace start point with the same label, implying that it looped back to the beginning. When the interpreter executes such a trace, the presence of such a looping instruction enables the tracer to know that it must restart the trace once it has reached the end. Now that the tracer has reached the starting point of the trace again, and has completed its recording, the tracer can start executing the trace it has just recorded instead of switching back to normal interpretation. We therefore switch the execution-phase of the tracer-state and replace the fourth field of the tracer-state by the trace-node we have just created, so that we can start executing it.

5.9 Trace execution

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 low-level

instruction ι on the program-state ς. Recall that a guard instruction is considered to be the same as any other low-level instruction.

$$\begin{aligned}&\mathbf{ts}(\,TE,\ tc,\ \varsigma,\ \mathbf{tn}(val,\ \iota : \tau)) \rightarrow \\ &\quad \mathbf{ts}(\,TE,\ tc,\ \varsigma',\ \mathbf{tn}(val,\ \tau)) \\ &\quad \text{if } \iota(\varsigma) = \mathbf{lliStep}(\varsigma')\end{aligned} \tag{5.10}$$

Figure 5.22: Trace execution if no guard fails

In rule 5.10, we explore the most common case: we apply an instruction from the trace on the current program-state and this instruction returns a **lliStep** containing the program-state resulting from applying this instruction. Other than the four guard instructions, all instructions always result in this kind of return type. For the resulting tracer-state, we must only update our program-state *as well as the trace that we are currently executing*. After executing this one instruction of the trace, we continue with the other instructions of the trace, so that in the next step we may apply the subsequent instruction. If we wouldn't update the trace, we would keep applying the same instruction forever.

$$\begin{aligned}&\mathbf{ts}(\,TE,\ tc,\ \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta),\ \mathbf{tn}(val,\ \iota : \tau)) \rightarrow \\ &\quad \mathbf{ts}(NI,\ tc,\ \mathbf{ps}(c,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta),\ \mathit{False}) \\ &\quad \text{if } \iota(\varsigma) = \mathbf{lliEvent}(\mathbf{guardFailed}(c))\end{aligned} \tag{5.11}$$

Figure 5.23: Trace execution if a guard fails

Rule 5.11 expresses the case where some guard has failed. The rule states that we should then switch our execution-phase to normal interpretation. Additionally, as mentioned in Section 5.6, interpretation should restart from the point in the program that corresponds with the guard failure. This point is defined as either an expression or a continuation. All guards are aware of to which point in the program they correspond, so when a guard fails, it can pass this location as part of the **guardFailed** signal, as was mentioned in Section 5.6. We therefore also replace the control component of the program-state by the new control, i.e., the expression or the continuation passed in the **guardFailed** structure. This ensures us that interpretation is restarted from the correct location in the code.

$$\begin{aligned}&\mathbf{ts}(\,TE,\ tc,\ \mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta),\ \mathbf{tn}(val,\ \text{`}())) \rightarrow \\ &\quad \mathbf{ts}(NI,\ tc,\ \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta),\ \mathit{False})\end{aligned} \tag{5.12}$$

Figure 5.24: Trace execution if the end of a trace has been reached

In rule 5.12, we consider the case where we have reached the end of a non-looping trace. Recall from Section 5.2 that we must then restart interpretation from the point in the program corresponding to right after the ending point of

the trace, i.e., the `can-close-loop` annotation that ended the trace recording. This ending point is easy to find: it is the continuation that currently resides at the top of the continuation stack of the program-state. Because we also trace the $popk()$ and $pshk(\phi)$ instructions that are used by the CESKθ-machine, we can be sure that the continuation stack is always kept up-to-date when executing the trace. This implies that when the end of the trace is reached, the same pops and pushes have been applied to the continuation stack during the execution of the trace as were applied during the recording of this trace. This then means that the continuation that was on the top of the continuation stack right after recording of the trace was finished, whatever this continuation may be, is now also located on the top of the stack. Jumping to the program location corresponding to right after the `can-close-loop` annotation that ended the trace is then nothing more than placing the continuation that is present on the top of the stack in the control component of the program-state.

$$\begin{aligned}&\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(val,\ \lambda : `())) \rightarrow \\ &\quad \mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(val,\ \tau)) \\ &\quad \text{where } \tau \text{ is the trace that has previously been recorded for } val\end{aligned} \tag{5.13}$$

Figure 5.25: Trace execution if the end of a looping trace has been reached

Rule 5.13 handles the case where we have reached the end of a trace that loops. Recall from Section 5.2 that we place a special looping instruction at the end of the trace when during the recording of the trace we notice that the trace loops. Recall also from this Section that we said that a trace *loops* if during trace recording we encounter a `can-start-loop` annotation carrying the same label as that of the trace we are recording. If we have reached the end of such a looping trace, we restart the trace: we look up the full trace belonging to the label of the trace we are executing and we replace the current, empty, trace by this new, full, trace.

Chapter 6

Extending the tracing framework

In this chapter we introduce several non-trivial extensions to the tracing framework that was presented in the previous chapter: we develop hot loop detection, guard tracing and trace merging in our framework. For each of these extensions, we give a small overview highlighting the intent of the extension, we then present an updated definition for the tracing machine and, if necessary, the low-level instruction set and interface. Afterwards we provide the formal semantics that specify how this extension is implemented. We develop these new extensions by gradually extending the previously defined formal semantics: each extension builds on top of the previous one. The creation of these extensions serves as the validation of this thesis. We create this set of extensions by extending the existing semantics, but we do not fundamentally change the underlying formalisms. This proves that our framework is both modular and powerful enough to allow for the development of additional features, i.e., we can introduce these extensions without having to fundamentally alter the existing semantics.

6.1 Hot loop detection

6.1.1 Overview

The first new extension we present is the detection of hot loops in a program's execution. A program loop is called "hot" if the loop is executed frequently enough, for some definition of frequent. By extending our framework with the capability for detecting hot program loops, we enable the tracer to more carefully select which loops to trace. Tracing compilation has the most effect when it is applied on those parts of the code where the program spends most of its time. Usually, these parts correspond to the program loops that are executed most frequently (Gal et al., 2009). Tracing parts of the program that are

executed only seldom generally does not result in any significant speed-up. It may even cause performance to degrade, as tracing always causes a runtime overhead in practice, and compiling and optimizing a trace is especially costly (Gal et al., 2006). Tracing a loop should therefore be seen as an investment which must be recouped: the compiler traces a loop in the hope that the time that is now saved in subsequent executions of this loop offsets the costs that had to be made for the tracing, compilation and optimization of the loop.

Loop hotness detection is a heuristic that is designed to aid the tracer in deciding which loops to trace. It works by holding off on tracing a loop when a `can-start-loop` annotation for a certain label is first seen until a `can-start-loop` annotation for this label has been encountered at least a fixed number of times. Although as a heuristic it does not completely remove the possibility of tracing uninteresting loops, it should at least cause the compiler to prioritize the tracing of hot loops over less frequently executed loops. This loop hotness detection heuristic is part of several tracing JIT compilers, including HotpathVM, TraceMonkey and SPUR (Gal et al., 2006, 2009; Bebenita et al., 2010).

6.1.2 Extending the tracing machine

We first extend the tracing machine in order to develop the semantics for detecting hot loops. The tracing machine is a state-machine operating on tracer-states. A tracer-state is defined as a four-tuple consisting of respectively: an $ExecutionPhase$, which expresses in which phase the tracing machine is currently executing the program, a $TracerContext$, which stores bookkeeping information related to tracing, as well as the traces that have already been recorded completely, the program-state, on which the interpreter and the low-level instructions from the trace operate, and possible a trace-node, which represent the trace that is currently being executed by the tracing machine, if the tracer has been set to the trace execution phase. Until now, the tracer-context was specified as a two-tuple consisting of a trace-node which associates the label that is currently being traced with the trace that has been recorded so far for this label, and a list of trace-nodes representing the traces that have already been recorded completely.

Our implementation for this heuristic depends on counting the number of times a `can-start-loop` annotation for a specific label is encountered, and only starting tracing once this counter exceeds a fixed number. Hence there must be some mechanism to find the number of times the CESKθ-machine has seen a `can-close-loop` annotation for each label. In our implementation, this is accomplished with the use of $LabelCounters$. A $LabelCounter$ associates a label with a counter. Since these $LabelCounters$ are additional bookkeeping information which our compiler must keep track of, we store these $LabelCounters$ in the $TracerContext$. When the interpreter sees a `can-start-loop` annotation, it updates the counter in the $LabelCounter$ associated with the annotation's label. Finding the number of times a label has been seen is then accomplished by looking up its counter in the list of $LabelCounters$.

$$
\begin{aligned}
tc \in \mathit{TracerContext} &= \mathbf{tc}(\mathit{False} + \mathit{TraceNode},\ \mathit{TNs},\ \mathit{LCs}) \\
lc \in \mathit{LabelCounter} &= \mathbf{lc}(\mathit{TraceKey},\ \mathit{Integer}) \\
\mathit{LCs} &= \mathit{LabelCounter} : \mathit{LCs} \\
&\mid \epsilon
\end{aligned}
$$

Figure 6.1: The updated tracing machine for loop hotness detection

6.1.3 Semantics

Because this extension is built on top of the tracing machine and tracing semantics that were defined in the previous chapter, we include this extension by extending the tracing semantics. Loop hotness detection does not change how execution of traces is accomplished nor how tracing itself is performed. It only alters the way in which tracing is *started*, i.e., how we transition from the normal interpretation phase of trace execution to the trace recording phase. We therefore only have to update some of the evaluation rules that determine how normal interpretation is performed by the tracing machine. Figure 6.2 shows the evaluation rules for the tracing machine which have remained unchanged. These rules were already used by the tracer for normal interpretation and must not be updated in order to implement this heuristic. $\mathbf{ts}$ refers to a tracer-state, $tc$to a tracer-context, ς to a program-state and $\mathbf{tn}$ to a trace-node.

$$
\begin{aligned}
&\mathbf{ts}(\mathit{NI},\ tc,\ \varsigma,\ \mathit{False}) \rightarrow \\
&\quad \mathbf{ts}(\mathit{NI},\ tc,\ \varsigma',\ \mathit{False}) \\
&\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathit{False})
\end{aligned}
\tag{6.1}
$$

$$
\begin{aligned}
&\mathbf{ts}(\mathit{NI},\ tc,\ \varsigma,\ \mathit{False}) \rightarrow \\
&\quad \mathbf{ts}(\mathit{NI},\ tc,\ \varsigma',\ \mathit{False}) \\
&\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathbf{CCL}(\mathit{val}))
\end{aligned}
\tag{6.2}
$$

$$
\begin{aligned}
&\mathbf{ts}(\mathit{NI},\ tc,\ \varsigma,\ \mathit{False}) \rightarrow \\
&\quad \mathbf{ts}(\mathit{TE},\ tc,\ \varsigma',\ \mathbf{tn}(\mathit{val},\ \tau)) \\
&\quad \text{if } \mathtt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathbf{CSL}(\mathit{val})) \\
&\quad \text{and where } \tau \text{ is the trace that has previously been recorded for } \mathit{val}
\end{aligned}
\tag{6.3}
$$

Figure 6.2: The normal interpretation rules which have not changed

Rule 6.1 handles the most common case during normal interpretation, where the interpreter has not encountered an annotation. Rule 6.2 expresses that when the interpreter encounters a `can-close-loop` annotation during normal interpretation, interpretation should just continue as before. Rule 6.3 states that if a `can-start-loop` annotation is encountered for a label that has already been traced, the recorded trace must be executed.

Figure 6.3 shows the evaluation rules that are added to these previous rules in order to implement loop hotness detection. In rule 6.4, the CESKθ-machine has encountered a `can-start-loop` annotation for a label that it has not yet seen and hence for which no entry in the list of $LabelCounters$ has been created yet. The tracing machine makes a $LabelCounter$ for this label, adds it to the list and continues interpretation. Rule 6.5 specifies the case where the CESKθ-machine sees a `can-start-loop` annotation for 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 6.6, a label is encountered that has turned hot, causing the tracing machine to start tracing.

$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ LCs),\ \varsigma,\ False) \rightarrow \quad (6.4)$$
$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ LCs),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathbf{CSL}(val))$$
$$\text{and if no } LabelCounter \text{ for } val \text{ exists yet}$$
$$\text{and if no trace for } val \text{ has been recorded yet}$$

$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ lc_1 : ... : \mathbf{lc}(val,\ k) : ... : lc_n),\ \varsigma,\ False) \rightarrow \quad (6.5)$$
$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ lc_1 : ... : \mathbf{lc}(val,\ k+1) : ... : lc_n),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathbf{CSL}(val))$$
$$\text{and if } k < Threshold$$
$$\text{and if no trace for } val \text{ has been recorded yet}$$

$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ lc_1 : ... : \mathbf{lc}(val,\ k) : ... : lc_n),\ \varsigma,\ False) \rightarrow \quad (6.6)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(val,\ \tau),\ TNs,\ lc_1 : ... : \mathbf{lc}(val,\ k) : ... : lc_n),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \tau,\ \mathbf{CSL}(val))$$
$$\text{and if } k >= Threshold$$
$$\text{and if no trace for } val \text{ has been recorded yet}$$

Figure 6.3: The additional normal interpretation rules

6.2 Guard tracing

6.2.1 Overview

Guard tracing is a technique designed to mitigate the performance penalties associated with guard failures. Under normal circumstances, a guard failure causes the execution of a trace to be aborted, and causes the tracing compiler to restart normal interpretation. This effectively creates a large hit to performance, because the compiled and heavily-optimized trace must be abandoned in favor of interpretation of the original, non-optimized, code. Additionally, the very act of restarting interpretation is also responsible for a large performance penalty (Bala et al., 2000; Chang et al., 2009).

The idea behind guard tracing is to start recording a trace from the point of a guard failure. Whenever the same guard in the same trace then fails again, execution can be switched from the first trace to the trace that was recorded for that guard. Instead of having to restart interpretation, trace execution can then keep continuing until the trace has come at an end or another guard fails for which no trace has been recorded yet.

Guard tracing is a feature often included in contemporary tracing compilers, such as the RPython and Dynamo compilers, SPUR and Tamarin-Tracing (Schneider & Bolz, 2012; Bala et al., 2000; Bebenita et al., 2010; Chang et al., 2009). Compilers that use this guard tracing mechanism often only start tracing a guard when it fails often enough, similar to how labels were only traced when they were considered to be hot in the previous section. Our guard tracing extension that is presented in this section does not contain such a mechanism however, tracing always starts immediately after a guard failure.

A trace that was recorded for a guard failure is called a *guard trace*. When a guard failure occurs during the execution of a trace, and this causes the compiler to start recording a guard trace, we say that the first trace *spawned* the guard trace.

6.2.2 Extending the tracing machine

Similar to the traces we defined in the previous chapter, guard traces are identified through trace-keys. Until now, trace-keys and labels both referred to the same idea: an expression that can be used to give a unique name to each trace. Trace-keys were used in the setting of the tracer-context, to keep track of which loop was being traced, while labels were used to construct `can-start-loop` and `can-close-loop` annotations. However, the difference between these two was superfluous in practice because every label could be perfectly matched onto a single trace-key and vice versa.

To introduce guard tracing, we first redefine the concept of a trace-key. We specify two kinds of trace-keys: guard trace-keys which are used to identify guard traces, and label trace-keys which are used for normal, non-guard traces. A label trace-key is nothing more than a wrapper for a label. Similar to guard traces, we name the kinds of traces that are identified by label trace-keys *label traces*. Guard traces must be identified not only by a guard, but also by the label of the trace in which that guard was located. Guard trace-keys therefore carry both a label and a *guard identifier*. Such a guard identifier is created for each guard in the trace so that if it fails, it is trivial to determine whether a guard trace for this guard already exists. Guard trace-keys and label trace-keys can carry the same labels, so a single label might appear in one label trace-key and multiple guard trace-keys. However, as before, a label cannot appear in more than one label trace-key.

$$\begin{aligned} tk \in \mathit{TraceKey} = \; & \mathbf{ltk}(\mathit{Val}) \\ & \mid \mathbf{gtk}(\mathit{Val},\ \mathit{GuardID}) \end{aligned}$$

$$gid \in GuardID = Identifier$$

Figure 6.4: The updated tracing machine for guard tracing

Other than the trace-keys, no components of the tracing machine must be updated.

6.2.3 Guard instructions

Because guards now carry special guard identifiers, we must change how guard instructions are formed. Making a guard carry an identifier can be accomplished by making the identifier a parameter of the guard instruction. This identifier is then created when the guard itself is created: when the guard is inserted into a trace by the tracer during trace recording.

In Figure 6.5, we give an example of how `guard-false` is now implemented. The implementation of the three other kinds of guards is identical.

$$\begin{array}{l}\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \#\mathtt{f},\ \theta) \xrightarrow{\mathtt{gfls(c,}\ gid\mathtt{)}} \\ \quad \mathbf{lliStep}(\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ \#\mathtt{f},\ \theta))\end{array}$$

$$\begin{array}{l}\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\mathtt{gfls(c,}\ gid\mathtt{)}} \\ \quad \mathbf{lliEvent}(\mathbf{guardFailed}(c,\ gid))\end{array}$$

Figure 6.5: The updated definition of `guard-false`

We must also update the interface used by low-level instructions, including guard instructions. When a **guardFailed** signal is piggy-backed after executing an instruction, we include the guard identifier of the guard that just failed. Note that we only have to change the definition for *EventSignal*.

$$EventSignal = \mathbf{guardFailed}(Control,\ GuardID)$$

Figure 6.6: The updated definition of *EventSignal*

6.2.4 Semantics

To update the semantics of the tracing machine in order to include guard tracing, we only alter the evaluation rules for trace recording and trace execution. We do not have to change any semantics for normal interpretation of a program, since guard tracing has no effect on this phase of a program's execution.

Trace recording

Recording a guard trace is identical to recording a label trace, including in how recording is terminated: we stop recording when we encounter a tracing annotation that carries the label of the label trace that initially spawned the guard trace. The updated evaluation rules for trace recording are therefore identical to the old rules, except that we now have to account for the fact that our trace-key can take two different forms. For brevity however, we can fuse these forms together. Figure 6.7 shows the updated evaluation rules for trace recording. The trace-key tk in these rules can refer to a trace-key of either kind. We also define a function **label** which, given a trace-key, extracts the label used in that trace-key. Note that labels are components in trace-keys of both kinds, so retrieving the label of a trace-key is always possible. Since we build this extension on top of the previous extension, loop hotness detection, the tracer-context **tc** is a three-tuple consisting respectively of: the trace-node **tn** that is currently being traced, the list of trace-nodes TNs of already recorded traces and the list of label-counters LCs that indicate how many times a `can-start-loop` annotation has been encountered for each label. The tracer-state **ts** is still a four-tuple consisting of: the $ExecutionPhase$representing the current phase of program execution, the tracer-context, the program-state ς and the current trace-node that is being executed. Because the rules that are presented here are evaluation rules for the trace recording phase, the $ExecutionPhase$is always set to TR, while the trace-node being executed is always $False$.

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs),\ \varsigma,\ False) \rightarrow \quad (6.7)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n),\ TNs,\ LCs),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ False)$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs),\ \varsigma,\ False) \rightarrow \quad (6.8)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n),\ TNs,\ LCs),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CCL}(val'))$$
$$\text{and if } \mathbf{label}(tk) = val$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs),\ \varsigma,\ False) \rightarrow \quad (6.9)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n),\ TNs,\ LCs),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CSL}(val'))$$
$$\text{and if } \mathbf{label}(tk) = val$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs),\ \varsigma,\ False) \rightarrow \quad (6.10)$$
$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ \mathbf{tn}(val,\ \tau) : TNs,\ LCs),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CCL}(val))$$
$$\text{and if } \mathbf{label}(tk) = val$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs),\ \varsigma,\ False) \rightarrow \quad (6.11)$$
$$\mathbf{ts}(TE,\ \mathbf{tc}(False,\ \mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n : \lambda) : TNs,\ LCs),\ \varsigma',\ tn)$$

where tn equals $\mathbf{tn}(\mathbf{ltk}(val),\ \tau')$
if $\texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{CSL}(val))$
and if $\mathbf{label}(tk) = val$

Figure 6.7: The updated trace recording rules

A guard trace-key stores a guard identifier to identify the guard that caused the creation of the associated guard trace. It also stores a label to make it possible to know when we should stop recording a guard trace-key. By capturing the label in a guard trace-key, we can terminate tracing just like we stop tracing for a label trace: when we encounter either tracing annotation carrying the corresponding label. Recall that if recording is stopped because of a `can-start-loop` annotation, we immediately start executing the recorded trace. This still holds true when we add guard traces, but stopping the recording of a guard trace because of a `can-start-loop` annotation causes the tracer to start executing the label trace associated with the label of that annotation, instead of the guard trace that was just recorded.

Trace execution

The trace execution phase of a program's execution must be updated to account for the novel handling of guard failures. Figure 6.8 shows the rules that remain unchanged. Rule 6.12 represents the most common case, where a normal instruction is executed and no event must be signalled. Rule 6.13 specifies that if execution of a non-looping trace has reached the end, normal interpretation must restart.

$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(val,\ \iota : \tau)) \rightarrow \mathbf{ts}(TE,\ tc,\ \varsigma',\ \mathbf{tn}(val,\ \tau)) \quad \text{if } \iota(\varsigma) = \mathbf{lliStep}(\varsigma') \tag{6.12}$$

$$\mathbf{ts}(TE,\ tc,\ \mathbf{ps}(e,\ \rho,\ \sigma,\ \phi : \kappa,\ v,\ \theta),\ \mathbf{tn}(tk,\ `())) \rightarrow \mathbf{ts}(NI,\ tc,\ \mathbf{ps}(\phi,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta),\ False) \tag{6.13}$$

Figure 6.8: The trace execution rules which have not changed

Figure 6.9 shows how we now handle guard failures during the execution of a trace. When a guard fails, we check whether a guard trace for the corresponding guard identifier already exists. Rule 6.14 states that if a guard trace exists, we swap the trace that is currently being executed for this guard trace. If there is no existing trace, we start tracing the guard. The guard trace-key is constructed by combining the label of the trace we were executing with the guard identifier carried back through the **guardFailed** signal. Similar to how guard failures were handled previously, we also switch the control component

of the program-state by the control passed via the guard. This is expressed in rule 6.15.

$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(tk,\ \iota : \tau)) \rightarrow \quad (6.14)$$
$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(\mathbf{gtk}(val',\ gid),\ \tau'))$$
$$\text{if } \iota(\varsigma) = \mathbf{lliEvent}(\mathbf{guardFailed}(c,\ gid))$$
$$\text{and where } \tau' \text{ is the trace that has previously been recorded for } gid$$

$$\mathbf{ts}(TE,\ \mathbf{tc}(\mathit{False},\ \mathit{TNs},\ \mathit{LCs}),\ \mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta),\ \mathbf{tn}(tk,\ \iota : \tau)) \rightarrow \quad (6.15)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(\mathbf{gtk}(val,\ gid),\ `()),\ \mathit{TNs},\ \mathit{LCs}),\ \mathbf{ps}(c,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta),\ \mathit{False})$$
$$\text{where } \mathbf{label}(tk) = val$$
$$\text{and if } \iota(\varsigma) = \mathbf{lliEvent}(\mathbf{guardFailed}(c,\ gid))$$
$$\text{and if no trace exists yet for } gid$$

Figure 6.9: The updated trace execution rules for handling guard failure

Note that we use the generic term tk to refer to the trace-key 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 guard trace-key for the new trace is the same as that of the guard trace-key for the trace that was aborted: the label of the label trace that spawned the initial guard trace.

Rule 6.16 in Figure 6.10 specifies that if we reach the end of a looping trace, no matter which kind of trace we are executing, we restart the loop by finding the label trace associated to the label of the trace-key whose trace we are currently executing. In other words, if we have reached the end of a looping guard trace, we do not restart this guard trace itself, but rather the label trace that spawned this guard trace.

$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(tk,\ \lambda : `())) \rightarrow \quad (6.16)$$
$$\mathbf{ts}(TE,\ tc,\ \varsigma,\ \mathbf{tn}(\mathbf{ltk}(val),\ \tau))$$
$$\text{where } \mathbf{label}(tk) = val$$
$$\text{and where } \tau \text{ is the trace that has previously been recorded for } val$$

Figure 6.10: Trace execution if the end of a looping trace has been reached

6.3 Trace merging

6.3.1 Overview

Trace merging is a technique intended to reduce redundant tracing. I works by merging traces when their underlying control-flow merges: instead of retracing a part of the program that has already been traced before, we isolate this

part and let both traces jump to this isolated trace when so required. To the best of our knowledge, this concept has only previously been explored in Tamarin-Tracing (Chang et al., 2009). It should be noted that the term 'trace merging' is used by some tracing compilers to refer to what is also called 'trace stitching'. Trace stitching is very similar to the technique of guard tracing, which we presented in the previous section: whenever a guard fails often enough, a guard trace is recorded. When this guard fails again, the guard trace is executed, instead of reverting to normal interpretation of the program.

Listing 6.1 shows some example code of how control-flow can merge. Whatever the value of `(= (random 2) 0)` may be, once the if-expression has been evaluated, the interpreter always executes the `(displayln "control-flow merged")` expression. The control-flow graph of this function can be represented as in Figure 6.11.

```
(define (do-something)
  (can-start-loop 'do-something)
  (if (= (random 2) 0)
      (displayln "true-branch")
      (displayln "false-branch"))
  (displayln "control-flow merged")
  (do-something))
```

Listing 6.1: A function where control-flow merges

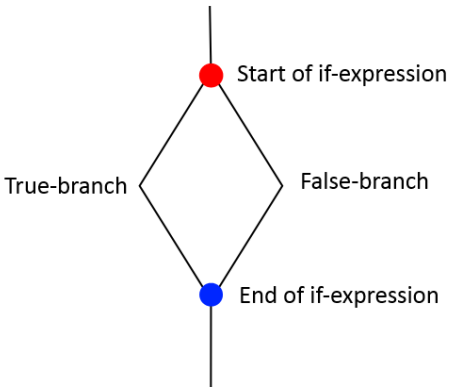
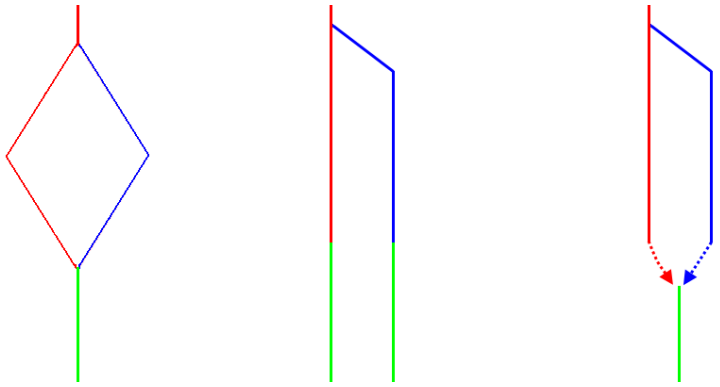


Figure 6.11: The logical control-flow of Listing 6.1

If the tracing machine traces `do-something`, it starts from the `can-start-loop` annotation, follows either the true- or the false-branch of the code, continues with the evaluation of `(displayln "control-flow merged")` and stops once the `can-start-loop` annotation is reached again in the next iteration. Suppose that the true-branch was traced, but that once the trace is executed, the guard corresponding with the `(= (random 2) 0)` condition fails. Because of the guard tracing feature that was developed in the previous section, the tracer starts recording a guard trace from the point of this guard failure,



(a) The logical control-flow (b) The resulting traces without trace merging applied (c) The resulting traces with trace merging applied

Figure 6.12: An example of how control-flow is translated into three traces

goes over the false-branch and the last `displayln`-expression and stops again when the `can-start-loop` annotation is next encountered.

Both the original label trace as well as the new guard trace then contain a part that is common to both: those instructions that were recorded after the end of the if-expression. In our tracing framework, having this duplicated part in both traces is redundant and undesirable: tracing always causes a certain runtime overhead and should therefore be avoided unless the resulting trace can sufficiently improve the performance of the compiler. It would therefore be better not to retrace this part of the program, but to reuse the first trace.

The idea for trace merging is to detect when control-flow merges while tracing. In that case, recording for the current trace is stopped and a new trace, which is called a *merge trace*, is started for this part of the program. If the tracing machine reaches the end of execution of the original trace, it continues execution in this new merge trace. If the first trace is executed, but a guard failure causes the machine to start recording a guard trace, we apply the same principles. Once the machine discovers that control-flow merges, it stops recording altogether and starts executing the merge trace. This ensures us that the part of the program after the merge-point is not duplicated in two separate traces.

Figure 6.12 shows which traces are formed when tracing a program with a certain logical control-flow. The red path represents the path that was taken through the program before reaching the control-flow split, and subsequently following the left branch of the control-flow. The blue path represents the other branch of the control-flow. The green path represents the path through the program immediately following the point where both branches of the control-flow merge back. Without trace merging, two traces are created and the green path appears in both. With trace merging, the green path is isolated into a separate merge trace and execution jumps from these two traces to the merge trace, as indicated by the dotted arrows.

Note that the trace that was created second before being merged is always a guard trace, since a split in control-flow always results in the failure of a guard, and hence the creation of a guard-trace. The first trace can be any kind of trace. It should also be noted that we only handle the case where the control-flow splits because of an if-expression. There are other possible locations where the control-flow may split, such as when calling a higher-order parameter. However, the trace merging extension that we introduce here is not designed to handle these cases, so they are ignored.

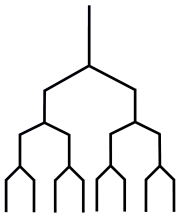
Trace explosion

The full potential of trace merging becomes evident when dealing with *trace explosion*. Trace explosion refers to the phenomenon where an exponential number of traces are created, because the number of underlying control-flow branches explodes. Trace explosion affects runtime performance, since an exponential amount of traces have to be recorded and trace recording causes a runtime overhead. It also increases the amount of memory required by the system.

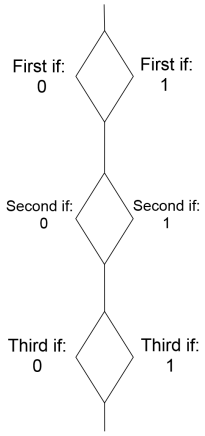
Consider the example program presented in Listing 6.2.

```
(define (trace-explosion)
  (can-start-loop 'trace-explosion)
  (if (= (random 2) 0)
      (displayln "First if: 0")
      (displayln "First if: 1"))
  (if (= (random 2) 0)
      (displayln "Second if: 0")
      (displayln "Second if: 1"))
  (if (= (random 2) 0)
      (displayln "Third if: 0")
      (displayln "Third if: 1"))
  (trace-explosion))
```

Listing (6.2) A program causing trace explosion to arise



(a) The resulting traces without trace merging



(b) The resulting traces with trace merging

Figure 6.13: An example of how trace merging solves the issue of trace explosion

When tracing this program without trace merging, a total of eight different traces are created. The total number of traces is exponential in the number of guards that are inserted in the trace. When using merge tracing however, we can take advantage of the fact that control-flow merges after each if-expression, so the number of traces remains linear. In fact, there are no parts of the program that are duplicated over multiple traces.

Trace merging and trace optimization

In real-world tracing compilers, trace merging can be detrimental to the runtime performance because of how it affects the optimizations that are applied on the traces generated by the compiler.

Consider the program shown in Listing 6.3.

```
(define (trace-explosion)
  (can-start-loop 'trace-explosion)
  (define x #f)
  (if (= (random 2) 0)
      (set! x 0)
      (set! x 20))
  (if (= (random 2) 0)
      (set! x (+ x 5))
      (set! x (- x 5)))
  (if (= (random 2) 0)
      (set! x (+ x 1))
      (set! x (- x 1)))
  (displayln x)
  (trace-explosion))
```

Listing 6.3: A program where trace merging may be detrimental

In this program, each branch of the control-flow results in a unique value for the variable `x`. If no trace merging is applied, a unique trace is generated for each control-flow branch. This means that each trace is able to determine what the value of `x` will be in the `displayln`-expression at the end of the program, allowing them to optimize the execution of this expression. When using trace merging, this is no longer possible, since this information disappears when the jumping from a trace to an intermediate merge trace.

6.3.2 Syntax

To detect the locations where control-flows merge, or inversely where control-flow splits in two, we use a new set of annotations. The `merges-control-flow` annotation signals that control-flow merges at this location, `splits-control-flow` expresses a split in the control-flow.

$$
\begin{aligned}
e \in \mathit{Exp} = \ & \ldots \\
& \mid \texttt{(merges-control-flow)} \\
& \mid \texttt{(splits-control-flow)}
\end{aligned}
$$

Figure 6.14: The new definition of *Exp*

Listing 6.4 shows how these annotations can be used in practice. We place a `splits-control-flow` annotation in our if-condition, which is the last ex-

pression that is evaluated before control-flow diverges into two branches. Once the if-expression has been completely evaluated, the `merges-control-flow` annotation signals that control-flow merges.

```
(define (do-something)
  (can-start-loop 'do-something)
  (if (begin (= (random 2) 0)
             (splits-control-flow))
      (displayln #t)
      (displayln #f))
  (merges-control-flow)
  (displayln "control-flow merged")
  (do-something))
```

Listing 6.4: An example of how the new annotations are used

In principle, these merging annotations have to be placed around each location in the code where control-flow may diverge. Although this might seem extremely cumbersome for the programmer, recall that the objective of our tracing compiler is to perform meta-tracing. This implies that we are only interested in performing trace merging on the traces generated by the control-flow of the underlying user-program. In that case, we only need to place these merging annotations around those parts of the input interpreter that handle the control-flow of the user-program. In practice, this comes down to only a handful of merging annotations at most.

6.3.3 Extending the tracing machine

To create these merge traces, we need some mechanism to identify them. We can accomplish this similar to how we identify guard traces: by using a combination of a trace's label and an identifier, called a *merge identifier*. We extend the tracing machine with a stack that stores these merge identifiers. Since we now use three kinds of traces, label traces, guard traces and merge traces, we define a third kind of trace-key **mtk** to serve as the trace-key for merge traces.

$$
\begin{aligned}
tc \in \mathit{TracerContext} &= \mathbf{tc}(\mathit{False} + \mathit{TraceNode},\ \mathit{TNs},\ \mathit{LCs},\ \mathit{MergeIDStack}) \\
tk \in \mathit{TraceKey} &= \mathbf{gtk}(\mathit{Val},\ \mathit{GuardID}) \\
&\mid \mathbf{ltk}(\mathit{Val}) \\
&\mid \mathbf{mtk}(\mathit{Val},\ \mathit{MergeID}) \\
gid \in \mathit{GuardID} &= \mathit{Identifier} \\
mid \in \mathit{MergeID} &= \mathit{Identifier} \\
\mu \in \mathit{MergeIDStack} &= \mathit{MergeID} : \mathit{MergeIDStack} \\
&\mid \epsilon
\end{aligned}
$$

Figure 6.15: The updated tracing machine for trace merging

6.3.4 Interface

Since we have introduced two new annotations, we also define two new annotation signals, **MCF** for **m**erges-**c**ontrol-**f**low annotations and **SCF** for **s**plits-**c**ontrol-**f**low annotations. When the interpreter encounters either of these two new annotations, it carries back the appropriate signal, as will be formalized later on, when presenting the semantics of trace merging.

$$\begin{aligned} \mathit{AnnotationSignal} = \ & \mathbf{CCL}(\mathit{Val}) \\ & \mid \mathbf{CSL}(\mathit{Val}) \\ & \mid \mathbf{MCF} \\ & \mid \mathbf{SCF} \end{aligned}$$

Figure 6.16: The new definition of $\mathit{AnnotationSignal}$

6.3.5 Handling merging annotations

The difficulty of implementing trace merging is twofold: while tracing, we must identify the correct locations where control-flow merges and while executing a trace, we must jump from the execution of the trace to the execution of the correct merge trace. The first problem is solved by introducing the two merging annotations. The second problem can be solved by using merge identifiers to correctly identify each merge trace and by inserting instructions into the trace that explicitly refer to the identifier of the merge trace to which the tracing machine must jump. Referring to these merge identifiers is accomplished through the merge identifiers stack that is part of the tracing machine: when the interpreter encounters a `splits-control-flow` annotation while a trace is being recorded, an instruction is placed in the trace that pushes a certain identifier on the stack when executed, and when the interpreter sees a `merges-control-flow` annotation, another instruction is inserted which, when executed, causes the tracing machine to pop the topmost identifier, abort execution of the current trace and jump to the merge trace that corresponds with this identifier.

We have to use a stack of identifiers instead of storing just the identifier that corresponds with the last `splits-control-flow` annotation because of the fact that if-expressions can be nested, as shown in Listing 6.5. Evaluating a nested if-expression results in the creation of multiple merge-traces, all of which must be properly identified. By using a stack, we only have to pop the merge identifiers stack to retrieve the merge identifier for the innermost merge trace. The merge identifier for the outermost identifier can be found by popping from this stack a second time.

```
(define (do-something)
  (can-start-loop 'do-something)
  (define x (random 4))
  (if (begin (< x 2)
             (splits-control-flow))
      (begin (if (begin (= x 0)
                        (splits-control-flow))
                 (displayln 0)
                 (displayln 1))
             (merges-control-flow)
             (displayln "smaller than 2"))
      (begin (if (begin (= x 2)
                        (splits-control-flow))
                 (displayln 2)
                 (displayln 3))
             (merges-control-flow)
             (displayln "greater than or equal to 2")))
      (merges-control-flow)
      (displayln "all control-flows have now merged")
      (do-something))
```

Listing 6.5: A nested if-expression

It is important to understand that a `merges-control-flow` annotation is always paired with a `splits-control-flow` annotation. The `splits-control-flow` annotation generates and pushes to the stack a merge identifier at the last possible location in the program that is common to both traces, so that when an identifier is popped from the stack in both traces, they always pop the *same* identifier. Figure 6.17 shows a concrete example of how these merging annotations are translated to instructions in the trace. We use the $pshm(mid)$-instruction to push mid onto the merge identifier stack, and $popm()$ to pop the first merge identifier and then immediately jump to the merge trace associated with it. The label trace contains the trace for the first part of this program, before the control-flow splits. Suppose that `(= (random 2) 0)` evaluated to `#t` while recording the trace. The label trace then contains a `guard-true` instruction. At the end of the label trace, a $popm()$ instruction is placed, so that mid is popped from the stack and execution continues in the corresponding merge trace. Once the `guard-true` instruction fails during the execution of the trace, the guard trace on the right is created. Similar to the label trace, the $popm()$ instruction at the end of the trace makes sure that the execution of both traces merges.

```
(define (do-something)
  (can-start-loop 'do-something)
  (if (begin (= (random 2) 0)
             (splits-control-flow))
      (displayln #t)
      (displayln #f))
  (merges-control-flow)
  (displayln "control-flow merged")
  (do-something))
```

Listing (6.6) A SLIPT program with diverging control-flow

```
...
pshm(mid)
...
gtru((displayln #f), gid)
...
litv(#t)
popk()
pshk(ratork(1))
lvar(displayln)
popk()
anat(1)
popm()
```

(a) The label trace

```
...
litv(#f)
popk()
pshk(ratork(1))
lvar(displayln)
popk()
anat(1)
popm()
```

(b) The guard trace

Figure 6.17: A concrete example of how two traces are merged together

Instead of generating a unique merge identifier whenever the interpreter sees a `splits-control-flow` annotation, we could have made the `merges-control-flow` annotation carry a label instead, similar to how `can-close-loop` and `can-start-loop` annotations carry labels. It would be very likely then however, that a certain merging annotation is encountered in multiple independent traces. If we place a merging annotation in a commonly executed function, then this function is likely to be inlined in multiple different traces. It would then be required to add a mechanism that makes sure that independent traces do not execute the same merge trace, because this would add unnecessary complexity to the trace merging feature. To avoid this issue entirely, a new identifier is generated every time a `splits-control-flow` annotation is encountered.

6.3.6 Low-level instructions interface

In the previous subsection, we introduced the $pshm(mid)$ and $popm()$ instructions. These instructions are used to manipulate the stack of merge identifiers in the tracing-state. Recall however that all low-level instructions operate on the program-state: they take a program-state as input and return an *InstructionReturn*: either a new program-state or an *EventSignal*, to signal events such as guard failures. Since these instructions have to alter the program-state instead of the tracing-state, we implement these instructions so that they always return an *EventSignal* in which an appropriate signal is piggybacked. The tracing machine is then updated to properly interpret these signals.

We update the low-level interface in the following way:

$$
\begin{aligned}
\mathit{InstructionReturn} &= \mathbf{lliStep}(\mathit{ProgramState}) \\
&\mid \mathbf{lliEvent}(\mathit{EventSignal}) \\
\mathit{EventSignal} &= \mathbf{guardFailed}(\mathit{Control}) \\
&\mid \mathbf{popSplitsCF} \\
&\mid \mathbf{pushSplitsCF}(\mathit{MergeID})
\end{aligned}
$$

Figure 6.18: The updated interface for low-level instructions

$pshm(mid)$ and $popm()$ are then implemented as shown in Figure 6.19.

$$
\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{popm()}} \mathbf{lliEvent}(\mathbf{popSplitsCF})
$$

$$
\mathbf{ps}(e,\ \rho,\ \sigma,\ \kappa,\ v,\ \theta) \xrightarrow{\texttt{pshm}(mid)} \mathbf{lliEvent}(\mathbf{pushSplitsCF}(mid))
$$

Figure 6.19: The implementation of the two new instructions

In essence, we make full use of the flexibility offered by the inclusion of **lliEvent** in the low-level instruction interface.

6.3.7 Semantics

Normal interpretation

We now define updated semantics for the tracing machine. All three phases of program execution must be updated: normal interpretation, trace recording and trace execution. We do not have to change any existing rules for the normal interpretation phase, but we do have two add two new rules to handle the evaluation of the two new merging annotations. For brevity, we only show these two new rules in Figure 6.20. The other evaluation rules for this phase, to which we add these new rules, are specified in Subsection 6.1.3.

Rule 6.17 states that if the CESKθ-machine encounters a `splits-control-flow` annotation, we generate a new merge identifier and push it to the stack of merge identifiers. Rule 6.18 specifies that a `merges-control-flow` annotation is evaluated by popping the topmost merge identifier from the stack.

$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ LCs,\ \mu),\ \varsigma,\ False) \rightarrow \quad (6.17)$$
$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ LCs,\ \alpha : \mu),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{SCF})$$
$$\text{and where } \alpha \text{ is a new merge identifier}$$

$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ LCs,\ mid : \mu),\ \varsigma,\ False) \rightarrow \quad (6.18)$$
$$\mathbf{ts}(NI,\ \mathbf{tc}(False,\ TNs,\ LCs,\ \mu),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{MCF})$$

Figure 6.20: The normal interpretation rules for the two new annotations

Trace recording

The semantics for recording a trace are almost identical to the previous semantics, except that we add three rules to handle the new merging annotations. When the interpreter sees a `splits-control-flow` annotation, we generate a new merge identifier, push it to the stack, but continue tracing as before. When we encounter a `merges-control-flow` annotation however, we first check whether a merge trace for the identifier currently at the top of the stack of merge identifiers already exists. If a merge trace for this identifier exists, we stop recording, store the trace and start executing this merge trace. Otherwise, we also stop recording and store the trace, but we start executing this merge trace instead of continuing tracing.

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs,\ \mu),\ \varsigma,\ False) \rightarrow \quad (6.19)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n : pshm(\alpha)),\ TNs,\ LCs,\ \alpha : \mu),\ \varsigma',\ False)$$
$$\text{where } \alpha \text{ is a new merge identifier}$$
$$\text{and if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{SCF})$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs,\ mid : \mu),\ \varsigma,\ False) \rightarrow \quad (6.20)$$
$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(\mathbf{mtk}(val,\ mid),\ `()),\ tn : TNs,\ LCs,\ \mu),\ \varsigma',\ False)$$
$$\text{if } \texttt{step}(\varsigma) = \mathbf{cesktStep}(\varsigma',\ \iota_1 : ... : \iota_n,\ \mathbf{MCF})$$
$$\text{and where } tn \text{ equals } \mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n : popm())$$
$$\text{and if no trace exists yet for } mid$$

$$\mathbf{ts}(TR,\ \mathbf{tc}(\mathbf{tn}(tk,\ \tau),\ TNs,\ LCs,\ mid : \mu),\ \varsigma,\ False) \rightarrow \quad (6.21)$$
$$\mathbf{ts}(TE,\ \mathbf{tc}(False,\ \mathbf{tn}(tk,\ \tau : \iota_1 : ... : \iota_n : popm()) : TNs,\ LCs,\ \mu),\ \varsigma',\ tn)$$
$$\text{Where } tn \text{ equals } \mathbf{tn}(\mathbf{mtk}(val,\ mid),\ \tau')$$
$$\text{and where } \tau' \text{ is the trace previously recorded for } mid$$

Figure 6.21: The updated trace recording rules

Trace execution

To update the semantics of our tracing machine for trace execution, we only add two new rules to handle the two new $EventSignals$. When a $\mathbf{pushSplitsCF}(mid)$ signal is returned, i.e., we have just executed a $pshm(mid)$ instruction, we push mid on the merge identifiers stack of the tracing-state and we continue executing the trace. When we receive a **popSplitsCF** signal, we pop the first merge identifier from its stack and start executing the merge trace associated with this identifier.

$$\mathbf{ts}(TE,\ \mathbf{tc}(False,\ TNs,\ LCs,\ \mu),\ \varsigma,\ \mathbf{tn}(tk,\ \iota : \tau)) \rightarrow \mathbf{ts}(TE,\ \mathbf{tc}(False,\ TNs,\ LCs,\ mid : \mu),\ \varsigma,\ \mathbf{tn}(tk,\ \tau)) \quad \text{if } \iota(\varsigma) = \mathbf{lliEvent}(\mathbf{pushSplitsCF}(mid)) \tag{6.22}$$

$$\mathbf{ts}(TE,\ \mathbf{tc}(False,\ TNs,\ LCs,\ mid : \mu),\ \varsigma,\ \mathbf{tn}(tk,\ \iota : \tau)) \rightarrow \mathbf{ts}(TE,\ \mathbf{tc}(False,\ TNs,\ LCs,\ \mu),\ \varsigma,\ \mathbf{tn}(\mathbf{mtk}(val,\ mid),\ \tau')) \quad \text{if } \iota(\varsigma) = \mathbf{lliEvent}(\mathbf{popSplitsCF}) \text{ and where } \tau' \text{ is the trace that has previously been recorded for } mid \tag{6.23}$$

Figure 6.22: The updated trace execution rules

6.4 Validation conclusion

In this chapter, we grew our minimalistic tracing compiler from Chapter 5 into a more extensive and useful application by adding three new, non-trivial, extensions: loop hotness detection, guard tracing and trace merging. To implement these features, we had to extend the syntax of SLIPT, the semantics of the tracing machine and the interface between the interpreter and the tracer. However, we did not have to fundamentally rewrite the semantics that were defined in Chapter 5: we extended the components of our framework.

Introducing these extensions is empirical proof that our semantics are powerful enough to model real-world, state-of-the-art tracing compilers and that our framework can be extended with even more features in the future. By providing formal semantics for each of these additions, we enable future research where we formally reason over the impact that these mechanisms have on tracing compilation.

Chapter 7

Conclusion

7.1 Summary

Trace-based just-in-time (JIT) compilation is a technique where hot program paths are identified at runtime, recorded into a trace, and subsequently compiled and optimized. The next time this path is selected, the compiled trace is executed instead of the original code. Meta-tracing JIT compilation is a variant of this approach in which the compiler does not trace a user-program directly, but traces the execution of a language interpreter while it evaluates the user-program. To enable meta-tracing, language implementers must place annotations at certain locations in their interpreter.

We have presented a minimalistic meta-tracing JIT compiler for SLIPT. We formally defined SLIPT by specifying its syntax and creating two execution models. The first model was a CESKθ-machine, which was later refactored into a low-level instruction (LLI) CESKθ-machine that uses traceable low-level instructions to execute its input programs.

We transformed this LLI CESKθ-machine into a meta-tracing compiler by introducing a formal model describing the execution of a tracing machine, implemented as a state-machine operating on tracer-states, and attaching this machine to the LLI CESKθ-machine. To attach both entities to each other, we constructed an interface between the CESKθ-machine and the tracing machine. This interface allows the tracing machine to interact with the CESKθ-machine and record its actions. We created another interface between the low-level instructions used by the LLI CESKθ-machine and the tracing machine, in order to correctly model the execution of traces and the handling of guard failures. These interfaces also create a clean division between the parts of the framework that deal with interpreting a SLIPT program, and those that deal with all tracing functionalities.

To validate this framework, we added several extensions to the compiler: loop hotness detection, guard tracing and trace merging. The fact that we could extend our model with additional, non-trivial features that were not originally

planned on being included in the framework, indicates that it should be feasible to add other features to our compiler as well in the future.

7.2 Contributions

This thesis makes the following contributions:

- It presents a **minimalistic** meta-tracing compiler. We define a minimalistic meta-tracing compiler as a compiler containing no more than the following features:
  - Capable of modelling all aspects of a program's execution: the recording and execution of traces, as well normal interpretation of the program without any tracing whatsoever
  - Capable of handling guard instructions and aborting the execution of traces when necessary
  - Capable of handling the hints used by language developers in their interpreters to enable meta-tracing of these interpreters

  All of these features have been implemented without adding any non-essential new features, in order to reduce complexity. This was accomplished by building a meta-tracing compiler from the ground up, starting from an ordinary SLIPT interpreter and developing the compiler *around* this interpreter. This allows us to define our compiler as a machine that records the actions performed by another machine, the interpreter. Recording the interpreter's actions happens via an interface between the interpreter and the tracing machine.

  Furthermore, this approach also allows us to make only as few changes to the interpreter as possible. We only had to wrap the return values of the interpreter's transition rules, so that the interpreter conforms to the interface, and introduce guard instructions into a handful of transition rules.

- Our compiler is **configurable**. Because interpretation of a program is separated from tracing, we can switch one interpreter for another. Although we have described our compiler in the context of an interpreter for the SLIPT programming language, it is entirely possible to switch out the current interpreter for any other, even for an interpreter that executes a completely different programming language. All that is required from an interpreter is that it conforms to a specific interface and that its actions can be reified in the form of (low-level) instructions, so that they can be traced. These instructions themselves must also adhere to a particular interface. Previous work in formalizing tracing compilation did not contain this characteristic, because the execution semantics of their programming language were too tightly coupled with the semantics for recording and executing traces.

- Our compiler is **extensible**. By carefully deciding how the interface between the interpreter and the tracer, as well as the interface between the low-level instructions of a trace and the tracer, is constructed, we allow for flexibility in our framework. Both interfaces define *signals* that can be sent by respectively the interpreter or the low-level instructions. It is likely that introducing a new feature requires the interpreter or the low-level instructions to send signals other than those that have been defined in our framework. Extending the set of allowed signals is straightforward however, as was proven during the development of trace merging. As our validation indicates, if both interfaces are flexible enough, new extensions can be introduced to the compiler while only having to update a subset of the previously defined execution rules.
- Our compiler is **executable**. We have not only provided formal semantics for all parts of our framework, we have also implemented our full framework, consisting of both CESKθ-machines, the low-level instruction set, the interfaces and the tracing-machine, in Racket. Furthermore, we tested our meta-tracing compiler by using it to execute an interpreter, written in SLIPT, for SLIP, thereby effectively meta-tracing these SLIP programs.

In contrast with the frameworks developed by Guo & Palsberg (2011) and Dissegna et al. (2014), our framework is not bound to one particular execution model. Furthermore, unlike their frameworks, it adequately covers not only the execution of traces, but also how traces are recorded and how normal interpretation proceeds, when no tracing is going on whatsoever.

7.3 Future work

7.3.1 Optimizations

Although we have added three extensions to our framework, loop hotness detection, guard tracing and trace merging, we did not add any optimizations that can be applied on a trace. Some features may have particular side-effects when they are combined with certain optimizations. The inclusion of trace merging for example, may affect how much a trace can be specialized. Without trace merging, two separate traces would be created when control-flow splits and these traces would never merge. This allows the compiler to specialize both branches with respect to the actions taken during the part of the program where control-flow was split. With trace merging, the compiler cannot make such specializations because the merge trace must contain instructions that are valid for the traces that result from both branches of the control-flow.

The introduction of several commonly applied optimizations would allow us to evaluate new features of our framework in more detail.

7.3.2 Direct versus meta-tracing

In this thesis, we presented a meta-tracing compiler. In the context of the framework that we demonstrated, this specifically meant that the language developer was required to add tracing or merging annotations to certain parts of the language interpreter. We effectively moved the responsibility for invoking certain features of the compiler, such as when to start or stop tracing, from the compiler itself to the programmer. Although it may seem infeasible for developers to be aware of where each annotation must be placed, this does not present any real problems in practice, because only the language developers who create their interpreter on top of our compiler are required to posses this knowledge. Practice has also proven that only a small amount of annotations are needed when meta-tracing user-programs on top of a language interpreter.

When using direct tracing, the issue of requiring programmers to know where to place each annotation becomes relevant again. When the input to a tracing compiler is a regular user-program, as with direct tracing, instead of a language interpreter, as with meta-tracing, many more annotations must be placed if one wishes to trace all relevant program paths. It is undesirable to expect all programmers to know where to correctly place each annotation.

We expect that we can convert our meta-tracing compiler to a direct tracing compiler by only updating the interpreter to which the compiler is bound, e.g., the LLI CESKθ-machine. Instead of sending only tracing signals when this interpreter encounters tracing annotations, this interpreter itself would be required to detect when the program loops or when control-flow splits or merges, and send the correct signal to the tracer.

7.3.3 Additional features

During the development of this compiler, other features were also considered to further validate our framework. Additional extensions included a heuristic that aborted a trace when it became too long, and a mechanism to differentiate between different kinds of loops, e.g., loops that always ran for only a fixed, small number of iterations or loops which were executed an undetermined number of iterations. Some of these features were partly or completely added to the executable version of our framework, but no formal semantics were constructed for specifying their implementation. Since it was possible to add these new extensions to our executable compiler, we expect it to be straightforward to also include them in our formal framework.

7.4 Overall conclusion

We have successfully created a formal execution model for a meta-tracing JIT compiler from scratch. This compiler is minimalistic: it contains only those features that are absolutely essential for the process of meta-tracing compilation,

such as modelling the recording and execution of traces, normal interpretation, guard failures and the handling of annotations. Our compiler works by explicitly recording the actions of a separate interpreter through an interface. This configuration allows us switch from tracing the actions of one execution model to another, as long as these models correspond to the defined interface. As a validation of our framework, we added three non-trivial extensions to our formal model: detection of hot loops, guard tracing and trace merging. This indicates that it should be possible to add even more features to our compiler.

References

Abrams, P. S. (1970). *An apl machine* (Doctoral dissertation, Stanford University, Stanford, CA, USA). Retrieved from http://www.slac.stanford.edu/pubs/slacreports/reports07/slac-r-114.pdf (AAI7022146)

Ancona, D., Ancona, M., Cuni, A., & Matsakis, N. D. (2007). Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 symposium on dynamic languages* (pp. 53–64). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1297081.1297091 doi: 10.1145/1297081.1297091

Aycock, J. (2003, June). A brief history of just-in-time. *ACM Comput. Surv.*, *35*(2), 97–113. Retrieved from http://doi.acm.org/10.1145/857076.857077 doi: 10.1145/857076.857077

Bala, V., Duesterwald, E., & Banerjia, S. (2000, May). Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, *35*(5), 1–12. Retrieved from http://doi.acm.org/10.1145/358438.349303 doi: 10.1145/358438.349303

Baumann, S., Bolz, C. F., Hirschfeld, R., Kirilichev, V., Pape, T., Siek, J., & Tobin-Hochstadt, S. (2015). Pycket: A tracing jit for a functional language. In *Icfp*. (accepted for publication)

Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., & Venter, H. (2010). Spur: A trace-based jit compiler for cil. In *Proceedings of the acm international conference on object oriented programming systems languages and applications* (pp. 708–725). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1869459.1869517 doi: 10.1145/1869459.1869517

Bolz, C. F. (2012). *Meta-tracing just-in-time compilation for rpython* (Doctoral dissertation, Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, Düsseldorf, Germany). Retrieved from http://cfbolz.de/stuff/cfbolz-meta-tracing.pdf

Bolz, C. F., Cuni, A., FijaBkowski, M., Leuschel, M., Pedroni, S., & Rigo, A. (2011). Allocation removal by partial evaluation in a tracing jit. In *Proceedings*

of the 20th acm sigplan workshop on partial evaluation and program manipulation (pp. 43–52). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1929501.1929508 doi: 10.1145/1929501.1929508

Bolz, C. F., Cuni, A., Fijalkowski, M., Leuschel, M., Pedroni, S., & Rigo, A. (2011). 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* (pp. 9:1–9:8). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2069172.2069181 doi: 10.1145/2069172.2069181

Bolz, C. F., Cuni, A., Fijalkowski, M., & Rigo, A. (2009). Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the implementation, compilation, optimization of object-oriented languages and programming systems* (pp. 18–25). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1565824.1565827 doi: 10.1145/1565824.1565827

Bolz, C. F., Leuschel, M., & Schneider, D. (2010). Towards a jitting vm for prolog execution. In *Proceedings of the 12th international acm sigplan symposium on principles and practice of declarative programming* (pp. 99–108). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1836089.1836102 doi: 10.1145/1836089.1836102

Bolz, C. F., Pape, T., Siek, J., & Tobin-Hochstadt, S. (2014, Jun). Meta-tracing makes a fast racket. In *Dynamic languages and applications (dyla).*

Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., … Franz, M. (2009). Tracing for web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the 2009 acm sigplan/sigops international conference on virtual execution environments* (pp. 71–80). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1508293.1508304 doi: 10.1145/1508293.1508304

Deaver, D., Gorton, R., & Rubin, N. (1999). Wiggins/redstone: An on-line program specializer. In *Proceedings of the ieee hot chips xi conference.*

D'Hondt, T. (2015, February 17). *Programming language engineering.* Retrieved 2015-05-21, from http://soft.vub.ac.be/~tjdhondt/PLE/introduction.html

Dissegna, S., Logozzo, F., & Ranzato, F. (2014). Tracing compilation by abstract interpretation. In *Proceedings of the 41st acm sigplan-sigact symposium on principles of programming languages* (pp. 47–59). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2535838.2535866 doi: 10.1145/2535838.2535866

Felleisen, M. (1988). The theory and practice of first-class prompts. In *Proceedings of the 15th acm sigplan-sigact symposium on principles of programming languages* (pp. 180–190). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/73560.73576 doi: 10.1145/73560.73576

Felleisen, M., & Friedman, D. P. (1987). A calculus for assignments in higher-order languages. In *Proceedings of the 14th acm sigact-sigplan symposium on principles of programming languages* (pp. 314–). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/41625.41654 doi: 10.1145/41625.41654

Fijalkowski, M. (2013, February 28). *10 years of pypy.* Retrieved 2015-04-07, from http://morepypy.blogspot.be/2013/02/10-years-of-pypy.html

Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., ... Franz, M. (2009). Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th acm sigplan conference on programming language design and implementation* (pp. 465–478). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1542476.1542528 doi: 10.1145/1542476.1542528

Gal, A., Probst, C. W., & Franz, M. (2006). Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on virtual execution environments* (pp. 144–153). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1134760.1134780 doi: 10.1145/1134760.1134780

Guo, S.-y., & Palsberg, J. (2011, January). The essence of compiling with traces. *SIGPLAN Not.*, *46*(1), 563–574. Retrieved from http://doi.acm.org/10.1145/1925844.1926450 doi: 10.1145/1925844.1926450

Hayashizaki, H., Wu, P., Inoue, H., Serrano, M. J., & Nakatani, T. (2011, March). Improving the performance of trace-based systems by false loop filtering. *SIGARCH Comput. Archit. News*, *39*(1), 405–418. Retrieved from http://doi.acm.org/10.1145/1961295.1950412 doi: 10.1145/1961295.1950412

Homescu, A., & Şuhan, A. (2011, October). Happyjit: A tracing jit compiler for php. *SIGPLAN Not.*, *47*(2), 25–36. Retrieved from http://doi.acm.org/10.1145/2168696.2047854 doi: 10.1145/2168696.2047854

Inoue, H., Hayashizaki, H., Wu, P., & Nakatani, T. (2011). A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th annual ieee/acm international symposium on code generation and optimization* (pp. 246–256). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=2190025.2190071

Might, M. (2015). *Writing an interpreter, cesk-style.* Retrieved 2015-05-21, from http://matt.might.net/articles/cesk-machines/

Mitchell, J. G., Perlis, A. J., & Van Zoeren, H. R. (1967). Lc2: A language for conversational computing. In *Symposium on interactive systems for experimental applied mathematics: Proceedings of the association for computing machinery inc. symposium* (pp. 203–214). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2402536.2402558 doi: 10.1145/2402536.2402558

Pall, M. (2013, November 29). *Re: How does luajit's trace compiler work?* Retrieved 2015-04-08, from http://www.freelists.org/post/luajit/How-does-LuaJITs-trace-compiler-work,1

Paška, M. (2012, August). *Installing and using pypy standalone compiler with parlib framework* (Tech. Rep. No. DCSE/TR-2012-09). University of West Bohemia, Pilsen. Retrieved from http://www.kiv.zcu.cz/site/documents/verejne/vyzkum/publikace/technicke-zpravy/2012/tr-2012-09.pdf

Sangiorgi, D. (1998, October). On the bisimulation proof method. *Mathematical. Structures in Comp. Sci.*, *8*(5), 447–479. Retrieved from http://dx.doi.org/10.1017/S0960129598002527 doi: 10.1017/S0960129598002527

Schneider, D., & Bolz, C. F. (2012). The efficient handling of guards in the design of rpython's tracing jit. In *Proceedings of the sixth acm workshop on virtual machines and intermediate languages* (pp. 3–12). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2414740.2414743 doi: 10.1145/2414740.2414743

Smith, R. B., & Ungar, D. (1995). Programming as an experience: The inspiration for self. In *Proceedings of the 9th european conference on object-oriented programming* (pp. 303–330). London, UK, UK: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=646153.679530

Thomassen, E. W. (2013). *Trace-based just-in-time compiler for Haskell with RPython* (Unpublished master's thesis). Norwegian University of Science and Technology, Norway.

Ungar, D., & Smith, R. B. (1987, December). Self: The power of simplicity. *SIGPLAN Not.*, *22*(12), 227–242. Retrieved from http://doi.acm.org/10.1145/38807.38828 doi: 10.1145/38807.38828

Van Horn, D., & Might, M. (2010). Abstracting abstract machines. In *Proceedings of the 15th acm sigplan international conference on functional programming* (pp. 51–62). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1863543.1863553 doi: 10.1145/1863543.1863553

Wu, P., Hayashizaki, H., Inoue, H., & Nakatani, T. (2011). Reducing trace selection footprint for large-scale java applications without performance loss. In *Proceedings of the 2011 acm international conference on object oriented programming systems languages and applications* (pp. 789–804). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2048066.2048127 doi: 10.1145/2048066.2048127

Yermolovich, A., Wimmer, C., & Franz, M. (2009). Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on dynamic languages* (pp. 79–88). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1640134.1640147 doi: 10.1145/1640134.1640147