

SELF-HOSTING FUNCTIONAL PROGRAMMING LANGUAGES ON
META-TRACING JIT COMPILERS

Caner Derici

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University
Sep 2025

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Doctoral Committee

Sam Tobin-Hochstadt, Ph.D.

Jeremy Siek, Ph.D.

Daniel Leivant, Ph.D.

Amr Sabry, Ph.D.

Date of Defense: 09/05/2025

Copyright © 2025
Caner Derici

Dedicated to Muharrem & Ayşe Derici,
and to the entire Derici, Gül, and Taylor families—
may all their names live on through this work.

ACKNOWLEDGEMENTS

ACK

Write acknowledgements.

And the dedication in the previous page.

Caner Deric

SELF-HOSTING FUNCTIONAL PROGRAMMING LANGUAGES ON META-TRACING
JIT COMPILERS

Self-hosting represents a significant milestone in the evolution of a programming language, indicating its semantic maturity and offering practical advantages in portability and flexibility. While self-hosting is traditionally achieved through meta-circular interpreters or custom compilers, the viability of using meta-tracing Just-in-Time (JIT) compilers for this purpose in full-scale functional languages remains underexplored. This dissertation investigates this open question by developing an operational self-hosting full Racket implementation on Pycket, a meta-tracing JIT compiler generated on the RPython framework. We integrate Racket’s runtime subsystems into Pycket using linklets—formally specified compilation units that enable self-hosting. To support and clarify this integration, we present formal operational semantics for linklets, validated via both Racket & Pycket reference implementations and a PLT Redex model. Our evaluation identifies specific performance challenges fundamental to self-hosting on meta-tracing, such as overspecialized traces and increased Garbage Collection (GC) overhead. We propose and explore targeted approaches to address these challenges, including methods to guide the tracer away from inefficient trace generation and a hybrid evaluation model combining CEK and stack-based interpreters to reduce memory overhead. Beyond demonstrating the feasibility of efficient self-hosting on meta-tracing JIT compilers, our findings offer insights into language-runtime co-design, suggesting a general strategy for implementing high-level language features without compromising run-time performance, thereby opening new avenues for further research into efficient language implementation techniques.

Contents

Acknowledgements	v
Abstract	vi
List of Abbreviations	x
List of Figures	xi
List of Tables	xiii
1 INTRODUCTION	1
1.1 Thesis	2
1.2 Motivation & Context	2
1.3 Dissertation Structure & Contributions	4
2 META-TRACING JUST-IN-TIME (JIT) COMPILERS	6
2.1 Tracing vs Meta-tracing in JIT Compilation	9
2.2 Pycket Primer: A Rudimentary Interpreter Built on RPython Framework	13
2.3 Trace Optimizations & Runtime Feedback	19

3	PROGRAMMING LANGUAGES & PORTABILITY	23
3.1	Enriching Compiler & Runtime Communication	26
3.2	Linklets as Units of Compilation	29
3.3	Formal Operational Semantics of Linklets	36
4	FROM RUDIMENTARY INTERPRETER TO FULL LANGUAGE IMPLEMENTATION	46
4.1	Implementing Linklets on Pycket	48
4.2	Enhancing Run-time with Bootstrapping Linklets	52
4.3	Growing Pycket into Full Racket	65
5	PYCKET AS A FULL RACKET RUN-TIME: CORRECTNESS & COMPLETENESS	70
5.1	Correctness by Construction	73
5.2	Completeness under Self-Hosting	82
6	PERFORMANCE EVALUATION OF SELF-HOSTING ON META-TRACING	85
6.1	Pycket’s Performance Characteristics	87
6.2	Module and Language Loading	91
6.3	Cost of Self-Hosting	94
6.4	The Nature of the Beast: Tracing Data-Dependent Computation	100
6.5	Memory Pressure under Self-hosting	114
7	APPROACHES TO IMPROVE SELF-HOSTING PERFORMANCE	118
7.1	Guiding Tracer Away from Branch-Heavy Computation	121
7.2	Mitigating Memory Pressure in Branch-Heavy Computation	132
8	CONCLUSION & SIGNIFICANCE	139

8.1	Related Work	141
8.2	Future Work	145
Appendix A	Linklet Kernel Language Specification	148
Appendix B	Complete Reduction Steps for Top-level Example Program	150
Appendix C	PLT Redex Model for Linklet Semantics	153
Appendix D	Semantics for the CEK & Stackful Hybrid Model	170
References		178
Curriculum Vitae		

List of Abbreviations

ABI	Application Binary Interface
AOT	Ahead-of-Time
API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control-Flow Graph
FASL	Fast-Load Serialization
FFI	Foreign Function Interface
GC	Garbage Collection
GHC	Glasgow Haskell Compiler
IR	Intermediate Representation
JIT	Just-in-Time
LICM	Loop-invariant Code Motion
LKL	Linklet Kernel Language
LLVM	Low-Level Virtual Machine
OS	Operating System
PC	Program Counter
REPL	Read-Eval-Print Loop
SSA	Static Single Assignment
VM	Virtual Machine

List of Figures

2.1	A trace in a tracing JIT compiler is a linear sequence of machine code instructions.	8
2.2	Pycket’s CEK loop uses interpreter hints to provide runtime feedback to the meta-tracer.	12
2.3	The RPython framework translates a given interpreter code in RPython into a meta-tracing JIT compiler in C.	14
2.4	Pycket’s original front-end	16
2.5	The CEK machine for the λ -calculus [7].	18
2.6	A trace-elidable function in Pycket.	22
3.1	Comparison of Racket implementations. Figure used from [5]	27
3.2	Comparison of Racket implementations. Figure used from [5]	28
3.3	Linklet Source Language	30
3.4	Regular Instantiation of a Linklet	31
3.5	Example Top-level REPL Interaction to Implement Using Linklets	32
3.6	Internal steps in linklet instantiations for step-by-step Top-level REPL interactions.	34
3.7	Linklet Runtime Language	37
3.8	Linklet Program Source Language	40
3.9	Standard Reduction Relations for Operational Semantics of Linklets	41
3.10	Top-level REPL Interaction from Figure 3.5 in Linklets Formalism	42
4.1	Representation of Linklets and Linklet Instances on Pycket	49
4.2	Overview of Pycket exposing values from a linklet at boot.	53
4.3	Comparison of two Pycket front-ends. (A) The original front-end with offline expansion; (B) The new front-end with expansion in run-time.	54
4.4	Loading racket/base Modules	56
4.5	Welcome to Pycket!	66
4.6	Loading racket/base Modules Using Compiled Code	67
4.7	A Racket program that uses entire #lang racket	68
5.1	Example #lang racket program running on Pycket.	72
5.2	An example test case for linklet semantics	75
5.3	Restricted LKL expression grammar for testing	78
5.4	Initial checks before the evaluation begins	79
5.5	Meta-function converting the <i>program</i> form to Racket	81
5.6	Restricted Linklet <i>program</i> grammar for testing	82

6.1	Fully-expanded program evaluation performance. Pycket with Self-hosting vs Original Pycket, relative to Racket 8.18 — lower is better.	96
6.2	Example benchmark used in measuring (A) Back-end and (B) Front-end performance.	98
6.3	Program-expansion performance. Pycket with Self-hosting vs. Racket 8.18 — lower is better.	99
6.4	Branchy: a branch-heavy program designed to exhibit data-dependent behavior.	102
6.5	RPython trace inner loop for Branchy running all-zero input taking a long 18-8-3-1 path.	104
6.6	A small regular expression matching program.	108
6.7	Trace of Pycket’s regexp (in RPython) matching <code>#rx"defg"</code>	108
6.8	Trace of Pycket using the regexp linklet, matching <code>#rx"defg"</code>	110
6.9	Comparison of matching <code>#rx"defg"</code> against a large <code>"aaa...defg...aaa"</code> string — lower is better.	111
6.10	A new abstraction level brought by self-hosting.	112
6.11	Effect of nursery size on total GC wall-time for Pycket with self-hosting.	116
7.1	Branchy; from Figure 6.4 annotated with meta-hints.	125
7.2	Experiment results for targeted improvement approaches running Branchy. Lower is better.	127
7.3	Experiment results of regular expression matching with targeted improvement approaches. All interpreters use the regexp linklet. Lower is better.	130
7.4	Stackful & CEK Switch	133
7.5	Experiment results of CEK-only Pycket vs CEK+Stackful Pycket running Branchy. Lower is better.	135
7.6	Source Language for CEK & Stackful Hybrid Model	138
A.1	Linklet Kernel Language Grammar	148
A.2	Linklet Kernel Language standard reduction relation	149
A.3	Linklet Kernel Language evaluator	149
D.1	Source Language for CEK & Stackful Hybrid Model	170
D.2	CEK Reduction Relation	171

List of Tables

3.1	Transformation rules for linklet variables used in <code>compile-linklet</code>	38
3.2	Top-level REPL Example Reduction Steps using $\longrightarrow_{\beta_p}$	43
4.1	Sizes and Number of Exported Primitives of Bootstrapping Linklets	53
4.2	Some notable type mappings for <i>rktio</i> FFI layer	63
5.1	Summary of Pycket Test Suites	74
6.1	Loading <code>#lang racket/base</code> without using any compiled code	92
6.2	Loading <code>#lang racket/base</code> using compiled code	93
6.3	JIT back-end summaries for branch-heavy computations taking repeating vs random branches. Redacted fields are all zero for both cases.	106
6.4	Memory footprint of original Pycket vs Pycket hosting Racket. Nursery size: 32M	115
7.1	Memory footprint of CEK-only Pycket, Hybrid Pycket with CEK & Stackful, running Branchy with randomized input. Nursery size: 32M	136

1. INTRODUCTION

CHAPTER SYNOPSIS

Thesis:

Efficient self-hosting of full-scale functional languages on meta-tracing JIT compilers is achievable.

Contributions:

1. Pycket, the first self-hosting implementation of a full-scale functional programming language on a meta-tracing JIT compiler, serving as a complete run-time for Racket.
2. The first formalization for the operational semantics of linklet semantics
3. Rudimentary operational semantics for a hybrid computational model (CEK & Stackful) that utilizes both the native stack and the heap for optimized memory use.
4. Identification and analysis of performance issues fundamental to self-hosting on meta-tracing JIT compilers for future studies.

Sections:

- Motivation & Context
- Dissertation Structure & Contributions

1.1 Thesis

Efficient self-hosting of a full-scale functional language on a meta-tracing JIT compiler is achievable.

1.2 Motivation & Context

One significant milestone in the evolution of a programming language implementation is the capability to compile or interpret its own source code. This process, historically known as “meta-circular evaluation” [1], is now commonly referred to as bootstrapping, or as we will call it throughout this dissertation, self-hosting. Self-hosting requires the language to possess certain essential features, such as expressive completeness and a robust run-time system. Therefore, the ability to self-host indicates a language’s semantic maturity, broadly considered as a combination of expressive power and practical real-world applicability.

Self-hosting dynamic programming languages inherently involves challenging performance trade-offs, even in specialized research compilers such as Tachyon for JavaScript [2], and these challenges become increasingly complex for fully developed, real-world language run-times. Ideally, we would like a canonical method to implement efficient run-times for dynamic languages while retaining semantic maturity. This approach would enhance language portability, enabling practical benefits such as leveraging the unique capabilities of different virtual machines (VMs) for various computational tasks without the need to re-implement the language repeatedly. Such redundant re-implementations are not only error-prone but also complicate performance comparisons across different VMs.

Meta-tracing has proven to be an effective implementation technique for dynamic programming languages, as demonstrated by Bolz in his foundational work [3]. Unlike traditional meta-circular approaches, meta-tracing enables the semantics of a language to be specified through an interpreter, automatically generating a virtual machine (via C code) equipped with a specialized tracing JIT compiler. This method has been successfully applied in PyPy, a Python implementation that significantly outperforms standard CPython in runtime performance [4]. Thus, meta-tracing represents a promising strategy for achieving both ease of language implementation and efficiency at the same time.

Given the practical advantages of self-hosting and the demonstrated strengths of meta-tracing, we hypothesize that efficient self-hosting of a full-scale functional language on a meta-tracing JIT compiler is achievable. This dissertation explores this open research question by investigating whether a meta-tracing run-time system can indeed serve as an efficient virtual machine for a self-hosting, general-purpose, dynamic functional programming language.

We choose Racket as our main language for demonstrating this hypothesis. Racket is a particularly suitable candidate because it is reasonably well-adopted, explicitly designed to facilitate programming language research, and has recently enhanced its portability by migrating its core run-time to Chez Scheme. This transition has significantly simplified targeting different virtual machines, making Racket an ideal language for exploring efficient self-hosting on a meta-tracing JIT compiler [5].

Additionally, Racket has an existing, although rudimentary, meta-tracing interpreter named Pycket, developed as a research project using the RPython framework—the same meta-tracing

framework on which PyPy was built [3, 6]. Pycket has already demonstrated notable performance improvements and has been shown to eliminate 90% of the performance overhead associated with gradual typing [7, 8]. These characteristics position Pycket as an excellent candidate for this study, providing a solid starting point for validating our hypothesis.

To substantiate our hypothesis, we first enhance Pycket from a simple interpreter to a full-scale virtual machine capable of hosting Racket. Leveraging Racket’s improved self-hosting capabilities, we demonstrate concretely that an efficient, self-hosting implementation of a full-scale functional language on a meta-tracing JIT compiler is achievable. This transformation provides clear evidence in support of our primary thesis statement.

During this process, we identify and analyze several performance issues fundamental to self-hosting languages using meta-tracing compilers. While these issues pose challenges, we provide evidence that such performance limitations can be effectively addressed and mitigated, reinforcing our claim that efficient self-hosting on meta-tracing JIT compilers is feasible.

1.3 Dissertation Structure & Contributions

This dissertation is structured as follows. We start by examining meta-tracing JIT compilers in Chapter 2, providing a primer on Pycket and its underlying RPython framework. We then analyze the concept of portability in programming languages in Chapter 3, giving a detailed explanation of Racket’s key innovation that significantly improved its portability.

In Chapter 4, we describe our methodology and detail the steps taken to achieve a full-scale, self-hosting implementation of Racket on Pycket, addressing the first part of our thesis statement.

We validate the correctness and completeness of our resulting system in Chapter 5, confirming that Pycket indeed functions as a complete Racket implementation.

Next, we address performance considerations in Chapter 6. We present a thorough evaluation of self-hosting on meta-tracing JIT compilers, identifying and analyzing critical performance bottlenecks that inherently arise in this approach. In Chapter 7, we explore various strategies to overcome these performance challenges. Through concrete evidence, we demonstrate that despite these fundamental limitations, self-hosting on a meta-tracing compiler can be efficiently realized.

Finally, we conclude with a discussion of related work and broader implications of our findings in Chapter 8.

The primary technical contributions of this dissertation are:

- Pycket, the first self-hosting implementation of a full-scale functional programming language on a meta-tracing JIT compiler, serving as a complete run-time for Racket.
- The first formalization for the operational semantics of linklet semantics, to clearly reason about their behavior.
- Rudimentary operational semantics for a hybrid computational model (CEK & Stackful) that utilizes both the native stack and the heap in an intelligent way for optimized memory use.
- Identification and thorough analysis of performance issues fundamental to self-hosting on meta-tracing JIT compilers.

2. META-TRACING JUST-IN-TIME (JIT) COMPILERS

CHAPTER SYNOPSIS

We do three things in this chapter:

1. Thesis statement refers to meta-tracing JITs, so we explain what a meta-tracing JIT compiler is.
2. Introduce RPython as a language, and as a framework that Pycket is built on.
3. Provide context for meta-tracing-related technical discussions in the rest of the study. traces, warmup, optimizations, runtime feedback, etc.

Sections:

- Meta-tracing & RPython Framework

What problem does PyPy solve? How do they capture loops in user programs? JIT drivers, interpreter hints, green/red variables etc.

- Pycket: A Rudimentary Racket Interpreter Built on RPython Framework

Primer on Pycket's fundamentals. It's language RPython, CEK core, and how it is built.

- Trace Optimizations & Runtime Feedback

Traces are the real performance currency. Relevant optimizations and runtime feedback (e.g promote, escape analysis, warmup, etc.).

In this chapter, we have three main objectives. First, we explain the concept of a meta-tracing JIT compiler, clearly connecting it to our thesis statement presented earlier, that efficient self-hosting of full-scale functional languages is achievable using meta-tracing JIT compilation. Second, we introduce the RPython framework, detailing its role as the foundational technology upon which Pycket is constructed, including aspects of translation and toolchain workflow. Lastly, we outline key concepts related to RPython and meta-tracing—such as trace formation, optimization strategies, and runtime feedback—which serve as essential context for technical discussions and evaluations relating to performance in Chapter 6 and Chapter 7.

A JIT compiler dynamically compiles frequently executed parts of a program at runtime, interleaving compilation with interpretation. Unlike Ahead-of-Time (AOT) compilers, which compile the entire program beforehand, JIT compilers selectively target hot paths identified through low-overhead profiling. When the interpreter repeatedly executes a particular sequence of instructions, the JIT compiler pauses interpretation momentarily, compiles and optimizes this sequence, and subsequently executes the optimized code whenever the same instruction path recurs [9].

JIT compilers have proven particularly effective for dynamic language virtual machines (VMs), generally adopting either method-based or trace-based approaches. Method-based compilers optimize frequently invoked methods, whereas trace-based compilers focus on frequently executed loops [10, 11]. This dissertation specifically focuses on trace-based compilation. Tracing JITs generate optimized machine code by tracing and compiling execution paths under two fundamental assumptions[4]:

1. Programs spend most of their execution time in loops.

2. Iterations of the same loop often follow similar execution paths.

<i># start of the trace (preamble)</i>	
label(p0, p1)	1
guard_not_invalidated()	2
guard_class(p0, ConsEnv)	3
p3 = getfield_gc_r(p0, ConsEnv.prev)	4
guard_class(p3, ConsEnv)	5
i5 = getfield_gc_i(p3, Fixnum)	6
i6 = getfield_gc_i(p0, Fixnum)	7
i7 = int_add_ovf(i5, i6)	8
guard_no_overflow()	9
guard_class(p1, LetCont)	10
p9 = getfield_gc_r(p1, LetCont.ast)	11
guard_value(p9, ConstPtr(ptr10))	12
p11 = getfield_gc_r(p1, LetCont.env)	13
p12 = getfield_gc_r(p1, LetCont.prev)	14
<i># peeled-iteration (inner loop)</i>	
label(p11, i7, p12, "64723392")	15
guard_not_invalidated()	16
guard_class(p11, ConsEnv)	17
i14 = getfield_gc_i(p11, Fixnum)	18
i15 = int_add_ovf(i14, i7)	19
guard_no_overflow()	20
guard_class(p12, LetCont)	21
p17 = getfield_gc_r(p12, LetCont.ast)	22
guard_value(p17, ConstPtr(ptr18))	23
p19 = getfield_gc_r(p12, LetCont.env)	24
p20 = getfield_gc_r(p12, LetCont.prev)	25
jump(p19, i15, p20, "64723392")	26

Figure 2.1: A trace in a tracing JIT compiler is a linear sequence of machine code instructions.

To exploit these assumptions, tracing JIT compilers identify certain execution paths as *hot loops*. Upon detecting a hot loop, the interpreter pauses evaluation to compile this loop into a *trace*, subsequently using the optimized trace whenever the same path is executed again. A trace is a linear sequence of instructions with a single entry and potentially multiple exit points. Figure 2.1 illustrates a simplified trace, with inputs $p0$ and $p1$, comprising a preamble (due to loop unrolling) and an inner loop that jumps back to itself. *Guards* within a trace define its exit

points, performing runtime checks to ensure that conditions remain consistent with the initial tracing context and handling conditions for loop termination or deviation.

2.1 Tracing vs Meta-tracing in JIT Compilation

When a tracing JIT compiler is applied to a language interpreter, it typically ends up tracing the loops of the interpreter itself—rather than the loops in the user program that the interpreter is executing. To understand why this is a problem, we need to distinguish three roles: the language interpreter (the implementation of the language semantics), the tracing interpreter (or meta-tracer), and the user program (the input being evaluated by the language interpreter). The meta-tracer effectively executes the language interpreter step-by-step, carrying out its operations and observing their effects. Without additional guidance, it has no visibility into the boundaries or semantics of the user-level code being interpreted, and so it naturally captures hot paths in the language interpreter logic rather than in the user program.

Within a language interpreter, the primary dispatch loop is typically the most significant loop. For a functional programming language, this usually manifests as a recursive descent loop, evaluating sub-expressions iteratively. However, while evaluating a user program, this loop structure in the language interpreter fundamentally violates one of the core assumptions of tracing JIT compilers—that several iterations of a hot loop tend to follow similar code paths—because it is uncommon for an interpreter to repetitively evaluate the exact same operation or expression in succession. Rather, loops in the user program often unfold as different branches taken through the language interpreter’s dispatch logic.

Furthermore, by nature, tracing captures the execution paths where a program spends most of its running time. In the scenario of applying a tracing JIT to a language interpreter, however, the interpreter’s dispatch loop itself is typically not the most essential computational path. Instead, the loops within the user program represent the true computational hot spots. Thus, what we seek from a tracing JIT in this context is to identify and optimize the loops of the user program, not the loops within the interpreter. This is precisely the problem addressed by *meta-tracing*, as defined in PyPy: it successfully redirects the tracing JIT compiler to identify and optimize the hot loops of the user program, rather than the interpreter logic itself [4].

A loop in the context of execution is essentially a backward jump to an instruction or a logical position (e.g., a previously seen program counter). The meta-tracer, lacking any intrinsic semantic knowledge of the language interpreter it is running, cannot inherently detect such loops occurring within the interpreted user program. To overcome this, meta-tracing frameworks expose an Application Programming Interface (API) through which the language interpreter explicitly communicates the occurrence and location of loops within the user program being interpreted. Specifically, the interpreter defines a logical *program counter* composed of interpreter-specific variables. For instance, a bytecode interpreter might represent this counter by combining the current bytecode index with other execution-specific information. When the meta-tracer repeatedly observes the same logical program counter value, it infers the presence of a loop within the user program.

For more fine-grained control of the traced execution paths, the language interpreter employs a set of annotations and hints to guide the meta-tracer as it executes the user program. Two of the most crucial annotations are *jit_merge_point*, marking stable loop headers, and *can_enter_jit*,

indicating potential backward jumps. The logical program counter used for detecting loops is built from what are known as *green* variables, which remain constant across multiple iterations, while other interpreter state variables that may change frequently are classified as *red* variables. Both sets of variables, along with these hints, are registered with a *JitDriver* reflection that provided by the RPython framework, an object acting as a mediator between the interpreter and the global meta-tracer. An interpreter may define multiple JitDrivers, each with its own logical compound program counter, although there remains a single unified meta-tracer. We discuss this interplay further in Section 7.1, when addressing advanced trace optimizations.

In this dissertation, we will study a concrete instance of a language interpreter built using meta-tracing: Pycket, an implementation of the Racket language built on the RPython framework. We will introduce Pycket in detail in the following section. Due to the use of meta-tracing techniques, Pycket’s tracing JIT compiler specifically captures and optimizes hot loops found within the user-level Racket code, rather than within its own interpreter logic. To understand this clearly, we will now examine concretely how an interpreter communicates loop information to a meta-tracer, as illustrated by Figure 2.2.

In this dissertation, we focus primarily on Pycket, an interpreter for the Racket language initially developed as a rudimentary implementation using RPython’s meta-tracing framework. Throughout the study, we progressively transform Pycket from this initial interpreter into a full-scale implementation of Racket, specifically by self-hosting Racket on top of Pycket itself. This evolution is facilitated by the meta-tracing approach, as Pycket’s tracing JIT effectively identifies and optimizes hot loops in the user-level Racket code, rather than within the CEK interpreter logic. To understand concretely how such loop detection and optimization occurs, we

next illustrate how the interpreter informs the meta-tracer about loops, as depicted in Figure 2.2, as an example usage of meta-tracing hints.

```
driver_two_state = jit.JitDriver(reds=["env", "cont"],           1
                                greens=["ast", "came_from"])    2
                                                                3
def inner_interpret_two_state(ast, env, cont):                  4
    came_from = ast                                           5
    while True:                                                6
        driver_two_state.jit_merge_point(ast=ast, came_from=came_from, env=env, cont=cont) 7
        came_from = ast if isinstance(ast, App) else came_from 8
        t = type(ast)                                         9
                                                                10
        if t is Let:                                          11
            ast, env, cont = ast.interpret(env, cont)         12
        elif t is If:                                         13
            ast, env, cont = ast.interpret(env, cont)         14
        elif t is Begin:                                       15
            ast, env, cont = ast.interpret(env, cont)         16
        else:                                                 17
            ast, env, cont = ast.interpret(env, cont)         18
        if ast.should_enter:                                   19
            driver_two_state.can_enter_jit(ast=ast, came_from=came_from, env=env, cont=cont) 20
```

Figure 2.2: Pycket’s CEK loop uses interpreter hints to provide runtime feedback to the meta-tracer.

Concretely, the Pycket interpreter defines a dedicated `JitDriver`, explicitly specifying green and red variables to inform the meta-tracer about user-level loops. In Pycket’s CEK machine, the green variables consist primarily of the AST node currently being evaluated and a `came_from` indicator, which together help encode recursive calls—the only way to make loops. Pycket’s interpreter further employs annotations such as `jit_merge_point` and `can_enter_jit` to explicitly mark stable loop entry points and potential backward jumps, respectively. These details are shown in Figure 2.2, and we will further examine their operation in detail in the following section (Section 2.2).

Having described Pycket’s use of meta-tracing annotations, we now transition to exploring its internal core architecture. Specifically, we will discuss how its interpreter is built around the CEK abstract machine, a foundational design component that remains constant throughout our study—even as we modify and extend Pycket’s broader architecture during the transition to a self-hosted Racket implementation.

2.2 Pycket Primer: A Rudimentary Interpreter Built on RPython Framework

Pycket was initially developed in 2014 as a high-performance tracing JIT compiler for the Racket language. From the start, Pycket aimed at efficiently supporting advanced Racket features such as contracts, continuations, structures, and dynamic binding, demonstrating significant performance improvements over existing compilers at the time, including Racket’s own JIT compiler [7]. It was later shown that Pycket could remove nearly 90% of the overhead associated with sound gradual typing [8]. In this section, we provide a primer on Pycket’s core fundamentals, focusing on the RPython framework upon which it is built, its CEK-based abstract machine, and the process of running Racket programs.

RPython (Restricted Python) is a statically typed, object-oriented subset of Python, originally created during the development of the PyPy project [12]. By restricting Python’s dynamic features, RPython enables type inference on programs, allowing efficient compilation to lower-level languages like C. Notably, RPython disallows mixing variable types at the same program location, ensuring all types are consistent and inferable. It also prohibits runtime reflection (such as modifying classes at runtime), closures, and assumes global and class-level bindings are constants [13, 14].

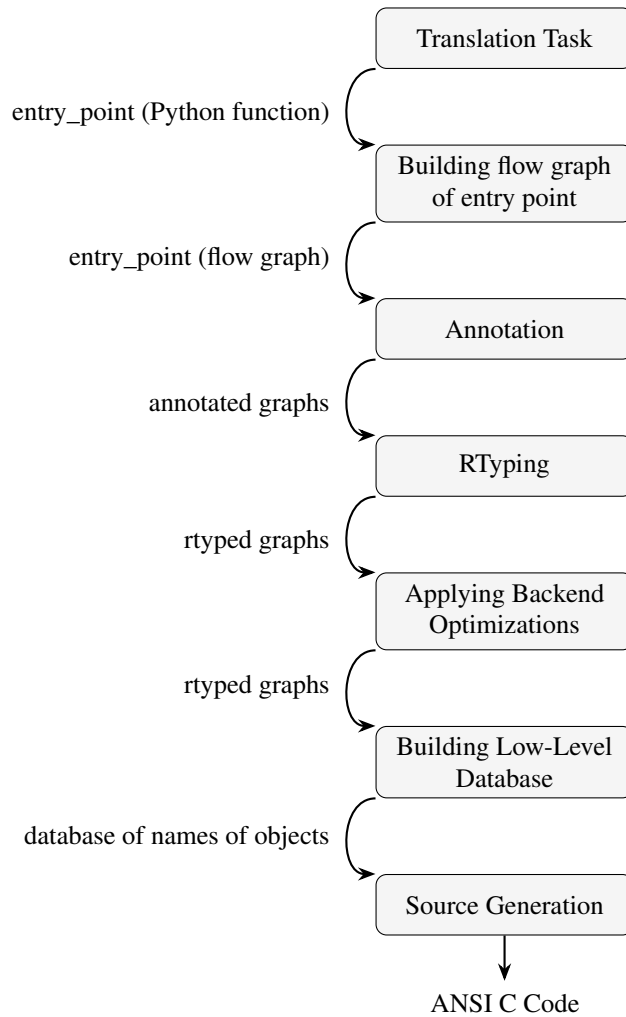


Figure 2.3: The RPython framework translates a given interpreter code in RPython into a meta-tracing JIT compiler in C.

The RPython framework is a toolchain designed to translate interpreters written in RPython into efficient low-level languages, typically C. As illustrated in Figure 2.3, this translation involves several phases, each with its own type system and type inference mechanism. Initially, the RPython program is loaded into a Python runtime environment, and Control-Flow Graphs (CFGs) are generated through abstract interpretation. Next, the annotator infers types and

annotates variables at each CFG node, reflecting possible Python object types. Following annotation, the RTyper (RPython typer) lowers annotations and operations into types compatible with the target language, acting as a bridge to code generation. Subsequent optional backend optimizations, such as inlining, malloc removal, and escape analysis, refine the resulting code. Because RPython programs assume automatic memory management, a garbage collector is inserted into the program before final translation into a binary executable [13, 12, 15].

Pycket was built using this RPython framework. In its initial design, shown in Figure 2.4, Pycket utilized the Racket executable to read and expand modules into fully expanded code, after which Pycket itself converted the expanded code into its own AST representation for execution within its interpreter loop [16, 6, 7]. This interpreter is based on the CEK abstract machine, defined by the state triple $\langle e, \rho, \kappa \rangle$ (where e is the AST representing the control, ρ is the environment mapping variables to values, and κ is the continuation, capturing the context of the computation) [17].

In its original design, Pycket leverages Racket’s macro expander [18] to preprocess macros, simplifying input programs into a small set of core forms that Pycket’s runtime can directly handle [16]. Initially, this macro expansion is performed externally by invoking the Racket executable, which expands modules and serializes them into JSON-encoded files for Pycket to read, convert into ASTs, and evaluate directly. As we will discuss in detail in Chapter 3, we shift the macro expansion process into Pycket’s runtime itself by importing Racket’s expander linklet into Pycket’s runtime. This shift is the key step that enables Pycket to become a fully self-hosted and independent execution environment for Racket.

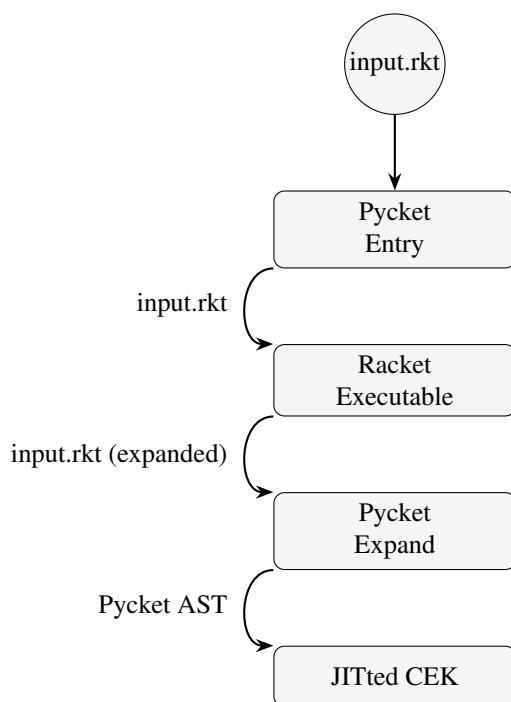


Figure 2.4: Pycket’s original front-end

Once a module is expanded to core Racket forms, Pycket performs two transformations on its AST. First, it converts the AST into A-normal form (ANF), ensuring that all complex expressions are explicitly named, simplifying subsequent evaluation and optimization [19, 20]. Second, all mutable variables targeted by `set!` operations are transformed into heap-allocated cells. This approach, common in Lisp-family languages, enables environments to be immutable mappings from variables to values and allows each AST node to carry its static environment explicitly [7].

Pycket implements approximately 1500 of Racket’s more than 2000 primitive operations and values. These primitives range from numeric computations, including bignums, rationals, and complex numbers, to regular expression handling, and sophisticated input/output abstractions.

A substantial portion of Pycket’s implementation consists of these primitives, which are further extended by bootstrapping linklets discussed in Chapter 3.

Pycket implements approximately 1500 of Racket’s more than 2000 primitive operations and values. These primitives range from numeric computations, including bignums, rationals, and complex numbers, to sophisticated input/output abstractions and regular expression handling. Some of these primitive sets, notably input/output and regular expressions, are subsequently replaced by implementations supplied directly from Racket through bootstrapping linklets, as discussed in Chapter 3. This approach allows Pycket to extend its runtime functionality using primitives provided as Racket code by the language itself.

Additionally, Pycket employs some advanced techniques such as hidden classes, a technique borrowed from prototype-based object systems, to implement proxies efficiently. Proxies, crucial for enforcing soundness in gradual type systems and Racket’s contract systems, benefit significantly from Pycket’s approach, enhancing performance in both gradually typed and contract-heavy programs [21, 8]. Moreover, Pycket naturally benefits from type specialization through tracing, eliminating the need for a separate type-feedback pass required by JITs such as method-based JIT compilers [22].

After the user program is fully converted into its final AST form, Pycket evaluates it using the CEK abstract machine, whose semantics are given by the transition rules in Figure 2.5. A CEK state is represented as $\langle e, \rho, \kappa \rangle$, where e is the expression currently being evaluated (the control), ρ maps variable identifiers to their associated values (the environment), and κ holds the computation context (the continuation). The continuation consists of frames that track pending

$$\begin{aligned}
e &::= x \mid \lambda x. e \mid e e \\
\kappa &::= [] \mid \mathbf{arg}(e, \rho) :: \kappa \mid \mathbf{fun}(v, \rho) :: \kappa
\end{aligned}$$

$$\begin{aligned}
\langle x, \rho, \kappa \rangle &\longmapsto \langle \rho(x), \rho, \kappa \rangle \\
\langle (e_1 e_2), \rho, \kappa \rangle &\longmapsto \langle e_1, \rho, \mathbf{arg}(e_2, \rho) :: \kappa \rangle \\
\langle v, \rho, \mathbf{arg}(e, \rho') :: \kappa \rangle &\longmapsto \langle e, \rho', \mathbf{fun}(v, \rho) :: \kappa \rangle \\
\langle v, \rho, \mathbf{fun}(\lambda x. e, \rho') :: \kappa \rangle &\longmapsto \langle e, \rho'[x \mapsto v], \kappa \rangle
\end{aligned}$$

Figure 2.5: The CEK machine for the λ -calculus [7].

evaluations, which come in two forms: an $\mathbf{arg}(e, \rho)$ frame captures an argument that is awaiting evaluation, and a $\mathbf{fun}(v, \rho)$ frame represents a function that is ready to receive its argument. Evaluating a variable involves simply looking up its value in the environment. Applying a function involves two steps—first evaluating the function itself and then evaluating its argument. The final step applies the evaluated function to its evaluated argument. Importantly, the CEK machine creates no additional continuation frames for tail calls, thus ensuring proper tail-calls [7].

The corresponding RPython code implementing the CEK loop (Figure 2.2) on Pycket continuously transforms CEK triples $(ast, env, cont)$ into new states by invoking the `interpret` method on the current AST node. This iterative process proceeds until an empty continuation is encountered, at which point a `Done` exception is triggered, storing and returning the computation’s final result.

Pycket help the meta-tracer identify loops using two complementary mechanisms: the *two-state tracking* and a dynamic *call-graph* approach. In two-state tracking, Pycket filters out false loops by associating potential loop points with caller-callee pairs, represented by the current and previous AST nodes. Without additional context, the JIT may incorrectly recognize various control-flow points as loops, and the two-state approach effectively reduces such false positives. However, this mechanism alone struggles with complex recursive patterns, such as indirect recursion or proxy-based calls. To complement two-state tracking, Pycket employs a dynamic call-graph technique. This method incrementally constructs a call graph at runtime, with nodes representing invoked functions and edges representing caller-callee relationships. Whenever adding a new edge creates a cycle, Pycket recognizes the target function as a loop header suitable for tracing. The combination of these two approaches ensures robust loop detection and optimized trace generation, particularly in programs with intricate control flows involving recursion and higher-order functions [7, 8].

While we will explore Pycket’s performance characteristics further in Section 6.1, we first turn to relevant RPython trace optimizations, which will provide the context necessary for a detailed technical discussion about performance considerations in Chapter 6.

2.3 Trace Optimizations & Runtime Feedback

Traces are central to the performance characteristics of tracing JIT compilers. Good-quality traces directly influence the efficiency and effectiveness of JIT compilation. Quality here implies that traces capture frequently executed execution paths clearly and succinctly, avoiding excessive guards or unnecessary complexity. For example, traces that are tighter—having fewer

exits—maximize the value derived from compiled machine code and facilitate other optimization opportunities, such as allocation removal. In contrast, poor-quality traces, characterized by multiple exit points or redundant control-flow checks, can introduce significant performance penalties, leading to frequent trace invalidations or costly fallbacks to interpreter execution. [23]

An important aspect affecting trace quality and overall JIT performance is warmup time. Warmup refers to the initial period during program execution when the JIT compiler is actively identifying and tracing hot loops. Initially, execution proceeds more slowly due to overheads from trace recording and compilation, as well as running interpreted code. Once the JIT compiler has completed its initial tracing and optimized key loops, execution starts to transition from interpreted paths to increased use of optimized compiled code, achieving a significant speedup. Thus, warmup time is the duration required for this transition, and minimizing it is critical for overall runtime performance.

RPython’s trace optimizer incorporates several classic compiler optimizations tailored specifically for traces, including common sub-expression elimination, constant folding, copy propagation, and dead-code elimination. One inherent advantage of trace-based compilation for optimization purposes is the linear nature of traces. Unlike general control-flow graphs, a trace’s straight-line structure simplifies analysis and optimization, typically requiring only forward and backward passes through the trace. This streamlined approach ensures efficient optimization, minimizing compilation overhead while maximizing runtime performance [24].

Inlining is a key optimization strategy significantly enhances performance in tracing JIT compilers, such as those provided by RPython. During trace recording, function calls encountered along the traced path are naturally inlined into the trace itself. This includes both high-level language functions and lower-level runtime routines. This aggressive inlining eliminates function call overhead, crosses abstraction level borders, thus significantly simplifies the execution path, and creates additional opportunities for further optimizations, such as constant propagation and automatic dead-code elimination [25, 4].

Loop-invariant code motion is another effective optimization applied by RPython’s trace optimizer. Due to the linear nature of traces, loop-invariant computations are easily identified and relocated outside the loop. RPython accomplishes this by peeling off a single iteration from the loop, performing standard forward analyses, and optimizing the peeled iteration separately from the loop body [24]. Additionally, allocation removal is particularly relevant to Pycket’s CEK-based interpreter, which involves frequent allocations of state components such as continuations and environments. By analyzing object lifetimes and usages within traces, the optimizer can eliminate many heap allocations, significantly reducing memory pressure and improving runtime performance. We provide an in-depth analysis of memory performance in Section 6.5.

Promotion is a specialized trace-optimization technique used by RPython that converts runtime variable values into compile-time constants within traces. This optimization explicitly introduces guards, such as `guard(x == 0)`, to assert runtime invariants, thereby enabling the compiler to treat the guarded variables as constants in subsequent instructions within the trace. Although the original source code might not contain explicit constants, promotion allows the op-

timizer to leverage runtime information, effectively specializing traces for frequently occurring execution contexts. Promotion significantly enhances the trace specialization capability of the optimizer, leading to improved runtime performance through aggressive constant propagation and folding [3].

```
@jit.elidable          1
def _is_ascii_elidable(s): 2
    for c in s:           3
        if ord(c) >= 128: 4
            return False  5
    return True           6
```

Figure 2.6: A trace-elidable function in Pycket.

Escape analysis and the notion of trace-elidable functions further enhance trace optimizations in RPython. A function is deemed trace-elidable if successive calls with identical arguments during program execution yields the same result. Naturally, such a function should be free of any side-effects. Annotating such functions with the `@elidable` decorator allows the optimizer to replace repeated calls with cached results, effectively performing trace-level memoization. Figure 2.6 provides an example of such an elidable function in Pycket. By replacing repeated function calls within traces with previously computed results, this optimization substantially reduces runtime overhead, further streamlining trace execution and improving overall performance [3].

3. PROGRAMMING LANGUAGES & PORTABILITY

CHAPTER SYNOPSIS

Making programming languages more portable is good, and here's a way to do that.

Sections:

- Enriching Compiler & Runtime Communication

Motivation for linklets. Making programming languages more portable is good.

- Linklets as Units of Compilation

Operational semantics for linklets, both formalism and PLT Redex model.

Linklets are good tools to improve communication between the compiler and runtime.

- Formal Operational Semantics of Linklets

Formal specification of the operational semantics of linklets. Form grammar, compilation semantics, reduction relation, the works.

A practical programming language typically consists of three essential components: first, a *language grammar* defining the syntax—expressions that can be formed; second, *core language semantics*, which determine the meaning of these expressions, typically via reduction rules mapping expressions to values; and third, a *runtime* responsible for realizing these semantics on a computing environment. The runtime provides mechanisms to read and expand user expressions, evaluate them, and produce values, effectively bridging the gap between abstract language semantics and concrete execution.

The *portability* of a programming language refers to its ability to be adapted to run on different runtimes or virtual machines. This notion is distinct from having multiple independent implementations of a language. Rather, portability implies deliberate support at the language’s syntax or semantic level, enabling runtimes to more easily adopt the language’s core implementation. Languages that explicitly expose an intermediate form or semantics conducive to plugging into different backend runtimes exemplify this approach. Notable examples include Haskell, whose Glasgow Haskell Compiler (GHC) toolchain reduces every module to an intermediate typed Core IR, facilitating easy backend integration such as Low-Level Virtual Machine (LLVM) [26]; and Common Lisp, particularly implementations like Clasp, which retain high-level runtime elements (macro expander, reader) in Lisp while swapping only the code generator [27]. Such designs demonstrate how exposing an intermediate representation at the language level significantly simplifies retargeting efforts, reducing engineering overhead.

Future-proofing and platform agility When language semantics are lowered to a portable, implementation-independent Intermediate Representation (IR), targeting new Virtual Machines (VMs) typically involves writing a minimal backend rather than a complete compiler. For

instance, the GHC gained support for ARM, PowerPC, and WebAssembly virtually “for free” upon integrating LLVM [26]. Reducing engineering costs per new target ensures the language remains viable across evolving hardware ecosystems, preserving the investments made by implementers and users.

Tooling & ecosystem leverage A standardized, information-rich IR allows external tools—such as optimizers, verifiers, and profilers—to uniformly support all eventual backends. Compiler research demonstrates that most optimization efforts naturally gravitate toward the IR layer because benefits propagate globally [28]. Recent advances, such as the SMACK verifier, illustrate that languages emitting LLVM IR immediately gain powerful, cross-language verification capabilities without rewriting analysis tools for each source language [29]. Hence, language-level portability not only expands the range of deployment targets but also enhances the reuse potential of infrastructure and tooling around the language.

Improving language portability fundamentally relies on enhancing semantic communication between the language’s core semantics and its runtime. Clear, explicit interfaces or intermediate representations are required for languages to effectively instruct runtimes on realizing their semantics, rather than each runtime manually handling tight couplings such as macro expansion, serialization, or user-level threading models.

In this chapter, we examine improvements to language portability, using Racket’s linklets as a concrete example. We introduce a formal specification along with an executable PLT Redex model to describe the operational semantics of linklets. This formalism forms the basis for understanding how self-hosting is achieved in the meta-tracing JIT-compiled Pycket

implementation described in Chapter 4, supporting the main thesis statement.

3.1 Enriching Compiler & Runtime Communication

A tight coupling exists between the runtime and the reduction semantics of a programming language. This coupling arises because the runtime must concretely implement the abstract semantics specified by the language. For example, languages with macro systems may require runtimes to adopt specialized parsing mechanisms to handle macro expansions, differing from standard reading approaches. Similarly, custom serialization semantics defined by a language necessitate that the runtime can correctly serialize and deserialize language-specific objects. Additionally, languages that define semantics for user-level (green) threads demand explicit runtime support for mapping these constructs to Operating System (OS) threads. Clearly specifying these interactions is essential for streamlined and efficient language implementations.

Without explicit support from the language itself, each runtime that aims to implement the language must manually reconstruct all of these intricate interactions. Such manual reconstruction is costly, error-prone, and challenging to maintain, especially since subtle changes in language semantics often require extensive adjustments within the runtime. Consequently, portability becomes limited, forcing each runtime to independently handle the complexities inherent in language semantics. Therefore, introducing a mechanism to mitigate these difficulties would significantly improve portability.

Racket is a good example that successfully leverages such a mechanism to significantly enhance its portability. In 2019, Racket transitioned to Chez Scheme as its primary runtime and introduced a structured intermediate representation called *linklets* as units of compilation,

designed to simplify the runtime implementation [5]. Linklets, discussed in detail in the next section, streamlined the process of re-targeting Racket to Chez Scheme and established a robust foundation for self-hosting on other platforms, directly aligning with our thesis.

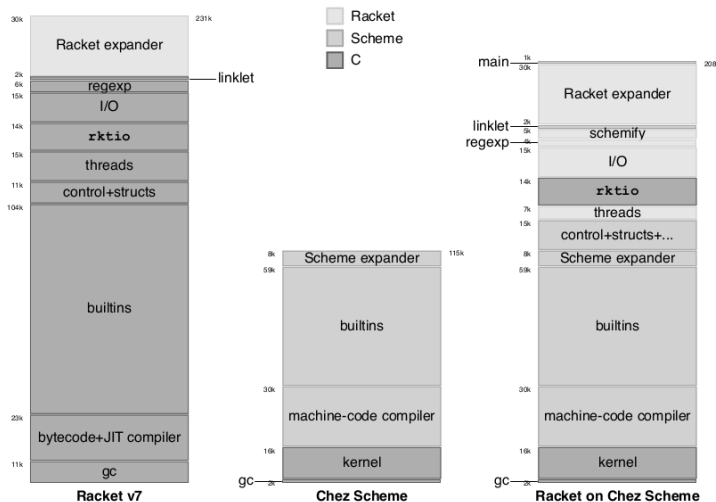


Figure 3.1: Comparison of Racket implementations. Figure used from [5]

The transition of Racket to Chez Scheme is illustrated in Figure 3.1. To concretely improve portability, Racket shifted several high-level components—initially and most notably the macro expander—from a C-based implementation to Racket itself. The left-most column of Figure 3.1 highlights the first ever version of Racket with its expander implemented in Racket. The macro expander defines critical functionality, including the macro system, module system, reader, and code expansion and elaboration. It is the essential component enabling reading and elaborating Racket code, thereby it plays a central role in supporting self-hosting of the language.

Furthermore, the expander transforms Racket code into a core IR designed explicitly for consumption by the hosting runtime. This intermediate representation closely resembles λ -

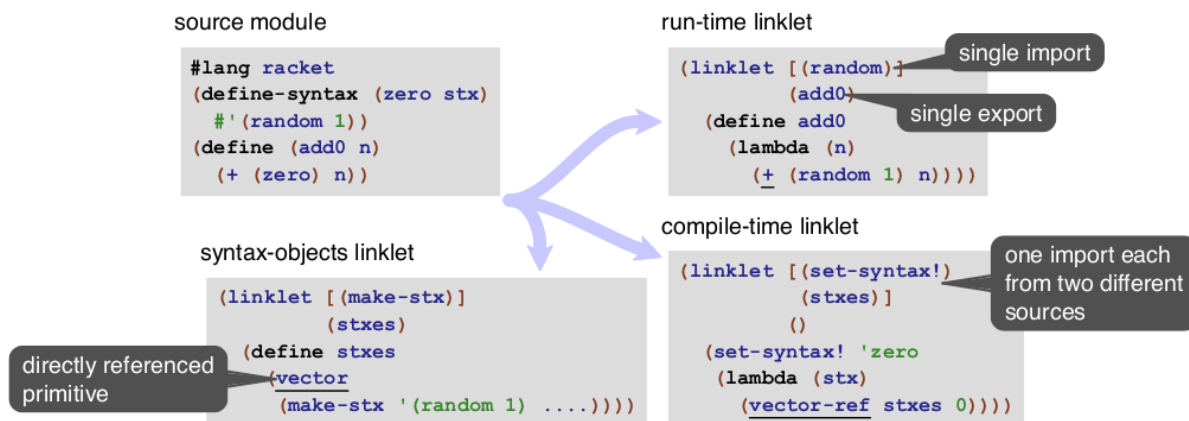


Figure 3.2: Comparison of Racket implementations. Figure used from [5]

calculus with some syntactic extensions. Instead of conventional lambda forms, which consume and produce values, this intermediate form employs linklet forms that consume and produce potentially mutable variables. By strictly segregating evaluation phases (e.g., compile-time versus run-time), the expander produces distinct linklets corresponding to different phases for a given Racket module. Figure 3.2 demonstrates this elaboration with an example module, showing distinct linklets for expansion-time (compile-time), run-time, and literal syntax objects to bridge these phases.

The bundle of linklets generated for a Racket module can subsequently be flattened into a single, self-contained runtime linklet. This way, Racket applies the macro expander to each layer of high-level functionality to produce self-contained linklets that can be exported offline by the language. For example, applying the macro expander to itself produces an independent linklet that encapsulates all subsystems essential for bootstrapping Racket, such as the module and macro systems. We will refer to the high-level functionalities that are exported in this way together as the *bootstrapping linklets* in the rest of the dissertation.

Given the bootstrapping linklets, a runtime that aims to implement Racket needs only to understand how to compile and execute serialized linklets to turn into a full runtime for the language. By loading these linklets, a runtime instantly acquires Racket’s entire high-level functionalities, such as the macro and module system, `read`, `expand`, and more without having to explicitly reimplement them separately. Consequently, it gains the capacity to self-host Racket, by utilizing the module system to locate and load the core Racket libraries, and reading and expanding all the modules to load a Racket language (e.g. *racket/base*), fully supporting Racket’s semantics. In Chapter 4 we present this process in detail by which we turn Pycket into a full Racket implementation.

In the next section, we delve into the semantics of the central construct enabling this enhanced portability—linklets. We present a formal operational semantics for linklets and describe an executable PLT Redex model that captures their compilation and execution semantics.

3.2 Linklets as Units of Compilation

In this section, we demonstrate through a practical example how linklets facilitate interesting runtime functionalities, laying the groundwork for improved compiler-runtime communication and, eventually, self-hosting.

A *linklet*, as depicted in Figure 3.3, is a lambda-like binding construct composed of three primary components: a set of imported variables, a set of exported variables, and a body containing definitions and expressions. Real-world examples of linklets, represented as s-expressions, can be seen in Figure 3.2. Imported variables allow linklets to utilize external definitions, while exported variables provide definitions to other linklets. Notably, the exported

$$\begin{aligned}
L &::= (\textbf{linklet } ((imp \dots) \dots) (exp \dots) l\text{-}top \dots e) \\
l\text{-}top &::= (\textbf{define-values } (x) e) \mid e \\
imp &::= x \mid (xx) \quad [\text{external-name internal-name}] \\
exp &::= x \mid (xx) \quad [\text{internal-name external-name}] \\
e &::= x \mid v \mid (e e \dots) \mid (\textbf{if } e e e) \mid (o e e) \\
&\mid (\textbf{begin } e e \dots) \mid (\textbf{lambda } (x_! \dots) e) \\
&\mid (\textbf{set! } x e) \mid (\textbf{raises } e) \\
&\mid (\textbf{var-ref } x) \mid (\textbf{var-ref/no-check } x) \\
&\mid (\textbf{var-set! } x e) \\
&\mid (\textbf{var-set/check-undef! } x e)
\end{aligned}$$

Figure 3.3: Linklet Source Language

set can include variables without corresponding definitions in the body, enabling dynamic binding resolution at runtime.

Linklets as standalone s-expressions are no more inherently meaningful than a conventional lambda expression. But when *compiled* by a hosting runtime into a linklet object, a linklet becomes ready for *instantiation*, a process analogous to a function application, wherein imported variables are provided, and the linklet body expressions are sequentially evaluated. Instantiation can occur in two modes that we will refer to as *regular instantiation* and *targeted instantiation* in the rest of the chapter, each resulting in different outcomes.

Regular instantiation is where a linklet evaluates its body after receiving imported variables from other linklet instances. As illustrated in Figure 3.4, regular instantiation produces a fresh *linklet instance*, which serves as a container holding exported variables and additional runtime data such as namespaces. The resulting linklet instance subsequently provides its variables to

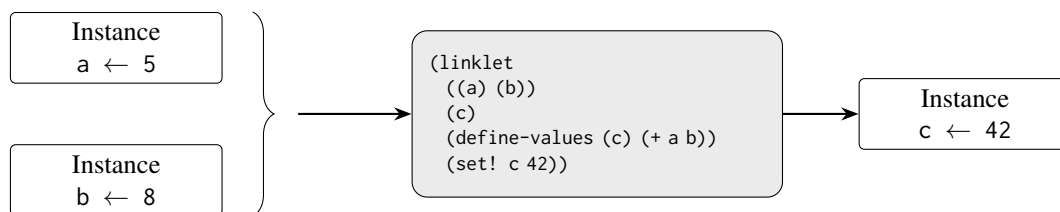


Figure 3.4: Regular Instantiation of a Linklet

other linklet instantiations.

A targeted instantiation, by contrast, is where a linklet is provided with a pre-existing *target* instance alongside linklet instances for the imported variables. In this case, the variables in the target instance is used and modified for the linklet definitions and expressions during the evaluation of the expressions within the linklet body. This kind of instantiation is often used for side-effects, and the result is the value of the last expression in the linklet rather than a new linklet instance.

The linklet design does not impose a specific internal representation, allowing a VM flexibility in selecting preferred representations for linklets and instances. For example, Chez Scheme represents linklets as Scheme functions, whereas Pycket employs a custom runtime class named `W_Linklet`. To manage these diverse representations uniformly, Racket specifies a minimal runtime API for VMs, comprising two functions: `compile-linklet` for preparing linklets and `instantiate-linklet` for executing them.

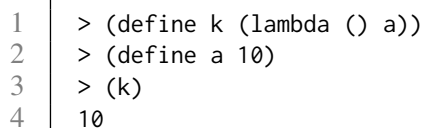
Loading linklets into a VM runtime involves compiling linklet s-expressions into runtime objects and subsequently instantiating these objects to obtain their exported variables. Specifically, `compile-linklet` transforms linklet s-expressions into runtime objects containing the runtime

AST for the enclosed code. Once instantiated, all exported variables become immediately available to other linklets via the resulting instance. These variables can encapsulate any Racket value, including callable closures, allowing the runtime to directly invoke powerful functions without explicitly reimplementing them, such as the `expand` function from the expander linklet.

3.2.1 Using Linklets: A Small Top-level REPL Interaction

To clarify and illustrate the semantics of linklets in a more concrete manner, we introduce a small yet informative example of how a VM implements a high-level Racket feature using linklets: the top-level Read-Eval-Print Loop (REPL). Although the top-level implementation itself is quite complex and beyond our scope, this example demonstrates clearly how the runtime utilizes high-level features provided via linklets—particularly those exported by the expander, including `read`, `expand`, and `eval`—to create a REPL behavior.

As a running example, consider the simple REPL interaction in Figure 3.5, in which a closure `k` references an initially undefined variable `a`. Subsequently, the variable `a` is defined, and the closure is invoked, yielding the result `10`. This scenario is intended to illustrate how linklets implement dynamic variable bindings.



```
1 > (define k (lambda () a))
2 > (define a 10)
3 > (k)
4 10
```

Figure 3.5: Example Top-level REPL Interaction to Implement Using Linklets

This interaction, despite seemingly contradicting lexical scoping, is perfectly valid within REPL environments, which are historically designed to evaluate expressions incrementally within mutable top-level namespaces. Such environments allow the definition of constructs like mutually recursive functions in a step-by-step manner. To simulate this dynamic behavior, our VM maintains a dedicated *top-level* linklet instance, initially empty, that serves as a mutable namespace for dynamically defining and modifying variables throughout the REPL session.

At each interaction step in the REPL, the VM receives a Racket s-expression to evaluate. It first applies Racket’s `read` and `expand` functions—provided by the expander linklet—to convert the s-expression into a linklet. This newly generated linklet is then compiled using `compile-linklet` into a linklet object, which undergoes targeted instantiation with the top-level instance as its target. Figure 3.6 shows the generated and compiled linklets along with the updated state of the environment and top-level instance after each step¹.

Each step in the REPL involves dynamically using Racket’s own `read` and `expand` functions provided by the expander linklet, rather than runtime-specific implementations. This demonstrates a core advantage of linklets, allowing the runtime to leverage the high-level functionalities directly exported from Racket itself, significantly simplifying the runtime’s implementation burden.

An interesting point about the top-level in this example is the handling of the undefined identifier `a`. When the runtime compiles a linklet containing such an identifier, it designates that identifier as an exported variable of the linklet (despite having no corresponding definition

¹For clarity, we will omit the non-essential details and focus only on the simplified run-time linklet for the expression

(a) Linklet generated by expand	(b) Linklet compiled by VM	(c) Environment after compile	(d) Target after instantiate
<hr/>			
> (define k (lambda () a))			
(linklet (...) (a k) (define-values (k) (lambda () a)))	(linklet (...) ((a0 a) (k0 k)) (define-values (k) (lambda () (variable-ref (LinkletVar a0)))) (variable-set! (LinkletVar k0) k))	$a_0 \rightarrow \text{var}_a$ $k_0 \rightarrow \text{var}_k$	$a \rightarrow \text{var}_a$ $k \rightarrow \text{var}_k$
<hr/>			
> (define a 10)			
(linklet (...) (a) (define-values (a) 10))	(linklet (...) ((a1 a)) (define-values (a) 10) (variable-set! (LinkletVar a1) a))	$a_1 \rightarrow \text{var}_a$	$a \rightarrow \text{var}_a$ $k \rightarrow \text{var}_k$
<hr/>			
> (k)			
(linklet (...) (k) (k))	(linklet (...) ((k1 k)) ((variable-ref (LinkletVar k1))))	$k_1 \rightarrow \text{var}_k$	$a \rightarrow \text{var}_a$ $k \rightarrow \text{var}_k$
<hr/>			

Figure 3.6: Internal steps in linklet instantiations for step-by-step Top-level REPL interactions.

for it). During the compilation, a new linklet variable is created for it with the special value *uninitialized* (e.g. var_a in our example). When the runtime for later refers to such an identifier, it will be dynamically provided using the *target* instance (in this case using the top-level instance), essentially forming a low-level dynamic scope. Note that in the first row of the Figure 3.6, we have the variable mappings for both a and the k within the target instance after the instantiation. They contain the values *uninitialized* and a closure with no arguments and body with a reference to var_a respectively.

Since each instantiation within the REPL is a targeted instantiation, changes made to the top-level instance persist across interactions. For instance, after the second interaction step depicted in Figure 3.6, a 's value (10) is directly assigned to var_a via a *variable-set!* operation in the compiled linklet. If var_a had not existed in the top-level instance initially, a new variable initialized as *uninitialized* would have been created (as occurred in the first interaction). In this case, however, since var_a existed, it was simply updated to hold the new value.

At the third interaction step, when executing the closure k , the identifier a inside the closure is dynamically resolved to the var_a via the *variable-ref* call inside the compiled linklet. Then the runtime consults the *target* top-level instance to obtain a 's current value (10), already set thanks to the previous step. This demonstrates linklets' flexibility in managing runtime variable bindings across dynamic execution contexts.

This dynamic interplay of linklets is made possible by the clever use of linklet variables and target instances, specified by the semantics that are embedded in the *compile-linklet*. In the subsequent section, we present these formal operational semantics in detail, underpinning the

correctness and robustness of linklet interactions and their use in runtime implementations.

3.3 Formal Operational Semantics of Linklets

In this section, we present a formal specification of the operational semantics of linklets, which serves as the foundation for their implementation in Pycket, discussed in the following chapter (Chapter 4). We develop this formalism using the PLT Redex language, making it executable and enabling rigorous validation through randomized testing [30, 31]. The validation methodology, including details of randomized testing and model correctness, is further elaborated in Chapter 5.

The grammar of the linklet language, presented earlier in Figure 3.3, defines a minimal subset of fully-expanded Racket programs extended with the special linklet form and top-level definitions (e.g., `define-values`). Besides common forms like `begin`, `set!`, and `if`, the grammar includes additional constructs specifically for manipulating variables (e.g., `var-ref`), as shown in the rule for expressions (e). These special constructs handle what we refer to as *linklet variables*, which semantically behave just like regular variables in the runtime but are explicitly manipulated within linklets. We will detail these variable manipulation forms further when discussing the semantics of `compile-linklet`.

The sub-language used within a linklet body, which we refer to as Linklet Kernel Language (LKL), is essentially a λ -calculus enriched with a minimal set of syntactic extensions. We present the detailed grammar, a standard reduction relation, and an evaluator for LKL in Appendix A, as these form the core language referred to by the linklet semantics. Importantly, LKL aligns closely with Racket’s `#%kernel` language, thus any rudimentary interpreter for

#%kernel forms can easily be extended to support linklets by solely adding the linklet-specific semantics.

$$\begin{aligned}
CL &::= \Phi^C(L) \\
L\text{-obj} &::= (\mathbf{L}_\alpha \text{ } c\text{-imps } c\text{-exps } l\text{-top} \dots) \mid (\mathbf{L}_\beta \text{ } x \text{ } l\text{-top} \dots) \\
LI &::= (\mathbf{linklet-instance} \text{ } (x \text{ } var) \dots) \\
c\text{-imps} &::= ((imp\text{-obj} \dots) \dots) \\
c\text{-exps} &::= (exp\text{-obj} \dots) \\
imp\text{-obj} &::= (\mathbf{Import} \text{ } x \text{ } x \text{ } x) \quad [\text{id internal external}] \\
exp\text{-obj} &::= (\mathbf{Export} \text{ } x \text{ } x \text{ } x) \quad [\text{id internal external}]
\end{aligned}$$

$\Phi^C : \mathbf{compile-linklet}$

Figure 3.7: Linklet Runtime Language

Figure 3.7 illustrates the runtime representations for compiled linklets and linklet instances. Given a linklet L , `compile-linklet` produces a compiled linklet object \mathbf{L}_α ready for instantiation. Such an object includes *Import* and *Export* entries that encode the imported and exported variables, respectively. For clarity, we omit the details concerning compilation of basic forms (e.g., `begin`, `set!`) into runtime Abstract Syntax Trees (ASTs), focusing instead only on the constructs specific to linklets. Furthermore, the body of a compiled linklet will include additional runtime expressions inserted during compilation to manage dynamic variable manipulations, as explained in detail in subsequent paragraphs.

Before `compile-linklet` is invoked, several preprocessing passes are performed on the linklet body. These passes produce *Import* and *Export* objects, each containing internal and external identifiers to accommodate potential renaming, as well as freshly (gensym) generated identifiers

for runtime use. Additionally, these preprocessing steps gather information about identifiers defined within the linklet body, and those targeted by mutation operations, all to be used during compilation. Given the sets of *Imports* (C_I), *Exports* (C_E), top-level defined identifiers (X_T), and mutated identifiers (X_M), `compile-linklet` processes each expression within the linklet body one-by-one. It utilizes the data collected in preprocessing to correctly compile variable references and mutation operations into the appropriate runtime forms.

Expression Pattern	Condition	Transformation / Action
(define-values (x) e)	$(\text{Export } x_{gen} \ x \ x_{ext}) \in C_E$	Emit (var-set! x_gen x)
(set! x e)	$(\text{Export } x_{gen} \ x \ x_{ext}) \in C_E$	Transform set! into (var-set/check-undef! x_gen e)
x (identifier)	$(\text{Import } x_{gen} \ x \ x_{ext}) \in C_I$	Transform x into (var-ref/no-check x_gen)
x (identifier)	$(\text{Export } x_{gen} \ x \ x_{ext}) \in C_E$ and $x \in X_M$ or $x \notin X_T$	Transform x reference into (var-ref x_gen)
Any other e	—	Recurse into sub-forms (if any), otherwise return e

Table 3.1: Transformation rules for linklet variables used in `compile-linklet`.

Table 3.1 lists the transformation rules applied by `compile-linklet` to handle linklet variables. Exported and top-level defined identifiers are compiled into linklet variables expected to be created during instantiation. An exported identifier that is targeted by a `set!` is compiled into a `var-set/check-undef!` operation to persist mutation across linklets. References to imported variables are also compiled as runtime linklet variable references. Additionally, an exported identifier without a definition, or one that is targeted by mutation, is similarly compiled into a runtime linklet variable reference, as it will be provided by the target instance during instantiation.

Instantiation of a compiled linklet begins by processing its *Import* and *Export* objects. Imported variables are collected from the provided linklet instances, and exported variables are either fetched from the target instance or freshly created, as previously described in the top-level example (Section 3.2.1). For simplicity, we denote by L_β a compiled linklet object after this initial step, meaning that L_β no longer explicitly includes import and export objects as these have been resolved. With such a prepared L_β , instantiation then proceeds to evaluation of its body expressions.

Recall that instantiation occurs in two distinct modes: *regular instantiation*, resulting in a new linklet instance, and *targeted instantiation*, which uses an existing target instance and returns the value of the last evaluated expression. As part of preparing a compiled linklet for instantiation, compile-linklet transitions from L_α to L_β by plugging in a target instance. In regular instantiation, this target instance is freshly created as an empty instance, whereas in targeted instantiation, it is the pre-existing instance provided as input to the `instantiate-linklet`. For simplicity, we unify these two modes in the semantics by always considering an explicit target instance, and denote the instantiation result uniformly as a pair consisting of the last expression's value and the target instance used during evaluation.

Since linklets alone are binding constructs and cannot initiate computation, we introduce a top-level construct named `program` to demonstrate their operational semantics in context. A program comprises a set of linklets to load and a single top-level expression to initiate computation. The grammar for `program` is shown in Figure 3.8, where the `let-inst` form names linklet instances, and the `seq` form sequences computations. Additionally, the function Φ^V (*instance-variable-value*) retrieves the value of an exported variable from a given instance.

$$\begin{aligned}
p &::= (\mathbf{program} ((x \ L) \ \dots) \ p\text{-top}) \\
p\text{-top} &::= v \\
&| (\mathbf{let-inst} \ x \ p\text{-top} \ p\text{-top}) \\
&| (\mathbf{seq} \ p\text{-top} \ \dots) \\
&| \Phi^I(l\text{-ref} \ x \ \dots) \\
&| \Phi^I(l\text{-ref} \ x \ \dots \ \#\mathbf{t} \ x) \\
&| \Phi^V(x \ x) \\
l\text{-ref} &::= x \mid L\text{-obj} \\
v &::= \dots \mid (v \ x)
\end{aligned}$$

Φ^V : **instance-variable-value** , Φ^I : **instantiate-linklet**

Figure 3.8: Linklet Program Source Language

Figure 3.9 presents the reduction relation, β_p , which defines the evaluation of linklet programs along with the reduction relation β_l for the LKL. For clarity and simplicity, the evaluation of programs and linklet instantiations are both expressed by the single relation β_p , thus forming a coherent, small-step evaluation sequence from programs to values. To maintain readability, trivial reduction rules for forms such as `seq` and `let-inst` are omitted.

3.3.1 Top-level REPL Example in the Formal Model

To concretely illustrate how our formal semantics capture the top-level REPL interaction described earlier (Section 3.2.1), we represent it as a program form and simulate its evaluation within our semantics. Figure 3.10 shows this top-level interaction represented as a program. We prepare individual linklets for each step of the interaction, then instantiate (execute) them sequentially using a shared target instance introduced by the `let-inst` form.

$$\begin{aligned}
& EP \llbracket E \llbracket x \rrbracket \rrbracket, \rho, \sigma \longrightarrow_{\beta_l} EP \llbracket E \llbracket \sigma[\rho[x]] \rrbracket \rrbracket, \rho, \sigma \\
& EP \llbracket E \llbracket (\mathbf{var-ref} \ x) \rrbracket \rrbracket, \rho, \sigma \longrightarrow_{\beta_l} EP \llbracket E \llbracket v \rrbracket \rrbracket, \rho, \sigma \quad \mathbf{where} \ x \in \rho, \rho[x] \in \sigma, v = \sigma[\rho[x]], v \neq \mathbf{uninit} \\
& EP \llbracket E \llbracket (\mathbf{var-ref/no-check} \ x) \rrbracket \rrbracket, \rho, \sigma \longrightarrow_{\beta_l} EP \llbracket E \llbracket v \rrbracket \rrbracket, \rho, \sigma \quad \mathbf{where} \ v = \sigma[\rho[x]] \\
& EP \llbracket E \llbracket (\mathbf{var-set!} \ x \ v) \rrbracket \rrbracket, \rho, \sigma \longrightarrow_{\beta_l} EP \llbracket E \llbracket (\mathbf{void}) \rrbracket \rrbracket, \rho, \sigma_1 \quad \mathbf{where} \ \sigma_1 = \sigma[\rho[x] \rightarrow v] \\
& EP \llbracket E \llbracket (\mathbf{var-set/check-undef!} \ x \ v) \rrbracket \rrbracket, \rho, \sigma \longrightarrow_{\beta_l} EP \llbracket E \llbracket (\mathbf{void}) \rrbracket \rrbracket, \rho, \sigma_1 \quad \mathbf{where} \ \sigma_1 = \sigma[\rho[x] \rightarrow v], x \in \rho \\
& EP \llbracket E \llbracket (\mathbf{set!} \ x \ v) \rrbracket \rrbracket, \rho, \sigma \longrightarrow_{\beta_l} EP \llbracket E \llbracket (\mathbf{void}) \rrbracket \rrbracket, \rho, \sigma_1 \quad \mathbf{where} \ \sigma_1 = \sigma[\rho[x] \rightarrow v] \\
\\
& EP \llbracket \Phi^V(x_{li}, x) \rrbracket, \rho, \sigma \longrightarrow_{\beta_p} EP \llbracket v \rrbracket, \rho, \sigma \quad \mathbf{where} \ v = \sigma[(\sigma[x_{li}])[x]] \\
& EP \llbracket (\Phi^I(\mathbf{L}_\beta \ x \ v \ \dots \ v_l)) \rrbracket, \rho, \sigma \longrightarrow_{\beta_p} EP \llbracket (v_l \ x) \rrbracket, \rho, \sigma \\
& EP \llbracket (\Phi^I(\mathbf{L}_\beta \ x_t \ v_p \ \dots \ (\mathbf{define-values} \ (x) \ v) \ l\text{-top} \ \dots)) \rrbracket, \longrightarrow_{\beta_p} EP \llbracket \Phi^I(\mathbf{L}_\beta \ x_t \ v_p \ \dots \ l\text{-top} \ \dots) \rrbracket, \rho_1, \sigma_1 \\
& \quad \mathbf{where} \ \rho_1 = \rho[x \rightarrow \mathbf{cell}], \sigma_1 = \sigma[\mathbf{cell} \rightarrow v] \\
& EP \llbracket (\Phi^I(\mathbf{L}_\beta \ x_t \ v_p \ \dots \ l\text{-top}_1 \ l\text{-top} \ \dots)) \rrbracket, \longrightarrow_{\beta_p} EP \llbracket \Phi^I(\mathbf{L}_\beta \ x_t \ v_p \ \dots \ v_1 \ l\text{-top} \ \dots) \rrbracket, \rho_1, \sigma_1 \\
& \quad \mathbf{where} \ l\text{-top}_1 \longrightarrow_{\beta_l}^* v_1 \\
& EP \llbracket (\Phi^I(\mathbf{L}_\alpha \ c\text{-imps} \ c\text{-exps} \ l\text{-top} \ \dots) \ LI \ \dots) \rrbracket, \rho, \sigma \longrightarrow_{\beta_p} EP \llbracket (\Phi^I(\mathbf{L}_\alpha \ c\text{-imps} \ c\text{-exps} \ l\text{-top} \ \dots) \ LI \ \dots \ \mathbf{\#t} \ x_t) \rrbracket, \rho, \sigma_1 \\
& \quad \mathbf{where} \ x_t \notin \text{dom}(\sigma), \sigma_1 = \sigma[x_t \rightarrow (LI)] \\
& EP \llbracket (\Phi^I(\mathbf{L}_\alpha \ c\text{-imps} \ c\text{-exps} \ l\text{-top} \ \dots) \ LI \ \dots \ \mathbf{\#t} \ x_t) \rrbracket, \rho, \sigma \longrightarrow_{\beta_p} EP \llbracket (\Phi^I(\mathbf{L}_\beta \ x_t \ l\text{-top} \ \dots)) \rrbracket, \rho_2, \sigma_1 \\
& \quad \mathbf{where} \ \rho_1 = V^I(c\text{-imps}, (LI \ \dots), \rho) \\
& \quad \quad (\rho_2, \sigma_1) = V^E(c\text{-exps}, x_t, \rho_1, \sigma) \\
& (\mathbf{program} \ ((x \ L), (x_1 \ L_1) \ \dots) \ p\text{-top}), \rho, \sigma \longrightarrow_{\beta_p} (\mathbf{program} \ ((x_1 \ L_1) \ \dots) \ p\text{-top}[x := \Phi^C(L)]), \rho, \sigma \\
\\
& eval \xrightarrow{\beta_p} (p) = v \ \mathbf{if} \ p, (), () \twoheadrightarrow_{\beta_p \cup \beta_l} (\mathbf{program} \ () \ (v \ _)) \\
\\
& V^I : c\text{-imps} \times (LI \ \dots) \times \rho \longrightarrow \rho \\
& V^I(\overline{((\mathbf{Import} \ x_{id} \ x_{int} \ x_{ext}))_n}), (LI_n), \rho = \rho[x_{id} \rightarrow LI_n[x_{ext}]] \\
\\
& V^E : c\text{-exps} \times x \times \rho \times \sigma \longrightarrow \rho \times \sigma \\
& V^E((exp\text{-obj} \ \dots), x_t, \rho, \sigma) = P(exp\text{-obj}, x_t, \sigma[x_t], \rho, \sigma) \ \dots \\
& P((\mathbf{Export} \ x_{id} \ x_{int} \ x_{ext}), x_t, LI_t, \rho, \sigma) = \begin{cases} \rho_1, \sigma & \text{if } x_{ext} \in \text{dom}(LI_t), \mathbf{where} \ \rho_1 = \rho[x_{id} \rightarrow LI_t[x_{ext}]] \\ \rho_1, \sigma_1 & \text{if } x_{ext} \notin \text{dom}(LI_t), \mathbf{where} \ \rho_1 = \rho[x_{id} \rightarrow var_{new}] \\ & var_{new} \notin \sigma \\ & \sigma_1 = \sigma[var_{new} \rightarrow \mathbf{uninit}, \\ & \quad x_t \rightarrow LI_t[x_{ext} \rightarrow var_{new}]] \end{cases}
\end{aligned}$$

Φ^V : instance-variable-value, Φ^I : instantiate-linklet, Φ^C : compile-linklet
 V^I : get import variables, V^E : create variables for exports

Figure 3.9: Standard Reduction Relations for Operational Semantics of Linklets

```

1 (program ([l1 (linklet () (a k) (define-values (k) (lambda () a)) (void))])
2           [l2 (linklet () (a) (define-values (a) 10) (void))])
3           [l3 (linklet () (k) (k))])
4 (let-inst t (make-instance)
5   (seq ( $\phi^I$  l1 #:t t) ( $\phi^I$  l2 #:t t) ( $\phi^I$  l3 #:t t)))

```

Φ^I : **instantiate-linklet**

Figure 3.10: Top-level REPL Interaction from Figure 3.5 in Linklets Formalism

The initial steps of the formal (multi-step) reduction, namely $\longrightarrow_{\beta_p}$, for the program are shown in Table 3.2, following the reduction relation defined in Figure 3.9. The reduction begins by compiling each linklet provided to the program. Once compilation is complete, the resulting L_α forms are plugged in the program’s body, and evaluation starts. For clarity, the remaining reduction steps, which ultimately yield the final value (10), are presented in Appendix B.

At the beginning of linklet instantiation, imported variables are gathered from the provided linklet instances. Each imported variable reference is then mapped in the environment according to the identifiers in the corresponding *Import* objects created during compilation. Recall that a linklet may export variables without corresponding definitions. For these, each exported variable is fetched from the target instance if present; otherwise, a new variable is created with an initial value of *uninitialized*. The environment and the target instance mappings are updated accordingly.

After setting up import and export variables, the linklet object L_α reduces to L_β , triggering evaluation of the linklet body expressions. The earlier compilation ensures each variable reference within the linklet body resolves correctly. Evaluation then proceeds through each ex-

	program	ρ	σ
	<pre>(program ([11 (linklet () (a k) (define-values (k) (lambda () a)) (void))) [12 (linklet () (a) (define-values (a) 10) (void))] [13 (linklet () (k) (k))]) (let-inst t (make-instance) (seq (ϕ^I 11 #:t t) (ϕ^I 12 #:t t) (ϕ^I 13 #:t t))))</pre>	[]	[]
$\longrightarrow_{\beta p}^*$	<pre>(program () (let-inst t (make-instance) (seq (ϕ^I (Lα () ((Export a1 a a) (Export k1 k k)) (define-values (k) (lambda () (var-ref a1))) (var-set! k1 k) (void)) #:t t) (ϕ^I (Lα () ((Export a1 a a) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))))</pre>	[]	[]
$\longrightarrow_{\beta p}^*$	<pre>(program () (seq (ϕ^I (Lβ t (define-values (k) (lambda () (var-ref a1))) (var-set! k1 k) (void))) (ϕ^I (Lα () ((Export a1 a a) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))))</pre>	$[k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[var_a, var_k \rightarrow \text{uninit},$ $t \rightarrow (LI$ $(a var_a) (k var_k))]$
$\longrightarrow_{\beta p}^*$	<pre>(program () (seq (ϕ^I (Lβ t (var-set! k1 k) (void))) (ϕ^I (Lα () ((Export a1 a a) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))))</pre>	$[k \rightarrow cell_1,$ $k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a, var_k \rightarrow \text{uninit},$ $t \rightarrow (LI$ $(a var_a) (k var_k))]$

Table 3.2: Top-level REPL Example Reduction Steps using $\longrightarrow_{\beta p}$

pression sequentially. Once all expressions are evaluated, the instantiation concludes, returning a pair consisting of the last expression’s value and the target instance that was used.

The formalism described in this section is fully implemented as an executable PLT Redex model. The complete Redex model code, including definitions and tests, is available in the Appendix C. Additionally, the Redex model implementation is accessible online at the project’s repository². The complete sequence of reductions for the top-level REPL example discussed previously is provided in Appendix B.

3.4 Conclusion

The ability to provide high-level functionalities as callable functions to a hosting VM is central to improving language portability. By exporting implementations of essential features—such as the module system and macro system—as self-contained linklets, Racket substantially reduces the effort required for runtimes to support its semantics. This approach enables runtimes to seamlessly incorporate and reuse complex, high-level functionalities directly, as demonstrated by the top-level REPL example in Section 3.2.1.

Moreover, the interaction between the Racket language and its hosting runtime via linklets is inherently bidirectional. On one hand, the runtime implements `compile-linklet` and `instantiate-linklet`, allowing it to load and invoke high-level functionalities provided by the language. On the other hand, these high-level language components—such as the `expander` linklet—rely directly on runtime-implemented primitives and the linklet APIs themselves. For instance, Racket’s `dynamic-require` function dynamically resolves module paths, interacts with

²<https://github.com/cderici/linklets-redex-model>

the file system, and utilizes runtime-provided functions to compile and instantiate necessary modules. Similarly, the `eval` function leverages the same set of runtime primitives to execute Racket code. This mutual dependence between high-level language features and low-level runtime support via linklets is fundamental to Racket’s enhanced portability.

In this chapter, we examined how enhancing compiler and runtime communication at the language level significantly improves portability. Using Racket’s linklets as a concrete example, we introduced a formal operational semantics, executable in PLT Redex, to precisely specify how linklets encapsulate high-level functionalities for convenient runtime adoption. Building on these semantic foundations, the next chapter (Chapter 4) concretely realizes self-hosting of Racket on Pycket, demonstrating the practical viability of language-powered runtimes and effectively providing proof for our thesis statement.

4. FROM RUDIMENTARY INTERPRETER TO FULL LANGUAGE IMPLEMENTATION

CHAPTER SYNOPSIS

Languages can influence the capabilities of the run-time they are implemented on.

Concrete realization of self-hosting of Racket on Pycket.

Sections:

- Implementing Linklets on Pycket

This is how Pycket is enhanced to interface with linklets. Representation choices, etc.

- Enhancing Run-time with Bootstrapping Linklets

Importing functionality straight from the language in the form of linklets allows the language to shape the behavior of the run-time.

- From Rudimentary Interpreter to Full Language Implementation

Pycket becomes a full working Racket, proving the self-hosting hypothesis.

Languages can influence and enhance the capabilities of the run-time environments or VMs they are implemented on. This influence is evident in mature language implementations such as Smalltalk’s Squeak, Erlang’s OTP on the BEAM VM, and Common Lisp’s CLOS Meta-Object Protocol (MOP). For example, Smalltalk allows core image manipulation within Smalltalk itself, extending VM semantics dynamically [32]. Erlang uses its OTP libraries written in Erlang to dynamically control crucial runtime semantics like process scheduling and fault tolerance [33]. Similarly, Common Lisp employs the CLOS MOP to redefine fundamental object system semantics at runtime without altering the native runtime [34].

While the extent of language influence on the runtime varies considerably, Racket’s linklets stand out by enabling the language to self-host itself. Linklets can carry crucial Racket subsystems, including the macro expander, the module system, and essential runtime components, to be expressed and distributed as language-level constructs that are supported by a thin primitive layer by the VM. This mechanism significantly reduces dependency on the underlying runtime, thereby simplifying portability across different VMs.

Recall Pycket from Section 2.2, initially introduced as a rudimentary interpreter for Racket, lacking capability required to independently execute real-world Racket programs. Pycket’s initial design relied on the existing Racket executable to read and expand Racket modules before evaluation. In contrast, a *full language implementation* refers to an environment capable of independently performing all necessary tasks required to load and execute programs written in that language. Such implementation can read source code, expand macros (load languages), evaluate modules, and provide the full spectrum of runtime facilities, such as a REPL, error handling, I/O operations, and more, without external assistance.

Building upon the concepts introduced in Chapter 3, this chapter presents concrete implementation details that transition Pycket from a rudimentary interpreter into a fully operational Racket implementation. By incorporating linklets, Pycket gains the ability to independently load, expand, and evaluate Racket modules, thus substantiating the self-hosting hypothesis outlined earlier. The following sections elaborate on the representation and implementation choices involved in this transition, highlighting runtime enhancements.

4.1 Implementing Linklets on Pycket

The first step in establishing an interface with linklets is to define their internal representation. Recall that the linklet design itself does not mandate a particular representation, leaving the decision to the runtime implementer. Chez Scheme, for example, represents linklets as first-class functions. Pycket, in contrast, represents both linklets and linklet instances as first-class Racket values using the custom `W_Linklet` and `W_LinkletInstance` classes, both derived from `W_Object` (the parent abstract class representing Racket values). Figure 4.1 lists the class definitions of linklets and linklet instances in Pycket. This choice facilitates seamless integration and leverages Pycket’s existing object model.

For linklet variables, Pycket employs a specialized AST node named `LinkletVar`, which evaluates to heap-allocated cells (`W_Cell` objects) rather than separate dedicated runtime values. This approach allows the trace optimizer to perform effective optimizations on the top-level environment. Using cells for linklet variables also makes it possible to leverage specialized strategies implemented for cells, such as type specialization and inline caching, thus enhancing runtime efficiency. Consequently, these representation choices significantly influence the

```

class W_Linklet(W_Object):
    _attrs_ = _immutable_fields_ = ["name", "importss", "exports", "forms"]

    def __init__(self, name, importss, exports, all_forms):
        self.name = name
        self.importss = importss    # [[Import ...] ...]
        self.exports = exports      # {int_id:Export ...}
        self.forms = all_forms      # [..., AST ,...]

class W_LinkletInstance(W_Object):
    _attrs_ = ["name", "vars", "data"]
    _immutable_fields_ = ["name", "data"]

    def __init__(self, name, vars, data=w_false):
        self.name = name
        self.vars = vars # {W_Symbol:W_Cell}
        self.data = data

```

Figure 4.1: Representation of Linklets and Linklet Instances on Pycket

design and implementation of *compile-linklet* and *instantiate-linklet*, as these functions directly manifest the runtime’s internal representation for linklets.

The *compile-linklet* function transforms a linklet expressed as an s-expression into a runnable linklet object, following the formal semantics described in Section 3.3. It begins by analyzing the s-expression through multiple passes to identify identifiers defined or mutated within the linklet body. Next, it processes the identifiers for imports and exports, creating runtime objects (*Import* and *Export*) and generating (via *gensym*) fresh identifiers for internal use as needed. Finally, it recursively converts the linklet body into an RPython AST, encapsulating all this information within a *W_Linklet* object ready for instantiation.

The first phase of linklet compilation involves handling imports and exports. Each import specification, kept in a linklet as `[Import ...]` as seen in Figure 4.1, references a specific imported instance, with each *Import* object corresponding to a variable provided by that

instance. If the linklet specifies distinct names for external (provided by the imported instance) and internal (used within the linklet body) references, these mappings are recorded explicitly within an `Import` object. Likewise, exports are treated similarly; each exported variable may have separate external and internal identifiers, the former visible to importing linklets and the latter used internally within the body. This explicit mapping simplifies variable management and prevents naming conflicts across linklet boundaries.

After imports and exports are handled, the compilation proceeds with processing the linklet body. Initially, it identifies all targets of `set!` expressions and variables introduced by `define-values` forms. Subsequently, each form within the body is individually compiled from s-expression to Pycket AST. Most compilation rules follow straightforward translations, with the notable exceptions involving linklet-variable-specific forms—such as `variable-ref`, `variable-set!`, and related variations—rules of which are detailed in Table 3.1 in Chapter 3. These forms explicitly handle interactions with linklet variables, ensuring correct runtime semantics during instantiation.

For each top-level defined identifier that’s also exported, an additional `variable-set!` form is inserted by the compiler to ensure the correct runtime assignment to the corresponding linklet variable. When an exported identifier appears as a target of a `set!`, the compiler translates the original form into a `variable-set!/check-undefined`, thereby ensuring proper handling of potentially uninitialized variables at runtime. References to exported identifiers that are either undefined at the top level or targeted by a `set!` are explicitly compiled into `variable-ref` forms, enabling dynamic resolution via target instance during instantiation. Finally, references to imported variables are transformed into `variable-ref/no-check` forms. These forms omit

runtime existence checks since instantiation will already fail if the variable cannot be imported successfully. Once the entire linklet s-expression has been processed, the *compile-linklet* function outputs a `W_Linklet` object containing all essential runtime information. Specifically, this object includes the prepared *Import* and *Export* mappings, as well as the compiled Pycket AST representing the linklet body.

On the other hand, *instantiate-linklet* performs the actual evaluation of a compiled linklet, implementing the semantics described by the reduction relation in Figure 3.9. Initially, it processes the *Import* and *Export* objects, collecting and creating linklet variables accordingly. If the number of instances provided for imports is insufficient, or if any import variable isn't exported by the corresponding input instance, instantiation halts immediately with an error. Next, the function ensures a target instance, either creating a new one during regular instantiation or using an explicitly provided instance during targeted instantiation. Finally, the instantiation proceeds by evaluating the body expressions using the CEK interpreter loop, loading specialized continuation frames (such as `instantiate_def_cont` and `instantiate_val_cont`) that effectively implement the transitive closure of the reduction relation $\longrightarrow_{\beta_p}$. Depending on the instantiation mode, it returns (by applying its own continuation to) either a value or a linklet instance (`W_LinkletInstance`).

Having implemented both *compile-linklet* and *instantiate-linklet*, Pycket gains the capability to independently execute any code expressed in Racket's core language (`#%kernel`) provided in the form of linklet s-expressions. Together with the internal representations and runtime mechanisms for linklets detailed earlier, these functions establish the foundation that allows Pycket to import and execute Racket's own runtime extensions, referred to as the bootstrap-

ping linklets. These bootstrapping linklets carry implementations of substantial high-level Racket subsystems—including the macro expander, module system, and other critical runtime components—thereby enabling self-hosting on any runtime environment capable of loading and instantiating linklets. The next section explores the integration and utilization of these bootstrapping linklets within Pycket in detail.

4.2 Enhancing Run-time with Bootstrapping Linklets

Bootstrapping linklets, as introduced in Chapter 3, provide a powerful mechanism for encapsulating major Racket subsystems, such as the macro expander, module system, and runtime components, into independently loadable units. By applying the Racket macro expander to any given module (including itself), one can generate a collection of linklets that may then be flattened and merged into a single, standalone linklet s-expression. Using this technique, Racket exposes essential functionalities as separate bootstrapping linklets, notably the expander, IO, threads, and additional utility linklets like `fasl` and `regexp`, each of which will be discussed in subsequent subsections.

With the capabilities provided by `compile-linklet` and `instantiate-linklet`, Pycket significantly transforms its startup process. Previously, Pycket depended on external tools for reading and expanding modules, as discussed in Section 3.1. Now, Pycket is able to independently load the bootstrapping linklets directly into its runtime during initialization, eliminating this dependency and enabling a fully self-contained bootstrapping phase.

Figure 4.2 illustrates Pycket’s process of loading a linklet at startup. Pycket begins by reading the linklet s-expression, then compiles it into a `W_Linklet` object using `compile-linklet`.

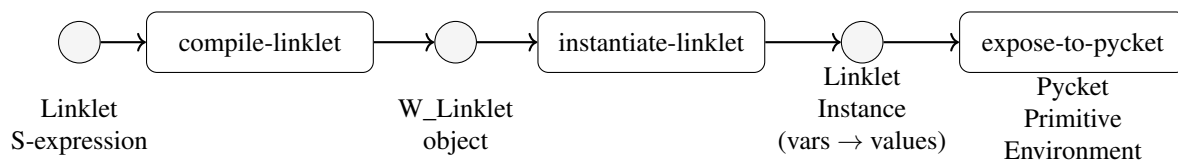


Figure 4.2: Overview of Pycket exposing values from a linklet at boot.

Subsequently, it instantiates this compiled linklet via `instantiate-linklet` to produce a linklet instance, which serves as a container holding all variables defined and exported by the linklet. These variables typically represent callable closures and other valid Racket values. Finally, Pycket exposes these variables in its global primitive environment, thereby making Racket-defined values—and more significantly, functions—directly callable from Pycket in a manner indistinguishable from its own primitives implemented in RPython.

Applying this loading mechanism, Pycket sequentially imports and integrates the bootstrapping linklets in the following order: *regex*, *thread*, *io*, *fasl*, and *expander*. Each linklet enhances Pycket’s runtime environment with specific Racket functionalities, exposing powerful primitives such as `read`, `write`, `expand`, and `namespace-require`. Table 4.1 summarizes the size of each linklet and the number of primitives each exposes. Notably, the *expander* linklet exports fewer primitives than the *IO* linklet despite having nearly three times as many lines of code.

Linklet	Lines of Code	# Exported Primitives
Expander	96,243	106
IO	36,292	222
Regex	8,280	23
Thread	12,947	105
Fasl	3,169	2

Table 4.1: Sizes and Number of Exported Primitives of Bootstrapping Linklets

In addition to standard bootstrapping linklets, Pycket loads a specialized linklet named `pycket_boot`, designed specifically for Pycket’s internal use. Previously, Pycket manually invoked Racket’s `dynamic-require` primitive from the global primitive environment to initialize the runtime, including loading core languages such as `racket/base`. With the availability of the linklet infrastructure, all such startup procedures—previously implemented directly in RPython—can now be written directly in Racket, encapsulated within the `pycket_boot` linklet. This approach significantly simplifies Pycket’s frontend initialization code, integrating setup procedures and configuration parameters like `current-library-collection-links`, `read-accept-compiled`, and `use-compiled-file-paths` entirely at the Racket level.

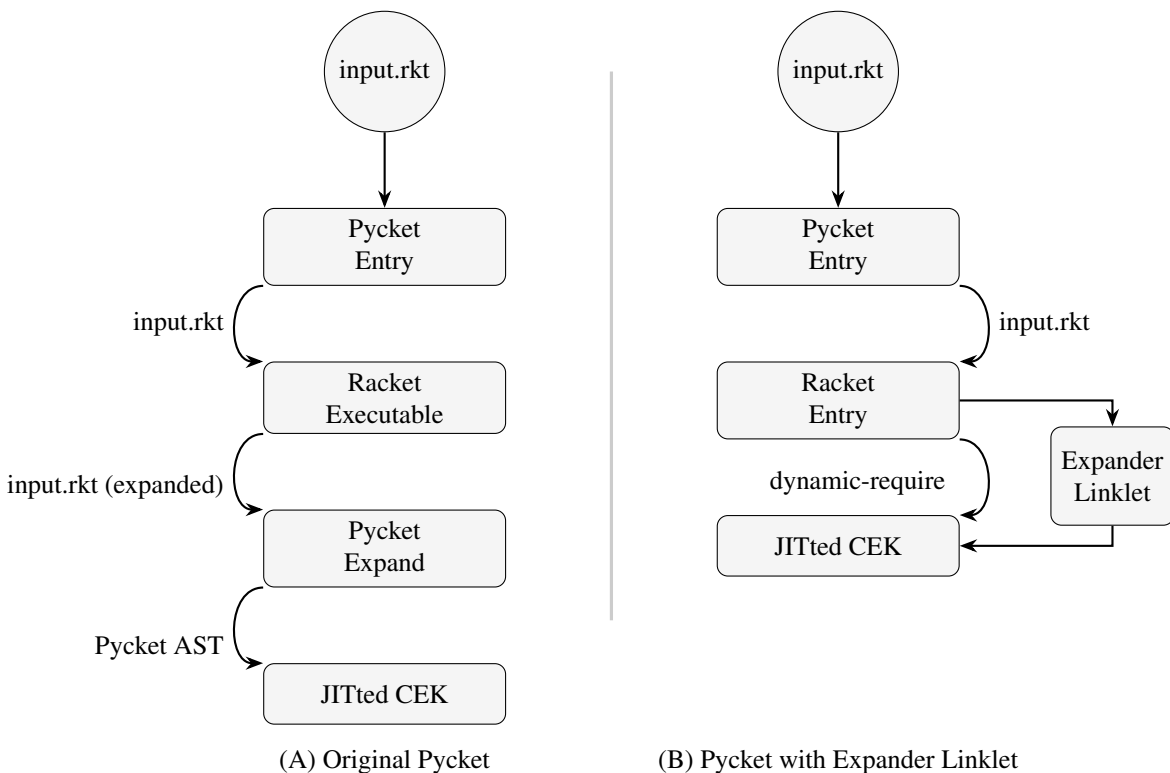


Figure 4.3: Comparison of two Pycket front-ends. (A) The original front-end with offline expansion; (B) The new front-end with expansion in run-time.

The integration of the bootstrapping and custom linklets fundamentally transforms Pycket’s frontend architecture, as depicted in Figure 4.3. Whereas previously Pycket had to invoke the Racket executable externally to expand Racket modules before evaluation, it now applies Racket’s macro expander internally to a given module and evaluates the resulting fully-expanded form—all directly within its CEK core. This architectural shift places all phases of module evaluation—reading, expansion, and execution—fully within Pycket’s runtime, thereby providing self-containment and independence. However, embedding the entire expansion phase internally has substantial implications for runtime performance, a topic examined in depth in Chapter 6.

The bootstrapping linklets provided by Racket depend on primitives supplied by the host runtime—in this case, Pycket. When compiling these linklets, Pycket generates AST nodes for each primitive reference. To ensure successful compilation, these primitives must be resolvable within Pycket’s global primitive environment. While Pycket already implements a many Racket primitives in RPython—as detailed in Chapter 2—some linklets, notably *thread* and *io*, require additional sub-systems such as engines and a Foreign Function Interface (FFI) layer, which will be discussed further in subsequent sections.

Loading bootstrapping linklets into the runtime environment serves as the central mechanism enabling self-hosting. Pycket can read and expand any Racket module by leveraging Racket’s own module and macro systems, provided by the expander linklet. For example, loading a language such as `#lang racket/base` involves the macro expander and module system generating individual linklets for each required module in its dependency tree. As illustrated in Figure 4.4, these modules are compiled and instantiated separately by Pycket. This mechanism is utilized in the next section, where we describe the use of pre-compiled Racket modules to achieve

```

(namespace-require (lib "racket/base")) ...           1
making instance : "'stx"                             2
making instance : "'#%runtime"                       3
making instance : "'qq-and-or"                       4
making instance : "'gen-temp"                        5
making instance : "'cond"                            6
making instance : "'member"                          7
making instance : "'define-et-al"                    8
making instance : "'#%paramz"                       9
making instance : "'#%unsafe"                      10
making instance : "'ellipses"                       11
making instance : "'sc"                             12
... <100 lines removed>
making instance : "'submodule"                       112
making instance : "'generic-interfaces"              113
making instance : "'print-value-columns"             114
making instance : "'kw-syntax-binding"               115
making instance : "'kw-syntax-binding"               116
"module-prefetch: (syntax/readerr) in: #<path>"      117
making instance : "'readerr"                         118
making instance : "'module-reader"                   119
making instance : "'module-reader"                   120
making instance : "(submod 'base reader)"             121
Init lib : racket/base loaded...                     122

```

Figure 4.4: Loading racket/base Modules

significantly faster startup performance.

4.2.1 Interfacing with the Compiler: Expander Linklet

As described in Chapter 3, the expander linklet is generated offline by running the expander on itself, resulting in a serialized s-expression. Pycket reads this serialized s-expression, compiles it into a linklet object using `compile-linklet`, and then instantiates it using `instantiate-linklet`. Once instantiated, Pycket incorporates all exported functions from the expander—such as `read`, `expand`, and `eval`—into its runtime environment, allowing direct invocation of these functions

as native primitives.

```
expander_linklet_obj = compile_linklet([expander_linklet_sexp, ...])
expander_linklet_instance = instantiate_linklet([expander_linklet_obj, ...])
expander_linklet_instance.expose_vars_to_prim_env()
```

Having direct access to the expander’s runtime functions enables Pycket to implement language-level operations such as a top-level REPL through Racket’s own primitives. A minimal implementation of a REPL can be realized simply by invoking Racket’s [read](#), [expand](#), and [eval](#) functions within Pycket, as illustrated in the small inline example below. Furthermore, since the expander linklet provides the complete implementations of Racket’s macro and module systems, Pycket can directly execute programs written in languages layered on top of the Racket core, such as `racket/base`. As a result, instead of implementing a custom REPL, Pycket can directly apply `dynamic-require` to Racket’s standard `racket/repl` module. This yields a REPL functionally identical to Racket’s own, executed within Pycket’s meta-tracing JIT environment.

```
while True:
    r_exp = read.call_racket([repl.readline(), ...])
    r_expanded = expand.call_racket([r_exp, ...])
    result = eval.call_racket([r_expanded, ...])
    print(result)
```

Programs executed through the bootstrapping linklets, including the linklets themselves, often require additional run-time support beyond basic primitives. For instance, running Racket’s REPL necessitates implementing delimited continuations at the run-time level. Similarly, handling exceptions originating from Racket code demands robust Racket-level exception handling within Pycket. Pycket’s existing implementation provides basic exception handling by translating Racket-level exceptions (implemented as Racket structs) into RPython exceptions. However, fully supporting Racket-level exceptions requires Pycket to handle the installation of exception handlers via continuation marks and dynamically propagate Racket-level exceptions to the appropriate (Racket level) handlers.

Among the critical functions exposed by the expander linklet are `namespace-require`, `read`, `expand`, and `eval`, each of which embodies substantial and complex functionality. `eval` functions effectively as an interpreter; `namespace-require` implements the core of the module system; and `read` and `expand` collectively represent the complete macro expander. Thus, the expander linklet itself encapsulates the entire compilation phase required by Racket modules.

Given the complexity and breadth of these functions—including extensive indirection, numerous cross-module references, and deeply nested loops—the resulting behavior severely complicates tracing and optimization by the meta-tracing JIT. Specifically, the meta-tracer and trace optimizer struggle to generate effective, reusable traces from such highly indirect and dynamic code paths. The Chapter 6 is dedicated entirely to analyzing the performance implications of this complexity, detailing specific cases and investigating substantial runtime overhead caused by the introduction of these internally evaluated Racket sub-systems.

4.2.2 Interfacing with the Host Environment: IO & Thread Linklets

Growing into a complete implementation also means the VM must interact closely with the host environment. The thread and IO linklets handle this interaction. Specifically, the thread linklet provides user-level (green) threads, while the IO linklet implements comprehensive input/output capabilities ranging from file and port abstractions to high-level Racket primitives like `write`. Collectively, these two linklets expose over 300 primitives, with the IO linklet explicitly relying on some primitives defined within the thread linklet.

Some bootstrapping linklets—including thread and IO—require specialized runtime support because they assume particular functionalities or depend on external system-level operations.

The thread linklet, for instance, requires the engines subsystem, and the IO linklet depends on the foreign function interface (FFI) to interact with the external *rktio* C library. Additionally, these linklets define and expose numerous internal primitives previously implemented as stubs in Pycket (e.g., `unsafe-start-atomic`). With the full integration of these linklets, such primitives now operate as originally intended.

Thread Linklet

The thread linklet provides the implementation for Racket’s user-level (green) threads, supporting preemption through engines. At a high level, it defines useful Racket abstractions such as channels, threads, futures, and places. At a lower level, it implements essential synchronization and concurrency primitives including semaphores, custodians, and critical-section operations such as `start-atomic` and `end-atomic`. Because these features are entirely implemented within the linklet, any host runtime, such as Pycket, only needs to provide minimal primitive support required by the linklet to gain all these high-level functionalities for essentially free.

Engines constitute one of the key primitive subsystems required by the thread linklet. Similar to an engine powering a vehicle, engines drive thread execution in Racket. Engines abstract the concept of timed preemption, providing a low-level mechanism for implementing time-sharing among arbitrary process abstractions. An engine is not itself a process abstraction; instead, it enables the time-sharing of any given computation. Within Racket threads, engines act as the computational core—each thread is associated with an engine that performs its assigned computation when provided with sufficient *fuel*. Engines can be interrupted or blocked either by exhausting their allocated fuel or by external events, upon which a new engine is returned,

encapsulating the remaining computation. This interrupted computation can subsequently be resumed by supplying additional fuel to the new engine instance [35].

To support full preemption, Racket’s engines differentiate between Racket-level continuations and host continuations through the concept of *meta-continuations*. Meta-continuations represent segments of the host’s continuation stack, delimited by prompt tags to define the dynamic extent of an engine’s computation. When an engine is interrupted, the host continuation stack must be properly unwound to reflect the interruption of the corresponding meta-continuation. Conversely, resuming an engine requires reinstalling the saved meta-continuation onto the host continuation stack, restoring the execution state to the point where it was previously suspended.

Pycket implements engines to provide the necessary primitive support for the thread linklet. However, Pycket does not support full preemption; rather, engines run to completion once started. This design choice stems from Pycket’s underlying RPython framework, which assumes a single-threaded runtime protected by a Global Interpreter Lock (GIL). Consequently, full integration with OS-level preemption via pthreads is unnecessary and intentionally avoided. Instead, Pycket relies entirely on cooperative scheduling, where threads yield control explicitly through Racket-level primitives such as channels, semaphores, or synchronization points.

With thread support enabled, all operations in Pycket run within the context of a Racket-level thread, initiated through the primitive `call-in-main-thread`. Additional threads are spawned from this main thread, including a thread responsible for running a REPL, as well as any threads created during interactions within that REPL. To illustrate this clearly, consider the following example within Pycket’s REPL:

```
Welcome to Pycket v8.17.0.3
> (define ch (make-channel))
> (thread (lambda () (channel-put ch 3)))
#<thread>
> (channel-get ch)
3
>
```

While seemingly straightforward, the example above demonstrates important runtime behavior. The value transferred from the newly created thread to the main thread (running the REPL) via the channel illustrates explicit thread synchronization. Specifically, for a channel to facilitate such a transfer, both the sender and receiver threads must be suspended during the operation. In this example, the main thread initially suspends when it invokes `channel-get`, awaiting the new thread to produce a value. The new thread runs and invokes `channel-put`, placing a value into the channel and suspending itself. Subsequently, control returns to the main thread, retrieving the value from the channel. After the transfer, the newly created thread resumes briefly to complete its execution, and finally, the main thread resumes fully, returning control to the REPL prompt.

The thread linklet also provides primitives utilized by other bootstrapping linklets, notably the IO linklet. As a result, the thread linklet is loaded earlier in Pycket’s startup sequence, ensuring all threading primitives are available when the IO linklet is instantiated. This ordering guarantees that thread-related primitives required by IO operations—such as synchronization primitives and channels—are already established within Pycket’s global primitive environment during the IO linklet’s instantiation phase.

IO Linklet

The IO linklet implements Racket’s extensive input/output capabilities, covering abstractions such as file systems, networking, and subprocess management. It exports essential high-level Racket primitives such as `write`, as well as lower-level facilities for byte handling and data encoding. Internally, the IO linklet depends on the thread linklet, leveraging 49 primitives provided by it to implement thread-safe IO operations.

To interact effectively with underlying operating system facilities, the IO linklet relies on a dedicated C library called *rktio*. This standalone library provides a unified, portable interface to critical OS-level features—including file systems, networking, and subprocess management—exposing over 200 functions implemented in C. Rather than implementing OS abstractions directly within the linklet, the IO linklet delegates these operations to the *rktio* layer, maintaining a clean separation between Racket-level abstractions and low-level system interactions.

To utilize the IO linklet, Pycket must load and expose *rktio*’s functionality to the Racket runtime. Since Pycket is implemented in RPython, it achieves this by creating an FFI layer that bridges Racket code with the underlying *rktio* library. This integration is facilitated by automatically generating wrapper functions around every *rktio* C function from the provided *rktio* registry (`rktio.rktl`). These wrappers are exposed as primitives within Pycket’s global primitive environment, allowing the IO linklet to transparently invoke OS-level operations as if they were regular Racket primitives.

Implementing this FFI interface requires establishing clear type mappings between three distinct type domains: the native `rktio` C types, Pycket’s internal Racket-level types (`W_Object` and its derivatives), and the primitive RPython `rffi` types that map directly to C types during code generation. Table 4.2 summarizes some notable examples. Pycket’s FFI implementation must carefully manage these mappings to correctly handle memory operations, data conversion, and error propagation across language boundaries, ensuring accurate and efficient interaction between the Racket runtime and the underlying `rktio` library.

rktio	Pycket	rffi
<code>rktio_ok_t</code>	<code>W_Fixnum</code>	<code>rffi.INT</code>
<code>rktio_bool_t</code>	<code>W_Bool</code>	<code>rffi.INT</code>
<code>rktio_const_string_t</code>	<code>W_Bytes</code>	<code>rffi.CCHARP</code>
<code>rktio_filesize_t</code>	<code>W_Fixnum</code>	<code>rffi.LONGLONG</code>
<code>intptr_t</code>	<code>W_Fixnum</code>	<code>rffi.SSIZE_T</code>
<code>uintptr_t</code>	<code>W_Fixnum</code>	<code>rffi.SIZE_T</code>
<code>unsigned-8</code>	<code>W_Fixnum</code>	<code>rffi.UNSIGNED</code>
<code>(ref void)</code>	<code>W_CPointer</code>	<code>rffi.VOIDP</code>
<code>(*ref (ref char))</code>	<code>W_STAR_REF_CCHARP</code>	<code>rffi.CCHARP</code>

Table 4.2: Some notable type mappings for *rktio* FFI layer

In addition to simple type mappings, the interface between `rktio` and Racket involves a small but significant Application Binary Interface (ABI) convention for pointer representations. Specifically, `rktio` differentiates between memory regions that are opaque to Racket (referred to as *ref*) and those transparent to Racket (referred to as **ref*). Opaque references represent pointers to memory whose layout and lifetime are fully managed by the C library, allowing Pycket to treat these as generic pointers (`rffi.VOIDP`). Transparent references, on the other hand, denote memory explicitly allocated, owned, and managed by Racket. For example, an

rktio C function expecting a **ref char* input indicates it will mutate memory provided by Racket (such as a bytes object). Pycket must therefore ensure that any in-place mutations performed by the rktio function are accurately reflected in Racket-level representations, like `W_MutableBytes`, maintaining consistency between the C library’s operations and Racket’s view of the data.

With these type mappings and conventions in place, Pycket automatically generates the majority of the rktio FFI wrapper functions directly from the rktio registry (`rktio.rktl`). Over 200 functions are automatically generated, while 26 additional primitives requiring special handling—such as complex struct access, specialized pointer dereferencing, or detailed error management—are implemented manually. To accommodate complex structures like `rktio_date_t`, Pycket includes specialized primitive definitions that form the connector layer. This connector layer explicitly manages struct layouts and pointer dereferencing, ensuring correct interactions between Pycket and rktio. Collectively, these automated and manually implemented primitives constitute a robust and comprehensive bridge, enabling seamless and efficient integration of the IO linklet into Pycket’s runtime environment.

4.2.3 Language Utilities: Fasl & Regexp Linklets

Racket also provides several utility linklets, notably the *fasl* and *regexp* linklets. The *fasl* linklet provides serialization and deserialization primitives (`fasl→s-exp` and `s-exp→fasl`), and the *regexp* linklet provides Racket’s regular expression facilities. Compared to other bootstrapping linklets, these utility linklets are smaller but essential for Pycket’s frontend operation. For example, Pycket employs `fasl→s-exp` to deserialize and load the initial bootstrapping linklets themselves.

Previously, the functionalities provided by these utility linklets were manually implemented in RPython within Pycket. With the introduction of linklets, Pycket replaces its internal RPython implementations with Racket’s own implementations of `fasl` and regular expressions. This architectural shift allows for direct comparison between Pycket’s native RPython implementations and the corresponding Racket implementations in terms of meta-tracing performance, an analysis thoroughly explored in the performance evaluations presented in Chapter 6.

These utility linklets further emphasize the modular, plug-and-play nature of the linklet-based architecture on Pycket. Once loaded, Racket’s native implementation becomes the active provider of the corresponding primitives, such as `regexp-match`. Any subsequently instantiated linklets—including critical ones like the expander—transparently utilize these primitives without being aware of their underlying implementation. If, however, these utility linklets are not loaded, Pycket defaults gracefully to using its internal RPython implementations. Thus, the primitives exposed in Pycket’s global primitive environment may originate either from RPython directly or from a loaded Racket linklet, without affecting the operation of dependent modules, given that both implementations are semantically equivalent.

4.3 Growing Pycket into Full Racket

With all the bootstrapping linklets successfully loaded into its runtime, Pycket completes its transition into a fully operational, self-hosting Racket implementation. This marks the point where Pycket no longer relies on external Racket executables for critical runtime functionality, as it now internally provides the necessary components—such as macro expansion, module handling, and I/O—required to independently load and run Racket modules. Consequently,

```
(namespace-require (lib "racket/base")) ...  
Init lib : racket/base loaded...  
  
Welcome to Pycket v8.17.0.3  
>
```

Figure 4.5: Welcome to Pycket!

Pycket now fulfills the self-hosting hypothesis introduced earlier.

Pycket’s enhanced front-end supports a variety of execution modes, including directly running a provided `.rkt` source file or launching an interactive session to start the Racket REPL, as illustrated in Figure 4.5. These capabilities demonstrate that Pycket integrates core Racket functionality via bootstrapping linklets and exposes standard user-facing features. Furthermore, as new features are added to Racket itself, Pycket can readily incorporate them by simply running the corresponding Racket modules. This capability ensures Pycket remains robust and future-proof as a complete implementation of the language.

The REPL provided by Pycket here is not custom-written; rather, it is the standard Racket REPL module (`racket/repl`), which Pycket dynamically loads by calling `dynamic-require` (provided by the `expander` linklet) at startup. Running the REPL necessitates loading the entire `racket/base` language immediately prior to executing Racket’s REPL implementation, thereby demonstrating Pycket’s capability to independently expand and evaluate sophisticated Racket language modules directly within its runtime.

```

(namespace-require (lib "racket/base")) ...      1
making instance : ''#%paramz"                  2
making instance : ''#%unsafe"                   3
making instance : ''#%flfxnum"                  4
making instance : ''require-transform"          5
making instance : ''#%runtime"                  6
making instance : ''#%place-struct"             7
making instance : ''#%utils"                    8
making instance : ''#%boot"                     9
making instance : ''#%expobs"                   10
making instance : ''#%linklet-primitive"        11
making instance : ''#%linklet-expander"         12
making instance : ''#%linklet"                  13
making instance : ''#%foreign"                  14
making instance : ''#%extfl"                    15
making instance : ''#%network"                  16
making instance : ''#%place"                    17
making instance : ''#%futures"                  18
making instance : ''#%terminal"                 19
making instance : ''#%builtin"                  20
making instance : ''kw-syntax-binding"          21
making instance : ''#%unsafe"                   22
Init lib : racket/base loaded...                 23

```

Figure 4.6: Loading racket/base Modules Using Compiled Code

Figure 4.6 illustrates loading the racket/base language during Pycket’s startup with debug output enabled. Notably, the individual modules previously loaded explicitly via linklets no longer appear in this process (compare with Figure 4.4). This change occurs because Pycket leverages the `write` and `fasl` functionalities provided by Racket to serialize loaded modules into compiled `.zo` files. A `.zo` file is Racket-specific bytecode stored in `fasl` format. By supplying the Racket parameter `(use-compiled-file-paths compiled/pycket)` to the module system at boot time, Pycket loads these serialized modules directly into memory, bypassing the expensive linklet compilation and instantiation steps. This optimization significantly reduces the loading

time of racket/base—from approximately 2 minutes down to about 7 seconds—achieving around a 95% improvement in startup performance.

Moreover, the availability of the expander linklet enables Pycket to execute higher-level languages built on top of racket/base, such as the full `#lang racket`. For example, consider the Racket program shown in Figure 4.7, which utilizes Racket contracts. This program cannot run directly using only racket/base, as it requires additional modules from full Racket. However, Pycket successfully executes it, demonstrating its capability to load and evaluate sophisticated language-level constructs and further validating its status as a complete Racket implementation.

```
1  #lang racket
2
3  ;; fibonacci with contract
4  (define/contract (fib n)
5    (→ exact-nonnegative-integer?
6        exact-nonnegative-integer?)
7    (match n
8      [0 0]
9      [1 1]
10     [_ (+ (fib (- n 1)) (fib (- n 2)))]))
11
12  (module+ main
13    (displayln (for/list ([i (in-range 10)])
14                  (fib i))))
```

Figure 4.7: A Racket program that uses entire `#lang racket`

With this, we conclude the first part of our thesis statement, the self-hosting hypothesis. We demonstrated concretely that it is possible to evolve a rudimentary interpreter into a full-featured, self-hosting implementation of a functional programming language on a meta-tracing JIT

compiler. In the subsequent chapters, we will discuss how correctness of these developments is ensured, examine performance characteristics of the resulting system, investigate a fundamental performance issue uncovered in this self-hosting setup, and finally propose and evaluate several approaches to address this issue.

5. PYCKET AS A FULL RACKET RUN-TIME: CORRECTNESS & COMPLETENESS

CHAPTER SYNOPSIS

We demonstrate three things in this chapter:

1. Self-hosted implementation of Racket on Pycket works.
2. Semantics of the self-hosted Racket on Pycket are identical to the semantics of Racket on Chez Scheme (Racket's own default runtime).
3. You can evaluate on Pycket anything you can express in Racket.

Sections:

- Correctness by Construction

Semantics of the self-hosted Racket on Pycket are identical to the semantics of Racket on Chez Scheme (Racket's own default runtime).

- Completeness under Self-Hosting

Pycket can evaluate any #lang that Racket can evaluate.

In this chapter, we demonstrate that Pycket’s self-hosted implementation of Racket is both correct and complete with respect to the original Racket implementation. Correctness here means that the semantics provided by Pycket exactly match those provided by Racket’s default runtime on Chez Scheme, and completeness means that any program expressible in Racket can be successfully evaluated by Pycket. Establishing these two properties is essential to demonstrate that Pycket can indeed serve as a fully functional drop-in replacement for Racket.

As the first step of our validation, we use a moderately sized program, shown in Figure 5.1, as an example to illustrate that Pycket’s runtime behavior is indistinguishable from Racket running on Chez Scheme. This program works correctly on both implementations.

For the program in Figure 5.1 to correctly run on Pycket, multiple essential components must function together seamlessly. At the outset, Pycket must support the full loading and evaluation of a `#lang racket` module, which alone involves significant runtime support for nearly all Racket core values and data structures. This includes handling numerical values, strings, symbols, hash tables, vectors, boxes, and particularly structs, which are fundamental to syntax objects used extensively by Racket’s macro expander. Indeed, before considering linklets, Pycket must already have robust runtime support for syntax objects—heavily reliant on Racket structs—to correctly handle macros such as `define-syntax`, `syntax-case`, and phase-level forms like `for-syntax`. Moreover, because the macro expander itself utilizes linklets to manage dependencies across different compilation phases, Pycket must correctly instantiate and evaluate these linklets, ensuring accurate handling of variable binding, closures, and across-phase interactions. Additionally, to correctly evaluate advanced control-flow mechanisms such as `shift0` and `reset0`, Pycket’s runtime must provide reliable and efficient support for delimited

```

1  #lang racket
2
3  (require racket/control racket/stxparam (for-syntax racket/base))
4
5  (define-syntax-parameter acc
6    (lambda (stx) (raise-syntax-error #f "used outside of with-accum" stx)))
7
8  (define-syntax (with-accum stx)
9    (syntax-case stx ()
10     [(_ body ...)
11      (with-syntax ([acc-id (datum->syntax stx (gensym 'acc))])
12        #'(let ([acc-id (box '())])
13            (syntax-parameterize ([acc (make-rename-transformer #'acc-id)])
14              (reset0 (begin body ...)))
15              (unbox acc-id)))))]))
16
17  (define-syntax-rule (accumulate! v)
18    (shift0 k (begin (set-box! acc (cons v (unbox acc))) (k (void)))))
19
20  (define (macro-delim-demo)
21    (with-accum
22      (for ([i (in-range 5)]) (when (even? i) (accumulate! i)))
23      (for ([j (in-range 3)]) (accumulate! (* 100 j)))))
24
25  (displayln (macro-delim-demo))

```

Figure 5.1: Example `#lang racket` program running on Pycket.

continuations. Further, the runtime must comprehensively support primitive operations required by the fully expanded program, including boxing/unboxing operations, structural equality checks, and errors. Since Pycket successfully executes this substantial program, it demonstrates the correctness and completeness of Pycket’s implementation of all these interdependent components.

The semantic equivalence between Racket running on Chez Scheme and Pycket ultimately relies on Pycket’s correct handling of the linklet semantics, as well as the correctness of its

primitive operations and core data structures exposed to linklets. Given these foundational components, completeness naturally follows from Racket’s layered language architecture, in which richer language constructs and functionalities are incrementally constructed atop more primitive layers. The following two sections detail the evidence supporting both correctness and completeness.

5.1 Correctness by Construction

Correctness by construction in Pycket begins with an already comprehensive and correct implementation of Racket’s core values and data structures. This includes the full numeric tower, symbols, hash tables, vectors, and structs, as described in detail in [7]. We build our high-level correctness argument upon this robust correctness of these fundamental elements.

Pycket ensures correctness of its primitive operations through an extensive suite of unit tests. Table 5.1 shows the number and scope of Pycket’s test suites, forming a total of 549 suites, each containing numerous individual test cases. Among all the baseline semantics, these suites comprehensively cover significant runtime components such as structs, hashes, equality, hidden classes, impersonators, regular expressions, AST handling, and linklets.

We considered leveraging Racket’s own comprehensive test suites to further demonstrate Pycket’s correctness. However, since Racket’s testing infrastructure heavily relies on external tooling such as `raco` (e.g., `raco test -p racket-test-core`), integrating it directly into Pycket’s testing workflow would exceed the scope of our study. Instead, we focus on targeted, manually adapted tests that precisely verify the critical aspects of our implementation.

Name	# Suites	Concepts Tested
test_arithmetic	54	(+, −, ×, ÷), numeric predicates, exact/inexact, division/remainder/modulo, gcd, lcm
test_ast	22	AST node creation, syntax tree structure, Pycket AST transformations
test_basic	61	Core Racket constructs, literal handling, sequencing and evaluation
test_conses	5	List operations ('cons', 'car', 'cdr'), list structure and mutation
test_entry_point	26	Pycket entry-point behavior, options & flags
test_equality	2	Equality predicate equal-always?
test_hash	40	Hashing mechanisms, 'hash', 'equal-hash', map/dictionary keys integrity
test_hidden_classes	4	Hidden class optimization, object layout, property access paths
test_impersonators	21	Object impersonation, proxies, type and identity behaviors
test_json	6	JSON parsing and serialization, representation of Racket values in JSON
test_larger	20	Larger-scale integration: multiple concepts (e.g. nqueens)
test_linklet	48	Linklet definition/loading, module-level linking semantics
test_mod	13	Module system, definitions and imports, namespace resolution
test_old_entry_point	24	Legacy Pycket entry-point behavior
test_prims	77	Primitive operations, core runtime primitives and their semantics
test_regexp	15	Regular expression compilation and matching, character classes, flags
test_regression	9	Regression test cases, bug fixtures across versions
test_string	31	String operations: concatenation, length, substring, comparison
test_struct	41	'struct' definitions, accessor/mutator behavior, identity and copying
test_vector	30	Vector operations: creation, indexing, mutation
Total	549	

Table 5.1: Summary of Pycket Test Suites

A crucial component of ensuring Pycket’s correctness is validating its implementation of the linklet semantics. The correctness of linklets is particularly important because linklets constitute the primary compilation units upon which Racket’s macro expander and module system are constructed. In the rest of the section, we explain the methodology employed to ensure that Pycket correctly implements these semantics, aligning precisely with the semantics defined by Racket itself.

Recall that the operational semantics we presented for linklets in Chapter ?? were developed using PLT Redex, which allows us to create an executable model of these semantics. The complete linklet model is provided in Appendix C. In the remainder of this section, we describe how we leveraged this executable model to validate Pycket’s implementation of linklet semantics.

To ensure correctness of Pycket’s implementation of linklet semantics, we manually adapted Pycket’s own linklet tests into the program form defined in our PLT Redex model (refer to Figure 3.8 for the definition of the program form). This adaptation enabled us to create a symmetric testing setup, allowing us to run each test across Racket itself, Pycket, and the Redex model simultaneously. Running these tests concurrently ensures that all implementations consistently agree on the expected operational semantics.

```
1  (test-equal
2    (term
3      (eval-prog
4        (program (use-linklets
5                  [l1 (linklet () (x) (define-values (x) 1))]
6                  [l2 (linklet ((x)) (y g)
7                              (define-values (y) 10)
8                              (define-values (g) (lambda (p) (+ x y)))
9                              (set! y 50))]
10                 [l3 (linklet () (y g)
11                       (set! y 200)
12                       (g))]
13                 (let-inst t1 (linklet-instance ()))
14                 (let-inst l1 (instantiate l1))
15                 (instantiate l2 l1 #:target t1) ; fill in the target
16                 (instantiate l3 #:target t1))))
17    201)
```

Figure 5.2: An example test case for linklet semantics

An example test case (adapted from Pycket’s suite into the Redex model) for the correctness of linklet semantics is shown in Figure 5.2. Here, the linklet **I2** exports a closure *g*, which references a variable *y* initially defined as 10 and subsequently mutated to 50 within the linklet (line 9). Since *y* is exported and mutated within **I2**, the compiler treats it as a linklet variable. Similarly, in linklet **I3**, the identifier *y* is exported without a corresponding local definition and is also mutated (line 11), thus compiled as a linklet variable as well. After compilation, we first instantiate **I2** with the target instance *t1* (line 15), populating it with exported variables including the closure *g* and the variable *y*. We then instantiate **I3** using the same target instance *t1*. Any reference to *y* within **I3** thus accesses and modifies the shared linklet variable in the target instance. At the time **I3** is instantiated, the value of *y* is 50, but before invoking the closure *g*, **I3** updates *y* to 200 (line 11). When the closure *g* is finally invoked (line 12), it yields a result of 201, highlighting the use of the updated linklet variable *y*, despite lexical scoping; otherwise, without linklet variables, the closure would have used the original lexical environment in which *y* was 10, returning 11. Notably, after this invocation, the variable *y* in the target instance *t1* remains modified to 200, which can be confirmed by invoking `(instance-variable-value t1 'y)`, demonstrating correct handling of linklet variables across instances.

5.1.1 Randomized Testing for Linklet Semantics

In addition to manually adapted test cases, we also utilized PLT Redex’s support for randomized testing to further verify the correctness of our implementation of linklet semantics. Redex enables generation of random terms according to a provided grammar, which are then checked against a correctness predicate. In our setup, we leverage randomized testing separately for two

executable models: the core language model (LKL) and the linklets model. For each randomly generated term, we asked whether the semantics defined by our model agreed with the actual implementation provided by Racket.

Specifically, we tested two correctness predicates, implemented as meta-functions within our Redex models: `eval-rc=racket-core`, defined for the LKL language (Figure ??), and `eval-prog=racket-linklets`, defined for the linklets language (Figure 3.3). The meta-function `eval-rc=racket-core` receives a randomly generated term in the LKL language, then simultaneously evaluates it using both the Redex model’s evaluator and the actual Racket interpreter (on which PLT Redex itself runs). The results are compared to verify that both implementations produce identical outcomes starting from an empty environment. Similarly, the `eval-prog=racket-linklets` predicate performs an analogous check for the linklets language, ensuring semantic consistency between our linklets model and Racket’s own implementation.

During randomized testing, we encountered a notable challenge due to the difference between the Redex language grammars and Racket’s actual syntax. Specifically, the grammars for both the LKL and the linklets languages allow the explicit representation of certain runtime entities—most notably closures—that do not have direct syntactic counterparts in Racket itself. Closures in both Racket and the LKL language consist of formal parameters, a function body, and the environment captured at closure creation time. However, Racket’s surface syntax does not allow for explicit construction of closures, as environments lack a direct syntactic representation. Consequently, randomly generated terms containing explicit closures (e.g., `(closure x (+ x y) ((y cell123))))`) would lead to runtime exceptions when evaluated by Racket’s interpreter, even though they were successfully evaluated by our Redex model.

```

e-test ::= x
          | n
          | b
          | (void)
          | ( e-test e-test ... )
          | (lambda (x! ...) e-test)
          | (if e-test e-test e-test)
          | (p2 e-test e-test)
          | (p1 e-test)
          | (set! x e-test)
          | (begin e-test e-test ...)
          | (let-values (((x) e-test) ...) e-test)
          | (raises e-test)

```

Figure 5.3: Restricted LKL expression grammar for testing

To mitigate this issue and reduce the frequency of errors caused by such randomly generated cases, we slightly restricted the grammar used by the random term generator. Specifically, we removed explicit closure terms (the non-terminal *c*) from the LKL grammar for randomized testing, ensuring that all randomly generated terms have direct syntactic representations in Racket. The restricted grammar for these LKL expressions is shown in Figure 5.3. Using this restricted grammar, we performed randomized testing with 1000 freshly generated terms, and successfully confirmed that our LKL model agrees with the actual Racket implementation. As we discuss later, the grammar restriction for the linklets language required a slightly more involved adjustment.

Testing the linklets model required a more elaborate approach compared to testing the LKL model, for two main reasons. First, because the terms are randomly generated, the linklets model needed to robustly handle numerous runtime corner cases inherent to module-like constructs. Imagine randomly generating complete Racket modules using the full Racket grammar and evaluating them: while such modules would be syntactically valid, they would frequently yield runtime errors due to undefined identifier references, function arity mismatches, or type errors—such as attempting arithmetic between incompatible types. Second, while terms generated from the LKL grammar are directly evaluable in both Racket and the Redex model without transformation, terms generated from the linklets grammar include forms (such as the top-level program form) that are not directly recognized by Racket’s evaluator, thus necessitating additional steps to enable evaluation in Racket.

$$\begin{aligned}
& \text{eval-prog} \llbracket (\text{program } (\text{use-linklets } (x_L, L) \dots) p_{\text{top}} \dots) \rrbracket \\
& = \text{run-prog} \llbracket ((\text{program } (\text{use-linklets } (x_L, L) \dots) p_{\text{top}} \dots) () () ()) \rrbracket \\
& \quad \text{where } \left(\begin{array}{l} \text{and(term check-free-varss} \llbracket L, \dots \rrbracket, \\ \text{term no-exp/imp-duplicates} \llbracket L, \dots \rrbracket, \\ \text{term no-export-rename-duplicates} \llbracket L, \dots \rrbracket, \\ \text{term no-non-definable-variables} \llbracket L, \dots \rrbracket, \\ \text{term no-duplicate-binding-names} \llbracket L, \dots \rrbracket, \\ \text{term linklet-refs-check-out} \llbracket (p_{\text{top}} \dots), (x_L \dots), \\ \text{get-defined-instance-ids} \llbracket (p_{\text{top}} \dots), () \rrbracket \rrbracket) \end{array} \right)
\end{aligned}$$

Figure 5.4: Initial checks before the evaluation begins

To handle the first challenge, where many randomly generated inputs were semantically invalid, we had two potential solutions. The first option was to employ a mechanism provided by Redex, such as the `#:prepare` argument in `redex-check`, which allows filtering generated

terms to discard those that are semantically invalid before applying the predicate. The second option was to directly embed checks within the linklets evaluator model to explicitly identify and handle these error cases. While the first approach would have provided a streamlined testing, we chose the second approach, as it aligns our model more closely with Racket’s own implementation. Specifically, we inserted preemptive checks into our linklets evaluator to detect and appropriately handle common semantic errors. The downside of this approach is that a significant portion of randomly generated test cases end up being rejected by both our model and Racket’s implementation, thus reducing the proportion of interesting cases that fully exercise the instantiation semantics. However, this trade-off ensures that our model accurately mirrors the behavior of Racket’s own system, which detects such semantic errors during the compile-linklet phase, where linklet objects are constructed from their s-expression representations. In contrast, our Redex model does not have a separate compilation phase; instead, the representation of the object is the object itself, as evaluation directly operates on Redex meta-terms. The details of these checks are shown in Figure 5.4.

To address the second issue—where randomly generated terms from the linklets grammar include forms not directly evaluable by Racket—we implemented a straightforward converter. This converter rewrites program forms from our linklets language into equivalent Racket syntax, utilizing standard constructs such as `let` & `define`, as demonstrated in Figure 5.5. This transformation allows us to evaluate generated terms directly using Racket’s evaluator, thereby enabling accurate semantic comparisons between Racket and our Redex model.

As noted previously for testing the LKL model, another complication in random testing arises from generating terms that include entities without explicit representations in Racket. Just as

```

1  > (term
2    (to-actual-racket
3      (program (use-linklets
4                [l1 (linklet () ())]
5                  [l2 (linklet ((b)) () (define-values (a) 5) (+ a b)])
6                  [l3 (linklet () (b) (define-values (b) 3)])]
7                (let-inst t3 (instantiate l3))
8                (let-inst t1 (instantiate l1))
9                (instantiate l2 t3 #:target t1))))))
10
11  '(let ((l1 (compile-linklet '(linklet () ())))
12        (l2 (compile-linklet '(linklet ((b)) () (define-values (a) 5) (+ a b))))
13        (l3 (compile-linklet '(linklet () (b) (define-values (b) 3))))
14        (define t3 (instantiate-linklet l3 (list)))
15        (define t1 (instantiate-linklet l1 (list)))
16        (instantiate-linklet l2 (list t3) t1))

```

Figure 5.5: Meta-function converting the *program* form to Racket

we addressed this issue by restricting the LKL grammar for randomized tests, we similarly restricted the grammar for the linklets language. Specifically, we removed explicit closure values and linklet instance terms from the grammar. Linklet instances, like closures, contain runtime-specific entities (variables mapped to cells), which cannot be directly expressed in Racket’s syntax. In place of these instance literals, our tests instead use linklet literals combined with the `instantiate` form, producing the desired linklet instances at runtime. The resulting restricted grammar for randomly generated linklet programs is shown in Figure 5.6.

Using this adjusted grammar, and increasing our coverage by running tests with 2000 freshly generated terms each time, we successfully confirmed that our linklets model aligns precisely with Racket’s own linklet implementation (`racket/linklet`). This consistency between the Redex model and Racket’s implementation confirms the correctness of Pycket’s implementation

$$\begin{aligned}
\textit{linkl-ref-test} &::= x \mid L \\
p\text{-test} &::= (\textit{program} \ (\textit{use-linklets} \ (x_L) \ \dots) \ p\text{-top-test} \ \dots \ \textit{final-expr-test}) \\
p\text{-top-test} &::= T\text{-test} \mid n \mid b \mid (\textit{void}) \mid \textit{stuck} \mid (\textit{let-inst} \ x \ I\text{-test}) \\
I\text{-test} &::= (\textit{instantiate} \ \textit{linkl-ref-test} \ x \ \dots) \\
T\text{-test} &::= (\textit{instantiate} \ \textit{linkl-ref-test} \ x \ \dots \ \textit{\#target} \ x) \\
\textit{final-expr-test} &::= n \mid b \mid (\textit{void}) \mid \textit{stuck} \mid (\textit{instance-variable-value} \ x \ x) \mid T\text{-test}
\end{aligned}$$

Figure 5.6: Restricted Linklet *program* grammar for testing

of linklet semantics.

In summary, by combining the correctness of Pycket’s foundational core values, data structures, and primitive operations with validation of its linklet semantics via targeted and randomized testing, we have ensured that Pycket accurately evaluates the linklet forms generated by Racket’s macro expander. The targeted tests—adapted to a symmetric setup—were simultaneously executed on Racket, Pycket, and our Redex model, further strengthening our confidence in Pycket’s correctness by ensuring consistency across all implementations. Moreover, incorporating untouched Racket code into Pycket, code already validated on Racket’s own runtime, effectively serves as integration testing. This provides additional assurance of correctness for any runtime that imports these functionalities via linklets.

5.2 Completeness under Self-Hosting

Racket is not only a programming language, but also a language-building platform. It provides an integrated framework—including a module system, a hygienic macro expander with syntax objects, customizable parsers via `#lang`, and a rich runtime—that makes it straightforward to

create new domain-specific or general-purpose languages. As Felleisen et al. put it, Racket’s design “supports a smooth path from relatively simple language extensions to completely new languages” [36].

Because Pycket correctly evaluates Racket’s expander—imported among the bootstrapping linklets—it inherits this full language-construction capability. The expander takes any `#lang` module and translates it down to linklets, which Pycket then evaluates using the primitive and linklet semantics we’ve already validated. Thus, any language built on Racket—for instance, Typed Racket, Scribble, or even educational languages—becomes directly executable in Pycket (a meta-tracing JIT compiler), just as it is in native Racket.

In essence, Pycket’s ability to successfully import and evaluate the expander forms the minimal sufficient condition for full Racket compatibility. From `##kernel` through `#lang racket/base` up to arbitrary `#lang` dialects, all macro phases, parser extensions, and runtime behaviors—including delimited continuations, contracts, closures, and error reporting—are retained. This validates Pycket as a complete drop-in Racket runtime.

In conclusion, this chapter has demonstrated that Pycket is both correct and complete as a fully self-hosting Racket run-time. We established correctness through foundational unit testing, careful validation of linklet semantics using targeted and randomized tests, and consistency checks across Pycket, Racket, and our executable Redex model. Completeness follows naturally from Racket’s layered architecture, which Pycket accurately supports via correct evaluation of the imported macro expander and the module system. Ultimately, the substantial example provided at the beginning of this chapter (Figure 5.1) decisively confirms our initial claim:

Pycket successfully executes non-trivial `#lang racket` programs, thereby validating its role as a drop-in, functionally transparent Racket run-time.

6. PERFORMANCE EVALUATION OF SELF-HOSTING ON META-TRACING

CHAPTER SYNOPSIS

Self-hosting on a meta-tracing JIT compiler introduces several performance issues.

Sections:

- 6.1 Pycket's Performance Characteristics

Pycket is generally fast, but self-hosting exacerbates issues in tracing interpreter-style code.

- 6.2 Module and Language Loading

Self-hosting adds a "boot and language expansion" overhead, which Pycket didn't have before.

- 6.3 Cost of Self-hosting

While self-hosting introduces minimal overhead for back-end performance, it degrades front-end performance; subsequent sections investigate the underlying reasons.

- 6.4 The Nature of the Beast: Tracing Data-Dependent Computation

Data-dependent, interpreter-style computations essential to self-hosting inherently produce overspecialized and inefficient traces under meta-tracing.

- 6.5 Memory Pressure under Self-hosting

Late binding, long continuation chains due to numerous control-flow indirections, and frequent allocation of large, long-lived heap objects significantly increase GC pressure in self-hosting on meta-tracing.

Self-hosting introduces performance issues on a language runtime in several ways. Data-dependent, interpreter-style, branch-heavy code produces overspecialized, non-reusable traces; self-hosting raises memory-usage concerns; and computations that are critical to self-hosting—such as program expansion—seem to be poorly suited to meta-tracing because they contain no hot loops for the tracer to capture. Although the exact symptoms may vary with the design of a given runtime and the hosted language, the underlying issues remain the same on every meta-tracing system.

In this chapter we study in detail and demonstrate the impact of these performance issues on a concrete system, namely *Racket on Pycket*. Recall from Chapter 4 that the original Pycket front-end invoked the stand-alone Racket executable to load core libraries, as well as the `#lang-language`, and fully expand the user program before handing the expanded `##kernel` form to its CEK back-end. The new self-hosting front-end instead runs the expander entirely inside Pycket: it evaluates the bootstrapping linklets that implement the expander, uses them to expand the user program, and then evaluates the resulting core program—all on the CEK back-end from the outset. This architectural shift brings clear benefits, but it also introduces fresh costs.

Note that, Pycket remains a tracing JIT compiler whose primary objective is to identify and trace hot loops in *user* code. But with the new front-end, part of every “user program” is now the language program that expands the user code. After expansion, the fully expanded `##kernel` program is identical to what the Racket stand-alone binary would generate. Thus, the meta-traced CEK interpreter ultimately executes the same core program in both configurations—the difference lies only in where and how that program is obtained.

In the remainder of this chapter, we proceed in a similar order a user program travels through the system. We first cover Pycket’s baseline performance profile and review those RPython back-end optimizations most relevant to our study (§6.1). Then we examine how the runtime cooperates with Racket’s module system to improve the loading of core libraries (§6.2). Afterward, we analyze the cost of program expansion itself—a step now executed inside Pycket (§6.4). We also study the overall memory overhead induced by self-hosting (§6.5). Finally, we measure how self-hosting affects the performance of a *fully-expanded* user program (§6.3).

In Chapter 7, we propose concrete remedies for each challenge we expose and, supported by preliminary evidence, argues that these directions warrant deeper investigation as a general strategy for improving the performance of self-hosting on meta-tracing systems.

6.1 Pycket’s Performance Characteristics

With an extensive suite of micro- and macro-benchmarks plus larger real-world experiments, the existing Pycket implementation consistently demonstrates competitive performance when executing Racket code. It benefits from the generic optimizations provided by the RPython framework—including common-subexpression elimination, copy propagation, constant folding, loop-invariant code motion, malloc removal, and the inlining that naturally arises from tracing [24, 37, 38]. It also benefits from interpreter-specific improvements such as environment pruning, data-structure specialization, strategy objects, and more dynamic features like hidden classes. Nevertheless, earlier studies revealed workloads on which Pycket lags behind every other system—namely “almost exclusively recursive programs with data-dependent control flow in the style of an interpreter over an AST” [7, 8]. Investigating these cases launched the

present line of research.

Pycket’s baseline shows that when programs feature tight, well-behaved loops—numeric kernels, tail-recursive traversals, or higher-order combinators—the tracer quickly identifies the hot paths and specialization pays off. Across the Larceny and R6RS suites, for example, Pycket typically matches or exceeds Chez Scheme’s throughput while starting up in a fraction of the time required by ahead-of-time native compilers; inside a trace, most primitive operations collapse to a handful of low-level nodes and even polymorphic arithmetic becomes monomorphic [8].

The picture changes when control flow branches unpredictably or when large interpreter-like dispatch loops dominate execution. Here Pycket must juggle deep continuation structures, environment chains, and dynamically typed tag checks, all of which inflate trace size and lengthen warm-up as well. Garbage-collector telemetry (in Section 6.5) further shows that such workloads allocate an order of magnitude more temporary objects per unit time than steady-state numeric code, stressing the nursery and triggering full collections. These observations motivate the detailed analysis that follows.

To develop the core issue we first recall from Chapter 4 how Pycket currently detects loops. While for the low-level languages such as in a byte-code interpreter a program counter is used to detect loops, in Pycket the focus is on function applications, since Pycket works on program ASTs and the only AST that may create a loop is an application. Whether it is a basic counter or a compound state, the idea is that a loop is registered as soon as a program state transitions to one of the previous states. As reported in the previous studies, Pycket utilizes two techniques,

namely the two-state tracking and the use of a dynamic call-graph. In both techniques the idea is to eliminate the “false-loops” among all the observed trace headers (i.e. potential start of a loop). The two-state tracking encodes the trace headers as a pair of a lambda body and its call-site, and the call-graph method detects cycles on a call-graph that’s dynamically generated within the interpreter to handle extra levels of indirection. These approaches together are proven to be very effective in tracing code with a heavy use of shared control-flow indirections, such as the contract system [7, 8].

A second crucial component is the relevant RPython back-end optimizations. For example, Loop-invariant Code Motion (LICM) performed by the JIT [24]. LICM hoists interpreter-state destructuring into a preamble, leaving a compact peeled iteration that forms the loop body. If a side trace (a *bridge*) can jump directly into this peeled iteration, the runtime avoids re-executing hoisted operations and gains a substantial speed-up.

Such a jump, however, is legal only when the program state—heap-allocated environment frames, continuation objects, virtualized temporaries, and range variables—matches exactly the state expected at the end of the destination trace’s preamble. In Pycket a major part of the interpreter state consists of the environment and the continuation, which are all heap allocated objects. If the shape differs when entering a trace of a loop, for instance because a non-tail call inserted an extra frame to the continuation, the bridge must fall back to the preamble or to interpretation, negating any benefits from LICM, as well as tracing in general.

Earlier versions of Pycket mitigated some of this mismatch by allowing the JIT to allocate just enough additional data to reconcile the states. Escape analysis and malloc-removal often

virtualize those allocations [38, 24]. A small heuristic therefore permits the JIT to materialize objects that would otherwise stay virtual, trading a bit of space for a jump into the peeled iteration; thus yielding roughly a 4 % speed-up for gradually typed programs with no measurable overhead [8].

Any functionality that Pycket now imports as Racket code¹ would, in a handwritten RPython interpreter, utilize meta-interpreter hints—such as `@jit.unroll_safe`—to guide the JIT back-end. Because the imported modules arrive as opaque ASTs, Pycket cannot insert these hints, causing the tracer to encounter extensive interpreter-style dispatch loops without guidance. Additionally, the runtime feedback extensively used by Pycket and other interpreters built on RPython (as discussed in Chapter 2) is ineffective in this context. For instance, it is impossible to *promote* values within the macro expander or mark methods and functions as *elidable* without manually annotating the Racket code.

Introducing the linklet layer lets Pycket execute large Racket programs—including the 2642-function expander—entirely inside the JIT. Bootstrapping the *racket/base* language alone instantiates close to a hundred modules before user code starts. Although micro-benchmarks seem mostly unaffected for fully-expanded programs (the back-end is unchanged), these large workloads exacerbate all the aforementioned issues in program expansion. In short, the very extensibility that makes self-hosting attractive also deprives the meta-tracer of inside information, limiting trace quality and prolonging warm-up.

¹via bootstrapping linklets

6.2 Module and Language Loading

A Racket runtime has to load the language of the user program—most prominently the language declared on the `#lang` line—*before* it can even begin to expand the user’s source; the expander should be able to resolve every identifier that the program may reference, and doing so requires loading, and instantiating the language’s own modules so that all the bindings are available-for all the phases.

Consequently, the complete transitive closure of the language’s module graph has to be loaded and expanded ahead of the user program. For a language such as `#lang racket/base` that closure already contains roughly 90 leaf modules; larger languages may exceed that number. In the original, non-self-hosting Pycket this preparatory step was executed by the external racket binary, so its cost was invisible to the JIT. Now it is part of the measured runtime.

Table 6.1 quantifies that cost of a cold boot when *no* pre-compiled code is used, loading the `#lang racket/base`. The loading of the bootstrapping linklets, their compilation into linklet objects and their instantiation, as well as instantiating all the Racket modules for the `#lang racket/base` (*init-lib*) together dominate the 2.7s wall-clock time. Note that the compilation of a linklet involves constructing AST nodes for every object that was in the s-expression, and the A-normalization and assignment conversion passes. Moreover, because these steps are internal to the interpreter and not part of the user code, these milliseconds are almost entirely spent in the interpreter.

Phase	Time (ms)	GC (ms)	Calls
instantiate-linklet	868	33	7875
make-instance	39	5	13405
startup	156	53	1
expander-linklet	135	53	1
regex-linklet	37	8	1
thread-linklet	17	6	1
pycket-boot-linklet	0	0	1
fasl-linklet	2	0	1
set-params	18	0	1
compile	1603	298	–
compile-linklet	106	38	680
compile-sexp-to-ast	773	125	685
compile-normalize	211	27	2026
compile-assign-convert	512	108	5533
boot	2740	403	–
init-lib	128408	8477	–

Table 6.1: Loading #lang racket/base without using any compiled code

Before self-hosting, none of that overhead appeared: Pycket received a fully-expanded core program, and the JIT could concentrate on the hot loops of *user* code directly. The new architecture gives Pycket full visibility into the macro expander, the module system, the reader, and more, but that transparency comes with an up-front cost; evaluating all these sub-systems to load and expand then language on the JIT.

One obvious exploitation of this transparency is to reuse compiled code. By setting up some (Racket-level) parameters such as `use-compiled-file-paths` and `read-accept-compiled`, Pycket instructs the expander to prefer `.zo` files over `.rkt` sources whenever possible. Doing so eliminates the most expensive compile-time passes (cf. the disappearance of `compile-linklet`, `assign-convert`, and `normalize` in Table 6.2), yet it also shifts work: module bodies must

be deserialized ($\text{fasl} \rightarrow \text{s-exp}$) and validated, and the resulting syntax objects still have to be parsed into ASTs suitable for Pycket’s evaluator.

Shown in Table 6.2, as expected, using pre-compiled `.zo` artifacts for Racket modules makes the *compiled-code* path dramatically faster: the `init-lib` phase’s wall-clock time falls from about 127s when loading source to roughly 3.4s with compiled code, while avoiding compilation saves large heap allocations, therefore the GC time drops from 8.5s to a few hundred milliseconds.

Phase	Time (ms)	GC (ms)	Calls
instantiate-linklet	903	40	684
make-instance	2	0	845
startup	129	38	1
expander-linklet	109	38	1
regex-linklet	20	4	1
thread-linklet	12	4	1
pycket-boot-linklet	0	0	1
fasl-linklet	1	0	1
set-params	18	0	1
read	146	29	–
$\text{fasl} \rightarrow \text{s-exp}$	144	29	74
$\text{s-exp} \rightarrow \text{ast}$	2	0	74
assign-convert-deser.	0	0	–
compile	1439	313	–
compile-linklet	0	0	9
compile-sexp-to-ast	836	128	764
compile-normalize	0	0	28
compile-assign-convert	603	185	11763
boot	2673	428	–
init-lib	3387	361	–

Table 6.2: Loading `#lang racket/base` using compiled code

Regardless of the source of each module, every macro expansion of user code ultimately depends on the performance of evaluating the underlying expander. Pycket’s ability to inline into, and sometimes across, expander helpers opens new optimization opportunities, but it also means that the performance deficiencies in the evaluation of the expander itself now directly affect application performance. The next sections discuss those interactions in detail.

6.3 Cost of Self-Hosting

Everything related to the integration of bootstrapping linklets into Pycket’s runtime is handled at the front-end. The only modification in Pycket’s CEK back-end is the addition of linklet evaluation semantics, and numerous primitives. As a result, we do not anticipate significant differences in back-end performance between the original and the current Pycket implementations, since the CEK machine core remains essentially unchanged.

Improving the performance of the original Pycket is not one of the goals of this particular study. The Racket expander and all imported functionalities share the same characteristics that typically make dynamic languages challenging to implement efficiently, such as late binding, frequent dispatching, and the boxing and unboxing overhead from dynamic typing. Consequently, Pycket with self-hosting is not positioned as a competitor in performance against the original Pycket or even against the Racket runtime.

Since fully-expanded Racket programs are identical in both the original and the current Pycket implementations, we have no reason to expect the self-hosting Pycket to outperform the original. However, we hypothesize that introducing self-hosting via Racket does not negatively affect the JIT’s back-end capability to discover critical loops in the fully expanded user programs. In

this section, we empirically test this hypothesis, isolating the evaluation of the back-end and front-end performances, and analyzing the distinct costs incurred by self-hosting, specifically the overhead associated with including expansion time in the overall runtime.

6.3.1 Experiment Setup

To test our hypothesis, we conducted experiments on a 10-node Kubernetes cluster, with each worker node running Ubuntu 24.04 LTS on an x86-64-v2-aes CPU at 2.1 GHz, 32 MB cache, and 16 GB of RAM. The cluster was virtualized using Proxmox v8.2.2. Every benchmark ran isolated on a separate worker node in an independent virtual machine. Kubernetes' podAntiAffinity rules ensured each worker executed a single pod at a time, and each pod executed exactly one benchmark, reported the measured runtime, and terminated without interference.

All benchmarks were executed using Racket version 8.18.0.1 with Pycket at revision 00be1e3. We ran each benchmark uninterrupted for 100 iterations at the highest priority within an isolated process on the worker nodes. Execution time measurements were taken from within the benchmark program itself, excluding startup overhead. However, we did not separate warm-up phases, thus reported times incorporate JIT compilation. For reporting, we present the arithmetic mean of all runs along with bootstrapped confidence intervals at the 95% confidence level [39].

All Kubernetes job specifications and benchmark scripts were auto-generated for reproducibility. Scripts for generating jobs, analyzing results, and plotting data are available online at <https://github.com/cderici/pycket-performance/>.

6.3.2 Back-end

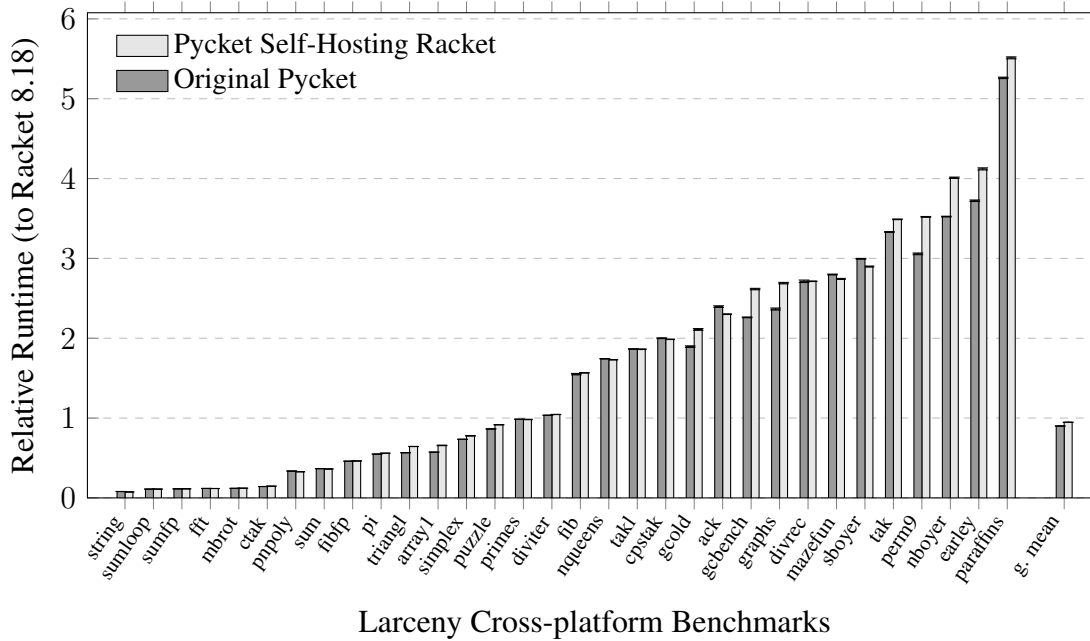


Figure 6.1: Fully-expanded program evaluation performance. Pycket with Self-hosting vs Original Pycket, relative to Racket 8.18 — lower is better.

The results comparing the original Pycket to the self-hosting Pycket using the Larceny Cross-platform Scheme Benchmark suite are summarized in Figure 6.1. We selected this benchmark suite because it was prepared and extensively used in earlier Pycket studies, with run time iterations adjusted to reduce jitter and all I/O operations moved outside the timed loop. The only modification introduced for this study was converting all benchmarks to use `#lang racket/base`, despite some benchmarks originally capable of using smaller languages, ensuring consistency across the benchmarks.

Note that because startup overhead is excluded from these measurements, the reported times do not include benchmark expansion, thus specifically isolating the measurement to back-end performance. However, the traces generated during the expansion of modules for `#lang racket/base` remain present in the JIT and are available during benchmark execution, although these traces from the expansion phase are not expected to be directly beneficial to the performance of the benchmarks themselves.

As demonstrated by the results—particularly when examining the geometric mean—both Pycket variants achieve very similar back-end performance, confirming our hypothesis that adding self-hosting does not significantly hinder the JIT’s ability to detect critical loops. However, in a few specific benchmarks, namely `earley`, `perm9`, and `nboyer`, Pycket with self-hosting shows a noticeable slowdown not observed in other benchmarks. We attribute this difference primarily to increased GC pressure: `perm9` is specifically designed to benchmark a system’s memory performance, `earley` involves complex grammar parsing, and `nboyer` is a logic programming benchmark—all of which frequently allocate and deallocate numerous heap objects during execution. Consequently, the memory-related challenges inherent to self-hosting become more prominent in these benchmarks. These memory issues are analyzed in detail in Section 6.5.

A closer inspection of the traces captured by the JIT further supports our hypothesis. For each benchmark, the most frequently used traces remain nearly identical between the original Pycket and the self-hosting variant, despite the latter including additional large traces generated during the expansion phase. Since our experimental results confirm that the introduction of self-hosting does not significantly affect Pycket’s back-end performance, the performance analyses from earlier studies on Pycket remain fully valid [7, 8].

6.3.3 Front-end

We now turn our attention to measuring the performance of the program expansion phase. During this phase, Pycket runs the macro expander provided by the expander linklet, transforming a given Racket program into a fully expanded `kernel` program. Unlike in Pycket’s original design, which utilized the standalone Racket binary to expand user programs externally, Pycket with self-hosting executes the `expand` function directly within its own runtime. For these measurements, we utilize the same experimental setup described previously in Section 6.3.1. Each benchmark previously used to evaluate back-end performance is transformed into a corresponding benchmark explicitly designed to evaluate the expansion performance of that same program.

```
#lang racket/base

(require "conf.rkt")

(define (ack m n)
  (cond ((= m 0) (+ n 1))
        ((= n 0) (ack (- m 1) 1))
        (else (ack (- m 1) (ack m (- n 1))))))

(for ([i (in-range outer)])
  (time (for ([j (in-range ack-iters)])
            (ack 3 9))))
```

(A) Back-end benchmark

```
#lang racket/base

(require racket/syntax "conf.rkt")

(define stx
  #'(module ack racket/base
    (define outer 100)
    (define (ack m n)
      (cond ((= m 0) (+ n 1))
            ((= n 0) (ack (- m 1) 1))
            (else (ack (- m 1) (ack m (- n 1))))))
    )))

(for ([i (in-range expand-outer)])
  (time (for ([j (in-range expand-inner)])
            (expand stx))))
```

(B) Front-end benchmark

Figure 6.2: Example benchmark used in measuring (A) Back-end and (B) Front-end performance.

Figure 6.2 demonstrates the nature of this transformation using the ack benchmark as an example. The back-end version of the benchmark repeatedly executes the fully-expanded program to measure runtime performance. In contrast, the transformed front-end version explicitly calls the `expand` function repeatedly on the same unexpanded syntax object, measuring the runtime performance of the expansion phase alone. Note that the time spent loading `#lang racket/base` remains excluded; only the runtime of the explicit calls to `expand` is measured. While these benchmarks may not perfectly represent typical expansion workloads, using the same set of programs for both back-end and front-end performance evaluations allows us to provide a consistent and comprehensive comparison of the overall costs associated with self-hosting in Pycket.

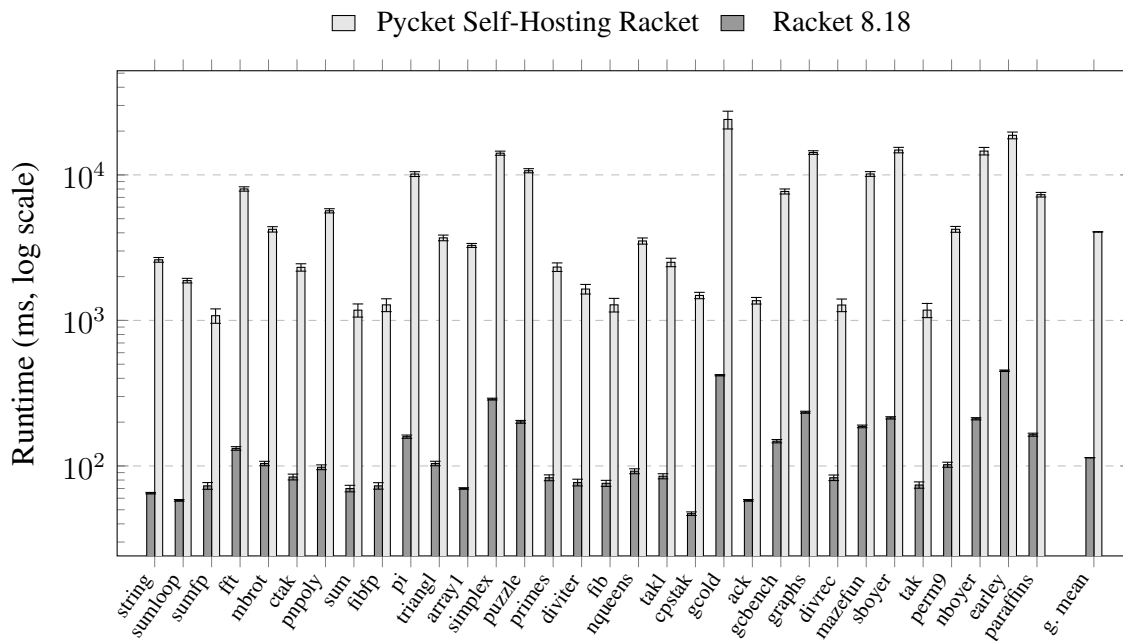


Figure 6.3: Program-expansion performance. Pycket with Self-hosting vs. Racket 8.18 — lower is better.

Figure 6.3 shows the performance comparison of the expansion phase between Pycket with self-hosting and Racket on Chez Scheme. Since the original Pycket delegates the expansion step to Racket’s standalone executable, directly measuring Racket’s expansion performance is equivalent to measuring the expansion performance of the original Pycket. The results indicate that the front-end performance of Pycket with self-hosting is approximately two orders of magnitude slower than that of Racket (and consequently the original Pycket) across all benchmarks. This significant difference aligns with earlier studies, which found that meta-tracing JIT compilers generally perform poorly on interpreter-style programs that have complex, data-dependent control flow [7, 3]. Since Racket’s expander inherently involves such interpreter-like computation, Pycket’s meta-tracer faces considerable challenges in generating efficient, reusable traces.

In the subsequent sections, we examine these issues more closely, providing targeted experiments and analyses to clearly identify the underlying problems. Specifically, Section 6.4 provides an in-depth investigation of why data-dependent, interpreter-style computations present fundamental challenges for meta-tracing JIT compilers. Additionally, Section 6.5 closely examines the memory-related issues, focusing particularly on the increased GC pressure that arises from self-hosting, and quantifies its impact on overall system performance.

6.4 The Nature of the Beast: Tracing Data-Dependent Computation

The *Racket macro expander*, loaded inside Pycket as the *expander linklet*, now runs in the same process as every user program; its performance therefore lies on the critical path, because (i) expansion itself contributes directly to end-to-end latency, as detailed in Chapter 4, and (ii)

the entire language hierarchy must be loaded and initialized first, as shown in the previous section.

Conceptually the expander is a tree-walking interpreter from higher level Racket languages to the `##kernel` language. Its control flow walks a rich lattice of macros and compile-time bindings, yielding deeply branch-heavy control flow whose exact shape depends on the user's program and the libraries it imports. Pycket's meta-tracing JIT therefore meets high-level Racket functions whose behavior cannot be trivially predicted at trace time; unlike the low-level CEK interpreter these helpers cannot be decorated with hints such as `@jit.unroll_safe`, and their extensive allocation and branching patterns frustrate the usual trace-specialization heuristics. Therefore, the traces generated for these high-level abstractions are substantially larger; the JIT spends significant time tracing and optimizing these paths only to encounter unpredictable branches, bail out, and fall back to interpretation—negating much of the anticipated speed-up.

In interpreter-style, data-dependent recursion the control flow fans out across many branches driven by the program's input, yet a tracing JIT captures only the *single* linear path taken while tracing and specializes the resulting code to that exact sequence of decisions. When new input steers execution down a different branch the JIT must abort, start another trace, and specialize again. Over time this produces a combinatorial explosion of narrow, redundant traces—each optimal for the data that birthed it but useless for most other paths—inflating compilation time, thrashing the trace cache, and frequently forcing the runtime to fall back on interpretation.

In Chapter 7 we propose a lightweight meta-hint mechanism that aims to help with this by helping to detect hot branches during interpretation and guiding the tracer away from branch-

heavy code paths, mitigating the trace compilation churn and the explosion of overspecialized traces. In the remainder of this section, we examine in detail how the meta-tracer in Pycket behaves on branch-heavy computations.

6.4.1 Tracing Branch-Heavy Computation

As detailed in Chapter 4, the Racket expander—containing the macro expander, module system, and more—is notably large and intricate. Investigating tracing JIT behavior directly within such a sizable and complex program can be impractical. Thus, we analyze the tracing behavior using smaller, controlled examples to isolate and clearly observe specific issues.

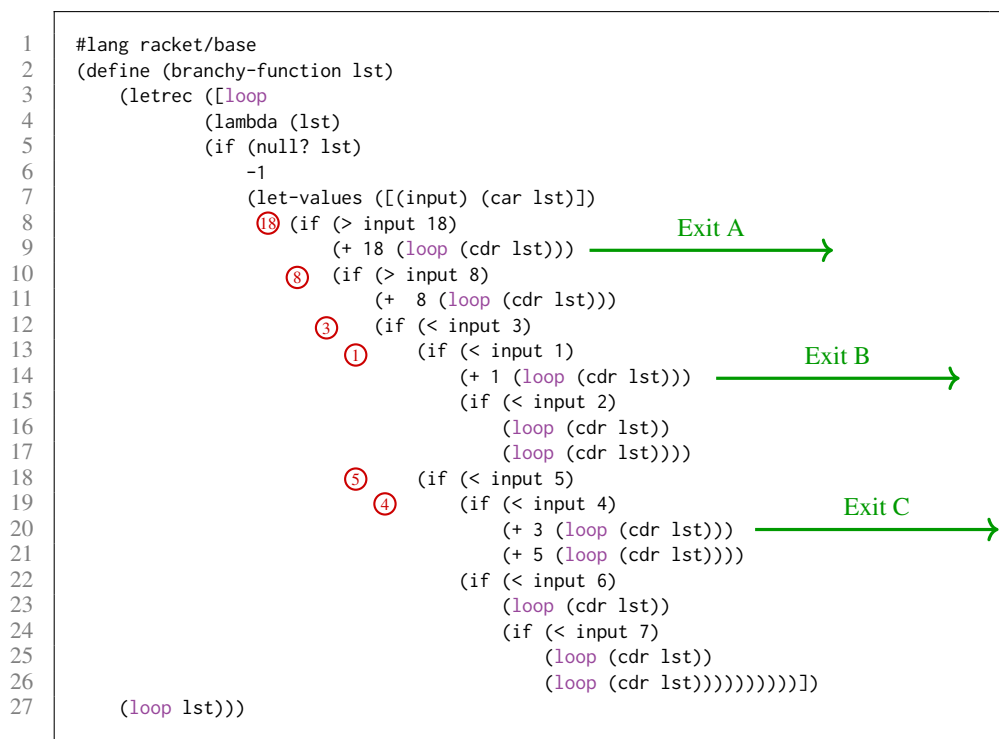


Figure 6.4: Branchy: a branch-heavy program designed to exhibit data-dependent behavior.

Figure 6.4 introduces *Branchy*², a synthetic benchmark intentionally designed to exhibit interpreter-style, branch-heavy, and data-dependent behavior. It is synthesized by simplifying Racket's `fasl→s-exp` function from the Fast-Load Serialization (FASL) library, which deserializes a given byte stream into an s-expression. Each iteration processes an element from an input list through nested conditional branches resembling a decision tree. Certain inputs produce consistently deep paths through this branching structure, while others immediately exit via shallow paths. For instance, an input list such as `'(0 0 0 ...)` repeatedly traverses the longer path sequence *18-8-3-1*, via Exit B, whereas `'(20 20 20 ...)` consistently follows the shorter path labeled *18*, repeatedly taking Exit A.

Note that describing static source code itself as "branch-heavy" can be misleading. The critical notion here is the data-dependent nature of branch decisions during runtime. A program could possess deeply nested conditional structures yet exhibit minimal runtime branching if the input consistently chooses shallow branches. Therefore, the "branchiness" of a computation is an empirical property, characterized by the frequency and complexity of branching decisions made during an actual evaluation over a particular input.

This data-dependency explicitly manifests through guards placed by a tracing JIT. As discussed in Chapter 2, guards serve as exit points in a trace, verifying assumptions made during tracing. Thus, each conditional encountered in the user program generates a corresponding guard in the trace to enforce assumptions about input data at trace time. An increased number of control-flow decisions therefore results in more guards within the trace. Additionally, guards

²Full code and alternative definitions of *Branchy* is available at <https://github.com/cderici/pycket-performance/blob/master/branchy/private/branchy.rkt>.

carry runtime information that allows the JIT to exit to the interpreter upon guard failure. Consequently, numerous guard operations can significantly increase memory consumption [40].

```

# Inner loop with 18 ops
label(p19, p24, p8, p21, p1, p22, p30)          54
i31 = getfield_gc_i(p24) ; FixnumCons._car      55
guard_not_invalidated()                        56

i33 = int_gt(i31, 18)                          57
guard_false(i33)                              58

i35 = int_gt(i31, 8)                          59
guard_false(i35)                              60

i37 = int_lt(i31, 3)                          61
guard_true(i37)                               62

i39 = int_lt(i31, 1)                          63
guard_true(i39)                               64

p40 = getfield_gc_r(p24) ; FixnumCons._cdr      65
guard_nonnull_class(p40)                      66
p42 = new_with_vtable()                       67
setfield_gc(p42, p8)                          68
setfield_gc(p42, p21)                         69
setfield_gc(p42, p1)                          70
setfield_gc(p42, ConstPtr(ptr43))             71
jump(p19, p40, p8, p30, p42, p22, p30)        72

```

Figure 6.5: RPython trace inner loop for Branchy running all-zero input taking a long 18-8-3-1 path.

Figure 6.5 presents the optimized inner loop of a trace recorded while running Branchy with an all-zero input, repeatedly taking the deep path sequence *18-8-3-1*, as shown earlier in Figure 6.4. Note that each guard operation within the trace corresponds directly to a conditional (if) in Branchy’s code. These guards ensure that runtime execution closely matches the assumptions made during tracing; if any guard fails, the trace prematurely exits, causing execution to either revert to interpretation or initiate another trace. Although this trace is optimized and compiled, it is overly specialized, limiting its applicability exclusively to identical or very similar inputs.

Thus, the internal structure of the input data significantly influences tracing outcomes. Repetitive input patterns, such as '(3 3 3 3 ...)', allow trace reuse within the pattern, thereby amortizing the overhead of tracing and compiling. However, highly repetitive inputs like these are uncommon in realistic workloads. Conversely, more varied or random inputs, such as '(1 3 5 1 3 5 ...)', significantly reduce trace reuse, because of the same specialization. This phenomenon of internal "loops" within the input data will be revisited in more detail in Section 6.4.2 when we discuss the nature of loops the tracer attempts to capture.

Table 6.3 illustrate this effect quantitatively. For inputs with highly repetitive branching decisions (e.g., all 20s), the meta-tracer records only two loops and three bridges, requiring a negligible fraction of runtime spent on trace compilation. However, even these traces remain highly specialized to the particular repetitive path taken, significantly limiting their reusability in more general cases where input variation occurs. In contrast, random inputs, more representative of realistic macro expander behavior, induce a combinatorial explosion: 39 loops, 248 bridges, and substantially more traced operations, resulting in significant overhead in trace compilation and optimization.

This observation naturally raises the question of what constitutes an ideal tracing strategy for interpreter-style, branch-heavy computations. Should a tracing JIT generate one highly specialized trace per unique branching pattern, incurring substantial memory and compilation overhead but achieving maximum runtime efficiency for those exact inputs? Or is it preferable to construct broader, less specialized traces containing multiple branches, potentially reducing memory footprint but introducing runtime overhead from unnecessary paths? Consider a repeated branching pattern such as *18-8-3-5-4-18-8-3-5-4-...*; capturing the entire pattern into

	Repeating Branches	Random Branches
Tracing (s)	0.004566	0.153118
Backend (s)	0.001239	0.044965
TOTAL (s)	1.185808	2.242329
ops	2130	147895
heapcached ops	572	41615
recorded ops	462	31202
calls	12	380
guards	128	7391
opt ops	243	11448
opt guards	75	3768
opt guards shared	46	2553
nvirtuals	74	1982
nvhols	8	126
nvreused	36	907
Total # loops	2	39
Total # bridges	3	248

Table 6.3: JIT back-end summaries for branch-heavy computations taking repeating vs random branches. Redacted fields are all zero for both cases.

one trace may yield high specialization, whereas identifying a shorter common prefix (e.g., *18-8-3-5*) might increase trace reuse at the cost of less specialization. However, for our specific scenario regarding self-hosting, neither tracing strategy sufficiently justifies the resources expended by meta-tracing during runtime. As we will elaborate in the next section, this inefficiency arises because program expansion occurs at compile-time, where the control flow lacks meaningful loops to capture—each computation path is effectively static and resembles the linear decision-making of a regular-expression matcher.

This fundamental tension contributes to the limited adoption of tracing JITs in the industry over the last decade. Highly specialized traces yield considerable speed-ups only for workloads

closely matching traced paths, while deviating workloads incur substantial tracing overhead without corresponding benefits. Consequently, the average-case performance may significantly degrade, exemplified by the observation that a tracing JIT that is ten times faster on half of a benchmark suite but ten times slower on the other half is, overall, more than five times slower in practice [41].

6.4.2 Performance of Program Expansion

Moving beyond synthetic examples, we now shift our attention to a practical, real-world use-case: regular expressions. By examining a realistic and widely used computational pattern, we deepen our understanding of the performance challenges meta-tracing encounters in more typical interpreter-style computations. Regular expression matching offers a valuable analog to program expansion, as it shares the interpreter-style, data-driven, branch-heavy computational nature inherent in real-world expansions. Through analyzing regular expressions, we can extrapolate critical insights into how Pycket’s meta-tracer handles such complex branch structures when processing large-scale expansions.

Fortunately, Pycket already includes a robust, efficient regular expression implementation in RPython, providing a direct benchmark. This allows us to precisely measure performance by comparing it against the identical functionality implemented in the regexp linklet, facilitating a clear, one-to-one performance analysis.

To illustrate concretely, we consider matching the simple regular expression `#rx"defg"` against a large string containing the substring `"aaa...defg...aaa"` positioned centrally, as shown in Figure 6.6.


```

1  #lang racket/base
2
3  (define l (make-string 4000 #\a))
4  (define r (make-string 4000 #\p))
5  (define str-big (string-append l "defg" r))
6
7  (define reg-r (regexp "defg"))
8  (define iter 1000000)
9
10 (time
11   (for ([i (in-range iter)])
12     (regexp-match reg-r str-big)))
13

```

Figure 6.6: A small regular expression matching program.

```

1  # Loop with 21 ops
2  # start of the trace (preamble)
3  label(i0, p1)
4  i3 = int_add(i0, 1)
5  p4 = getfield_gc_r(p1)
6  i5 = strgetitem(p4, i0)
7  i7 = int_eq(i5, 100)
8  guard_false(i7)
9  i8 = getfield_gc_i(p1)
10 i9 = int_lt(i3, i8)
11 guard_true(i9)
12 # peeled-iteration (inner loop)
13 label(i3, p1, p4, i8)
14 i11 = int_add(i3, 1)
15 i12 = strgetitem(p4, i3)
16 i14 = int_eq(i12, 100)
17 guard_false(i14)
18 i15 = int_lt(i11, i8)
19 guard_true(i15)
20 # jump
21 jump(i11, p1, p4, i8)
22

```

Figure 6.7: Trace of Pycket's regexp (in RPython) matching `#rx"defg"`.

As shown in Figure 6.7, the critical operation is essentially a linear literal search through the string. The RPython implementation successfully identifies and optimizes this pattern into

a tight, efficient loop trace, highlighting its capability to leverage loop invariants and avoid unnecessary allocations. Figure 6.8 shows the trace for most used loop in Pycket using the regexp linklet implementation.

The most notable difference is the size of the trace, with 191 operations vs 21, and significantly many parameters in the optimized loop demonstrates the generic and loose nature of the trace, as opposed to the tight loop that the RPython implementation was able to capture. Furthermore, there are numerous arguments to the trace, therefore, many objects that existed before the trace and that are passed in as arguments, therefore the partial evaluating optimizer performing escape analysis just copies all the operations that use them into the output trace.

The most notable difference is the trace size, with 191 operations compared to 21. The significantly larger number of parameters in the optimized loop highlights the generic and loosely structured nature of the trace, in contrast to the tight loop effectively captured by the RPython implementation. Moreover, the large number of parameters also represent objects that were live prior to tracing and are passed into it. Consequently, the partial evaluator's escape analysis duplicates all operations involving these arguments directly into the resulting trace, which increases size dramatically.

The comparative results, shown in Figure 6.9, highlight significant performance disparities: the handwritten RPython implementation on Pycket achieves roughly a 30-fold improvement over standard Racket and approximately a 50-fold improvement compared to Pycket using the regexp linklet. This substantial advantage arises because the RPython implementation explicitly leverages interpreter hints, type specialization to optimize allocation strategies, and domain-

```

# Loop with 191 ops
# start of the trace (preamble)
label(p0, p1)
guard_class(p0)
p3 = getfield_gc_r(p0) ; ConsEnvSize1.vals
guard_nonnull(p3)
i5 = instance_ptr_eq(p3, ConstPtr(ptr4))
guard_true(i5)
guard_class(p1)
p7 = getfield_gc_r(p1) ; LetCont.ast
guard_value(p7, ConstPtr(ptr8))
p9 = getfield_gc_r(p1) ; Cont.env
guard_not_invalidated()
p10 = getfield_gc_r(p1) ; Cont.prev
guard_class(p9)
p12 = getfield_gc_r(p9) ; ConsEnv.prev
guard_class(p12)
p14 = getfield_gc_r(p12) ; W_Closure1AsEnv.caselam
guard_value(p14, ConstPtr(ptr15))
i16 = getfield_gc_i(p9) ; ConsEnvSize1Fixed.
i18 = int_add_ovf(i16, 1)
guard_no_overflow()

# ... ~120 guards / field loads elided
# ... checks on start-range, lazy-bytes, regexp state, etc.

# peeled-iteration (inner loop)
label(p20, p12, i18, p10, p20, p29, p45, p46, i48, p60, p67, p68, i70, p61, i73, p72, p25, i82, p81, p43)
guard_not_invalidated()
i89 = int_add_ovf(i18, 1)
guard_no_overflow()
guard_nonnull_class(p20, ForwardLink) ; Continuation mark walk

# ... more guards elided

i92 = int_sub_ovf(i89, i70) ; compute new byte offset
guard_no_overflow()
i94 = int_lt(i92, 0) ; bounds check (low)
guard_false(i94)
i95 = int_ge(i92, i73) ; bounds check (high)
guard_false(i95)
i96 = getarrayitem_gc_i(p72, i92) ; in_3[pos]
i97 = int_ge(i96, i82) ; second-vector bound
guard_false(i97)
i98 = getarrayitem_gc_i(p81, i96) ; indexing into mutable-byte vectors
i100 = int_eq(1, i98) ; (eq? 1 byte)
guard_false(i100)

# more "not-AppRand784" range checks

i101 = arraylen_gc(p43)
jump(p20, p12, i89, p10, p20, p29, p45, p46, i48, p60, p67, p68, i70, p61, i73, p72, p25, i82, p81, p43)

```

Figure 6.8: Trace of Pycket using the regexp linklet, matching #rx"defg".

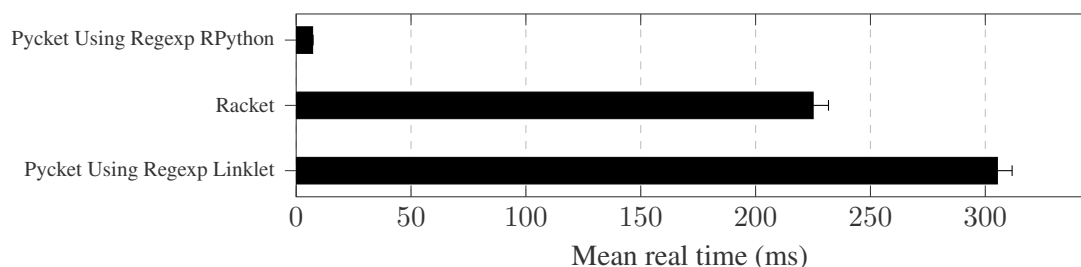


Figure 6.9: Comparison of matching `#rx"defg"` against a large `"aaa...defg...aaa"` string — lower is better.

specific optimization techniques like caching and contexts. Collectively, these optimizations allow the RPython implementation to efficiently manage control flow and memory usage.

The advantages inherent in the RPython implementation are further highlighted by analyzing performance *without* JIT compilation. Turning off the JIT for both versions reveals insightful patterns: the RPython implementation slows down approximately six-fold, whereas Pycket using the regexp linklet suffers two-orders of magnitude performance degradation. This difference emphasizes how JIT compilation substantially benefits the evaluation of Racket code. However, the absolute fastest runtime remains with the RPython implementation, reflecting its superior capacity to capture the computation’s essence and avoid pitfalls such as mispredicted branches and overspecialized trace bailouts. This issue is evident in the extensive, allocation-heavy traces generated for the Racket-based implementation (as seen in Figure 6.8), significantly increasing garbage collection overhead, an issue explored further in the following section.

Data dependency is again prominent in our analysis. The trace behavior specifically observed for the input `#rx"defg"` differs entirely when matching a pattern like `#rx"a*`", underscoring the inherent variability in program paths that meta-tracing attempts to capture. Extending this

observation to the broader context of Racket’s expander, we anticipate a similar scenario but exacerbated by complexity: a diverse stream of varied cases bombarding the tracer, which attempts yet fails to efficiently reuse specialized traces due to the rapidly shifting data-dependent execution paths.

Recall that PyPy addresses tracing challenges by identifying and tracing loops directly in the user program rather than at the interpreter level. Essentially, PyPy elevates the abstraction level at which the tracing takes place [4].

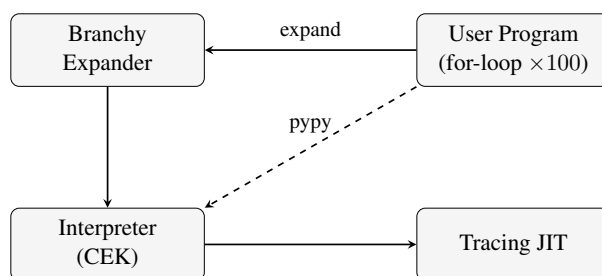


Figure 6.10: A new abstraction level brought by self-hosting.

Figure 6.10 shows how Pycket, in turn, pulls that abstraction level back to where it was, by essentially turning the user program into the interpreter that meta-tracing was able to bypass. With this design, the meta-tracing interpreter evaluates the Racket macro expander that expands the user program, thus, seeking hot loops within a higher abstraction layer—where loops are fundamentally complicated, and typical programs become mere streams of heterogeneous inputs, inheriting all the problems that come with tracing branch-heavy code as we detailed in Section 6.4.1.

Transforming or expanding a program shares fundamental characteristics with regular expression matching or FASL parsing—each being equally data-dependent. The essence of these computations lies primarily in the ordered manipulation and parsing of symbolic structures rather than in executing the semantics within those structures. Importantly, expanding or transforming a user program that contains loops represents a fundamentally different computational pattern to a tracing JIT than simply interpreting those loops. When interpreting loops, the meta-tracing JIT can trace iterations and capture hot paths. In contrast, program expansion involves repeatedly branching and manipulating data structures, rarely exhibiting consistent iteration patterns suitable for effective tracing. Nevertheless, the loops within a fully expanded user program remain efficiently capturable by Pycket’s meta-traced CEK machine core. We will discuss the implications of this multi-level tracing scenario and its resulting performance characteristics further in Section 6.3.

Addressing this challenge requires further research and careful consideration. At the heart of this challenge is the fundamental violation of the second assumption underlying tracing JITs, as previously discussed in Chapter 2. The second assumption—that multiple iterations of the same loop are likely to traverse similar code paths—is directly contradicted by the highly data-dependent, branch-heavy nature of interpreter-style computations. This inherent violation explains why such computations, although critical for self-hosting, remain particularly ill-suited to meta-tracing and tracing JITs in general.

6.5 Memory Pressure under Self-hosting

Another significant problem we identified in self-hosting functional languages on meta-tracing JITs, such as Pycket, is increased pressure on the memory, and garbage collection. In the original Pycket design, module expansion was delegated to Racket’s standalone executable, leaving Pycket’s heap usage predominantly limited to user program evaluation. However, the self-hosting design now requires Pycket to execute large-scale expansions internally—particularly for modules employing extensive macros or sophisticated language features. As described in Chapter 4, this change substantially increases heap allocations, notably in the form of deeply nested continuation objects and large, persistent structures, such as linklet instances containing extensive sets of functions. Consequently, Pycket’s heap exhibits characteristics significantly different from typical interpreter workloads, placing considerable stress on the GC, which relies heavily on allocation behavior assumptions for its efficiency.

To better understand the impact of these heap allocation patterns, we briefly review the garbage collector employed by the RPython framework—namely, the minimark GC. As discussed in Chapter 2, RPython translates interpreter-level memory operations (e.g., `malloc`) into managed allocations using its built-in garbage collector. The minimark GC is a generational, semispace-copying collector optimized for short-lived objects by employing a nursery sized roughly to match the CPU’s Level 2 cache. Objects are initially allocated in the nursery; upon nursery saturation, surviving objects are copied into an older generation to reduce full collections. The underlying assumption is that most objects die young, a property which typically holds for workloads encountered by tracing interpreters, as young, short-lived temporaries dominate their allocations. However, in the context of self-hosting expansion, Pycket gen-

erates numerous large, long-lived objects—such as continuation chains and complex linklet instances—which consistently survive minor collections and must be managed in older generations, triggering expensive full collections and undermining the generational assumptions of minimark GC. [12, 3, 42, 43]

The integration of Racket’s expander into Pycket’s runtime alters heap allocation patterns significantly, causing a substantial increase in old-generation heap objects and long-lived continuation chains. Consider, for example, the instantiation of the expander linklet itself, which consists of over 2600 functions and persists throughout Pycket’s runtime. Similarly, macro expansions and module loading involve deeply nested continuations due to extensive control-flow indirections common in real-world Racket programs. Within RPython’s minimark GC, objects identified as large or old³ are moved to an external generation, where they remain stationary and are collected through a serial mark-and-sweep phase. As a result, not only does Pycket fail to fully benefit from the nursery’s fast collection cycles, but it also incurs additional overhead from the blocking mark-and-sweep collections of these long-lived external objects.

	Collections	Bytes copied (KiB)	Old-gen growth (MiB)	GC wall-time (ms)
Original Pycket	15	1367.1	19.2	40.6
New Pycket	500	546.4	148.0	890.6

Table 6.4: Memory footprint of original Pycket vs Pycket hosting Racket. Nursery size: 32M

The practical effects of these memory-management changes on Pycket are quantitatively evident in Table 6.4, which contrasts the original Pycket with the current implementation

³The age of an object is the number of collections it survived.

integrating Racket’s self-hosting architecture. All values presented in the table are averages computed over 1000 runs. Switching to the self-hosting design dramatically increases the frequency of minor garbage collections—from an average of 15 collections in the original Pycket to 500 collections in the current implementation. Although each minor collection in the current Pycket typically copies fewer bytes (approximately 546 KiB versus 1367 KiB), the cumulative impact is substantial. Specifically, the growth of the old-generation heap rises nearly eight-fold, from 19.2 MiB to 148.0 MiB. Consequently, total garbage-collection time increases significantly, from approximately 41 milliseconds in the original Pycket to about 891 milliseconds in the current Pycket, underscoring that the introduction of large, persistent, and pointer-rich Racket structures fundamentally challenges the assumptions underlying RPython’s generational minimark GC.

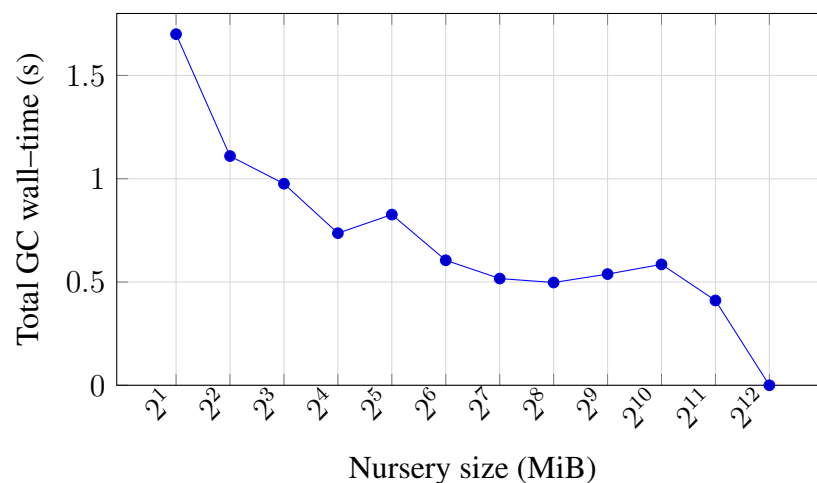


Figure 6.11: Effect of nursery size on total GC wall-time for Pycket with self-hosting.

The impact of nursery size on garbage-collection overhead in Pycket is demonstrated in Figure 6.11, highlighting clear trade-offs in tuning this parameter. At smaller nursery sizes

(2–8 MiB), frequent minor collections lead to substantial total GC overhead, reaching up to 1.7 seconds. Increasing the nursery size to align with typical CPU cache sizes (32–64 MiB) significantly reduces this overhead, with GC times stabilizing around 0.5 seconds, thus identifying an optimal performance region. However, further enlarging the nursery provides diminishing returns, only slightly reducing the overhead while increasing memory demands. These results confirm that while nursery-size tuning can mitigate the pressure induced by large and persistent objects, it cannot completely address the underlying challenges posed by Racket’s expansion workloads in Pycket.

7. APPROACHES TO IMPROVE SELF-HOSTING PERFORMANCE

CHAPTER SYNOPSIS

We have identified solution approaches that are worthy of further investigation for improving self-hosting performance.

You either try to avoid branch-heavy code, or you try to get better at it.

Sections:

- 7.1 Guiding Tracer Away from Branch-Heavy Computation

Present approaches for detecting branch-heavy computations and helping tracer avoid them.

- 7.2 Mitigating Memory Pressure in Branch-Heavy Computation

A hybrid model of computation (e.g., a stackful model alongside CEK) seems to mitigate memory issues in some cases.

We demonstrated extensively in Chapter 6, that the predominant performance degradation on Pycket self-hosting Racket arises from intricate branching patterns during macro expansion and associated runtime computations. This degradation significantly inflates both trace sizes and the GC overhead. In other words, branch-heavy computation and memory problems appear central to the performance challenges encountered when self-hosting on a meta-tracing JIT compiler.

Addressing these core issues has the potential to transform self-hosting from a theoretical convenience into a practical and robust methodology for building language runtimes. By reducing the impact of branch-heavy computations and controlling memory usage, we can achieve consistent performance benefits from meta-tracing JIT compilers.

The fundamental nature of self-hosting entails extensive branching, especially during the macro expansion phase. In Racket’s case, macro expansion inherently involves deeply nested pattern-matching and conditional branching [5]. Similarly, the limitations of meta-tracing JIT compilers like RPython when handling interpreters with complex control-flow patterns—such as extensive branching and recursion—are extensively documented in Bolz’s dissertation [3]. Additionally, Bolz et al.’s work further emphasizes how intricate branching within interpreters can lead to trace explosion and limited reuse [4]. More generally, literature on compiler bootstrapping and self-hosting underscores that these processes inherently involve extensive branching and recursive computations, making simplification intrinsically challenging [44]. Consequently, while simplifying these computations might theoretically seem feasible, practical evidence and prior research consistently indicate intrinsic limitations in reducing branch complexity within self-hosting environments.

Partial evaluation (PE) is a commonly used optimization technique in tracing JIT compilers. Being a program transformation (and specialization) operation, given a program and some *static* variables (e.g. inputs or annotations), PE attempts to constant-fold away every operation that it can infer to be constants. The rest of the program (the operations that it can't fold) are outputted as the –hopefully– more optimized and specialized version of the original program. [45] uses PE to collapse interpreter dispatch, and [46] reports good results across JavaScript, Ruby, and R. [47] applies a closely related form of runtime specialization, while [48] treats both tracing and partial evaluation as general meta-compilation techniques.

Despite these successes, applying PE to Pycket's embedded Racket macro expander would be detrimental, particularly due to the branch-heavy nature of macro expansion. In Pycket, the Racket macro expander is loaded as ordinary Racket code through the *expander linklet* and evaluated at boot time. Partially evaluating this code would transform a single, general-purpose expander into many *program-specific* expanders. Given the macro expander's reliance on deeply nested, pattern-matching conditionals, the specialized traces would inline substantial amounts of branching logic. This process in a meta-tracing system results in trace explosion, diminished reuse across different programs, and frequent trace invalidation once trace-size thresholds are reached—behavior already documented extensively in PyPy's meta-tracer.

Furthermore, since macro expansion typically occurs infrequently (usually once per module load), any upfront overhead incurred by PE would not be amortized by repeated execution, diminishing its potential performance benefits. Given these factors, we opted not to implement PE in our current setup. Implementing such specialization within the RPython framework would entail development efforts comparable to building Pycket itself from scratch, rendering it

impractical as an exploratory optimization.

In this chapter, we explore targeted strategies to enhance performance specifically for branch-heavy computations on meta-tracing JIT compilers and propose viable directions for addressing these central performance challenges. First, we discuss techniques to guide the compiler away from generating extensive, non-reusable traces that degrade performance. Second, we propose methods aimed explicitly at mitigating memory pressure—a critical bottleneck in self-hosting environments. While recognizing the preliminary nature of our current evidence, as discussed in detail in Chapter 6, we nonetheless demonstrate the potential efficacy of these approaches, providing motivation for further research and investigation.

7.1 Guiding Tracer Away from Branch-Heavy Computation

A critical observation from Chapter 6 is that performance overhead in self-hosting Racket on meta-tracing JIT compilers primarily comes from generating large and overly specialized traces during the macro expansion phase. Such expansive traces degrade runtime performance due to the frequent and intricate branching patterns inherent in macro expanders. These traces are rarely reusable, as they encode extensive branching decisions tied closely to specific input cases, thus compounding the inefficiency.

Importantly, analyses in Chapter 6 demonstrate that once macro expansion concludes, the performance of self-hosted Pycket closely aligns with that of the original non-self-hosted implementation. This clearly indicates that the macro expansion phase, dominated by branching complexity, constitutes the primary bottleneck to overall runtime efficiency. Therefore, any improvement targeted specifically at managing branch-heavy computations during macro

expansion is likely to yield significant performance benefits.

Thus, guiding the tracer away from branch-heavy computations that yield large, non-reusable traces represents a targeted strategy to alleviate this bottleneck. Rather than eliminating branching entirely—an unrealistic goal given the nature of macro expansion—the idea is to minimize or selectively avoid tracing computations involving deep or overly complicated branching. This selective tracing strategy can prevent generating costly traces, improving the runtime efficiency during the most performance-critical phases.

To investigate the effectiveness of this targeted approach, we experimented with specific strategies to guide the tracer away from branch-heavy computations. We will describe each strategy clearly and provide evidence regarding their efficacy. Although, as we will see, none of these strategies completely resolves the performance issues, our experiments confirm that the proposed techniques do indeed provide measurable improvements in specific cases, demonstrating that they are worthy of further investigation, while also highlighting the substantial nature of the underlying challenge. Consequently, this challenge warrants extensive additional research, potentially constituting a separate research effort in itself.

Recall from Chapter 2 that a language interpreter written in RPython leverages *JitDriver* reflection provided by the RPython framework to define a custom Program Counter (PC). Interpreter hints such as `jit_merge_point` and `can_enter_jit` are used to indicate loop headers and points where the interpreter might jump backwards, respectively. Each invocation of `can_enter_jit` increments a profiling counter associated with the green variables, as illustrated previously in Figure 2.2 in Section 2.1. Pycket’s two-state tracking defines its PC (the green

variables) as a pair consisting of an AST (lambda) and its corresponding call-site (application), which helps determine the loop headers accurately. Additionally, each lambda AST node in Pycket has methods `enable_jitting()` and `disable_jitting()` that indirectly control whether `can_enter_jit` is invoked during execution of the CEK loop.

Unfortunately, RPython currently lacks a dynamic mechanism for instructing the tracer to abort an ongoing trace explicitly at the interpreter level (i.e., there is no option to declare “this path is not worth tracing, abort tracing now”). As a result, strategies involving the dynamic detection of deeply nested branching to abort the tracing of a lambda function mid-trace are unavailable. This limitation constrains our approach to guiding the tracer away from problematic branch-heavy paths, necessitating alternative strategies that work either statically or via explicit annotations in user code.

Statically disabling JIT for branch-heavy lambdas A straightforward approach to avoid tracing excessively branch-heavy paths is to detect such patterns statically and prevent them from entering the JIT altogether. Recall from Chapter 2 that Pycket performs whole-code analyses, including *a-normalization* and *assign-convert*, which rebuild the AST from the inside out. During these transformations, we mark any lambda with deeply nested conditional branches as *jit-blocked*, effectively setting its `should_enter` field to be always `False`, thus ensuring that `can_enter_jit` is never invoked for that lambda. However, as discussed in Chapter 6, branching complexity is not purely a static property but depends on runtime input patterns. Consequently, this static approach can inadvertently prevent tracing of functions that, while containing many nested conditionals, also possess frequently executed shallow branches. Thus, the overall performance can sometimes degrade due to a lack of JIT optimization on paths that would

benefit from it.

Using meta-hints to guide the tracer An alternative and more dynamic approach involves the introduction of *meta-hints*, a generalization of interpreter hints, which expands the communication between the interpreter and the tracing JIT at the user-code level. We introduce two new Racket forms: `define/jit-merge-point` and `meta-can-enter-jit`. The form `define/jit-merge-point` creates lambdas that are *jit-blocked* by default (with their `should_enter` set to `False`). Tracing is enabled dynamically only when a `meta-can-enter-jit` annotation is encountered in the Racket code, flipping the switch to invoke `can_enter_jit` explicitly. By carefully placing `meta-can-enter-jit` annotations at shallow branches within loops, we ensure that profiling counters associated with the loop header and the call site (green variables) increment only when shallow, less branch-intensive paths are repeatedly executed. In other words, deep branches within a loop do not contribute to the hotness of that loop initially; only the shallow ones increase the hotness. This selective approach encourages tracing and compilation of loops that are more likely to produce reusable and efficient traces, potentially reducing the overhead from deeply branched computations.

Figure 7.1 shows the Branchy example from Figure 6.4, now annotated explicitly with the introduced meta-hints. In this annotated version, only the shallow loop paths—for example the branch leading to Exit A (marked as ⑮)—are labeled with `meta-can-enter-jit`. Consequently, during execution, taking deeper branches (such as those leading to Exit C at ④ or beyond) does not increment profiling counters, thus not contributing to the perceived hotness of the loop. However, it is important to note that while this technique helps ensure that only loops frequently taking shallow branches trigger tracing, it does not entirely prevent deeper branches from being

```

1  #lang racket/base
2  (define/jit-merge-point (branchy-function lst)
3    (letrec ([loop
4              (lambda (lst)
5                (if (null? lst)
6                    -1
7                    (let-values ([([input] (car lst))]
8                      (if (> input 18)
9                        (meta-can-enter-jit (+ 18 (loop (cdr lst))))
10                       (if (> input 8)
11                         (meta-can-enter-jit (+ 8 (loop (cdr lst))))
12                         (if (< input 3)
13                           (if (< input 1)
14                             (+ 1 (loop (cdr lst)))
15                             (if (< input 2)
16                                 (loop (cdr lst))
17                                 (loop (cdr lst))))))
18                         (if (< input 5)
19                           (if (< input 4)
20                             (+ 3 (loop (cdr lst)))
21                             (+ 5 (loop (cdr lst))))
22                           (if (< input 6)
23                               (loop (cdr lst))
24                               (if (< input 7)
25                                   (loop (cdr lst))
26                                   (loop (cdr lst))))))))))
27    (loop lst)))

```

Diagram annotations: Red circles with numbers 18, 8, 3, 1, 5, 4 are placed next to the corresponding `if` expressions. Green arrows labeled "Exit A", "Exit B", and "Exit C" point to the right from the `(meta-can-enter-jit ...)` expressions at lines 9, 14, and 20 respectively.

Figure 7.1: Branchy; from Figure 6.4 annotated with meta-hints.

included in traces once tracing has commenced. In other words, once a shallow path has become sufficiently hot to initiate tracing, subsequent deep branches executed during tracing will also be recorded, potentially causing traces to become larger and less reusable again.

A notable downside of the meta-hints approach is the requirement for explicit annotations in the interpreted Racket code (e.g. the macro expander). While feasible in small-scale, isolated programs such as Branchy, annotating complex and larger-scale programs—such as the macro expander used in Racket’s self-hosting setup—can quickly become impractical due to their complexity and the extensive indirections they often employ. Additionally, explicit annotation compromises modularity by making exported functionalities cumbersome to use directly, as they

require meta-hint integration to maintain optimal performance. Thus, while this approach has potential benefits in targeted scenarios, the practical overhead of annotation presents significant limitations for general use in large-scale scenarios.

Add nested if depth to green variables Recall from Pycket’s two state representation we have the `lambda AST` and `come_from` information to detect loops (the green variables). Each time the JIT crosses a `can_enter_jit` with the same green variables, it increments a counter associated with those exact green variables (i.e. the compound program counter). In this approach, we add a third variable to the green variables that represents a “depth” score for nested conditionals. For instance in Branchy, Exit A would have a depth score of 1, and Exit B would have a depth score of 5. These scores are assigned to the application nodes during the A-normalization automatically. The idea is that every time we cross those looping points, if the depth score is more than a certain number, we add that number to the depth. So for example, if we have a threshold of 3, next time we take the Exit B, the depth variable would become 10 (because $5 > 3$), but at Exit A the depth stays the same (because $1 < 3$). This way, the green variables that `can_enter_jit` method increases the counter for will be different each time we loop at a deep branch. This way, the deep branches won’t contribute to the hotness of the loop, while the shallow branches will consistently increment the same counter.

Figure 7.2 shows the runtime results of Branchy for three input scenarios: an "only shallow" path (input always taking the shallow Exit A), an "only deep" path (input always taking the Exit B), and randomized inputs. We compare vanilla Pycket with the two proposed approaches: meta-hints and augmented green variables with depth information. Each measurement is averaged over 1000 runs with input lists containing 2000 numbers each, with bootstrapped

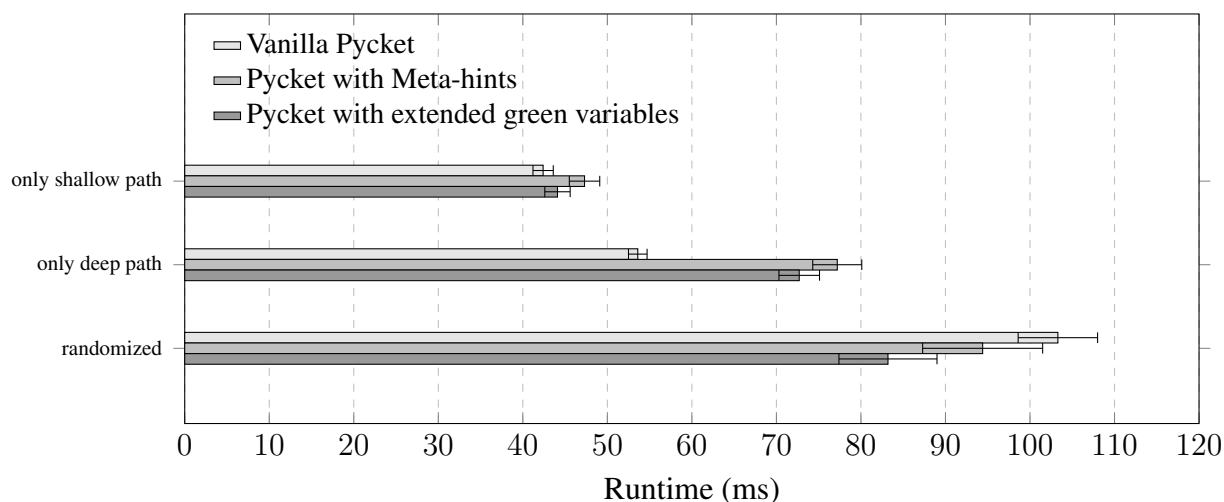


Figure 7.2: Experiment results for targeted improvement approaches running Branchy. Lower is better.

confidence intervals at the 95% confidence level [39]. All run times are averages over 1000 runs with the same setup explained in Section 6.3.1. The random inputs are generated freshly at each iteration and kept the same across the interpreters at each run.

For the shallow path scenario, all approaches perform similarly to vanilla Pycket, as expected, since shallow branches are traced by all three interpreters. For the deep path scenario, both meta-hints and the augmented green variables approaches successfully avoid generating overly specialized, long traces. However, this does not translate directly into better overall performance; instead, runtime performance worsens because the deeply nested loops remain interpreted rather than JIT-compiled. In other words, we do eliminate time for recording, compiling, and optimizing, but if these deep branchy paths are not JITted, then it gets much worse as they are frequently taken. With randomized inputs, however, we observe a beneficial trade-off: the optimized versions avoid some overhead from large, unnecessary trace compilations. Nonetheless, performance here remains highly input-dependent; once tracing starts due to

sufficient shallow branch hits, subsequent deep branch executions still accumulate into the recorded trace, reducing potential gains. These results confirm our hypothesis that strategically avoiding tracer entry into deeply nested branches can indeed yield performance improvements, while highlighting the complexity and limitations of applying these strategies more broadly.

Now we turn to an experiment that's closer to a real-world scenario to better evaluate the practical applicability of the techniques we introduced. Recall from Section 6.4.2 that Pycket has a manually written regular expression matcher implementation in RPython, as well as an equivalent Racket-based version provided through a utility linklet bundled in the bootstrapping linklets, which we call the *regexp linklet*. Previously, we leveraged this dual implementation to highlight performance issues during program expansion; results comparing these two implementations are presented in Figure 6.9. To further assess the techniques discussed earlier in this section, we devised a set of targeted experiments using the regexp linklet implementation under scenarios analogous to those used in our earlier Branchy experiments. Specifically, we evaluated the following interpreter variants:

- Pycket that uses the regexp linklet without any optimizations, establishing a baseline performance.
- Pycket that uses the regexp linklet with the meta-hints optimization. For this version, we manually annotated the Racket implementation of the regexp matcher with `define/jit-merge-point` and `meta-can-enter-jit`, specifically targeting shallow branching loops to encourage efficient trace generation.
- Pycket that uses the regexp linklet with extended green variables incorporating nested conditional depth information, as described earlier in this section, to dynamically discourage

the tracing of deeply nested branching paths.

Input scenarios:

- A. A regexp match scenario that corresponds to a shallow branching loop, identical to the experiment presented previously in Section 6.4.2. In this scenario, the matcher attempts to match `#rx"defg"` against a large input string structured as `"aaa...defg...aaa"`.
- B. A regexp match scenario designed specifically to create a deeper, heavily branched loop. In this scenario, the regexp matcher attempts to match the longer pattern `#rx"abcdefghijklmnopqrstuvwxy(*1000)...abcdefghijklmnopqrstuvwxy"` against an input string with the form `"abcdefghijklmnopqrstuvwxy(*1000)...abcdefghijklmnopqrstuvwxy"`. Notice that the prefix repeatedly excludes the final character "z", forcing deep branching during matching until the pattern is finally matched at the very end.
- C. A complex randomized input scenario designed to explore diverse internal branching paths of the regexp matcher. Here, the matcher attempts to match the complex alternation pattern `#rx"(?:defg|d[aeiou]{0,3}fg|d(?:x|y|z)*fg|d.*?fg)"` against large strings randomly generated to include various matching alternatives explicitly present in the pattern. The generated randomized input sequences remain consistent across different interpreters at each run.

Figure 7.3 shows the results of the experiments described above. All runtime values represent averages over 1000 runs. In experiment (A)—the shallow branching loop—the captured traces were identical across all interpreter configurations. The meta-hints approach exhibits

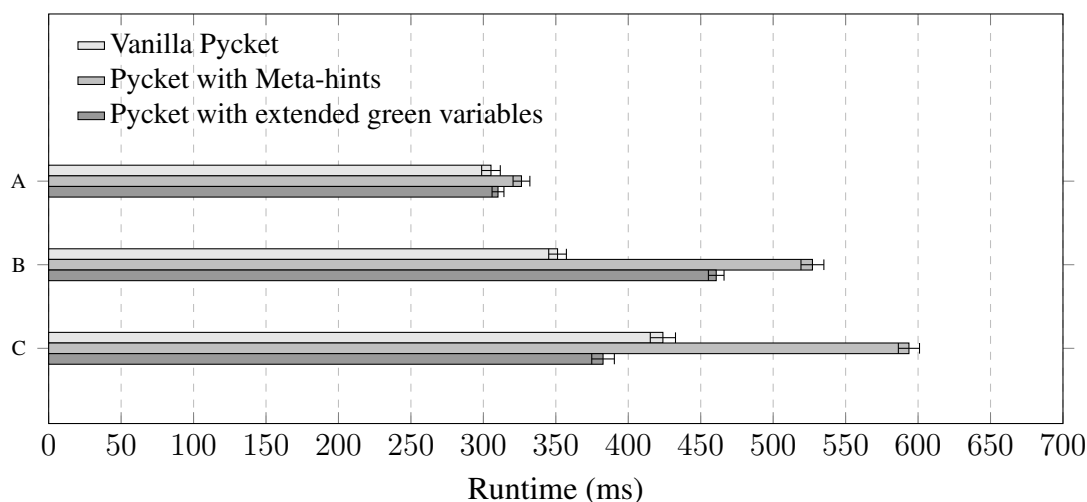


Figure 7.3: Experiment results of regular expression matching with targeted improvement approaches. All interpreters use the regexp linklet. Lower is better.

a slight slowdown due to its own overhead, however both optimizations perform comparably to vanilla Pycket using the regexp linklet, as anticipated. In experiment (B)—the deeper, repeatedly branched loop—both optimization approaches demonstrate a slowdown similar to what we observed previously with Branchy. This slowdown occurs because these optimizations successfully prevent tracing and compiling overly specialized complex traces, leading to frequent interpretation of the heavily branched paths. For experiment (C)—the complex randomized input—we observe a significant slowdown for the meta-hints approach. This is because, once `can_enter_jit` is activated by shallow branches, the tracer subsequently follows all branches regardless of their depth, resulting again in long, overly specialized traces. Conversely, the extended green variables approach yielded more promising results, consistent with the earlier Branchy experiments. Since this approach indirectly favors shallow branches without explicitly controlling `can_enter_jit` annotations, it successfully avoids generating large, over-specialized traces while encouraging shallow computations to be JIT compiled.

Consequently, this method seems to effectively balance trace utilization with the performance trade-off between JIT-compilation and interpretation, suggesting its superiority as a potential optimization technique for branch-heavy computations.

While the targeted nature of these approaches prevents them from delivering a clear overall performance improvement for the general case, evidence validates their potential effectiveness in certain scenarios, thus indicating that they are worth further investigation. In particular, the meta-hints strategy can be extended and generalized by allowing dynamic modification of the green variables at the user-code level. Similar to our augmented-depth approach, the interpreter could dynamically adjust the profiling performed by the tracer, providing finer-grained control over tracing behavior. Although such extensions would likely amplify the annotation burden on the user-level code and further specialize the profiling mechanism, this fine-grained control may enable more precise and effective tracing decisions. Additionally, the interpreter could leverage multiple JIT drivers simultaneously—one driver focusing on branch-heavy computations alongside another utilizing the original green variable definitions—to achieve even finer control over trace generation and optimization strategies.

Fundamentally, as discussed extensively in Chapter 6, the challenge lies in identifying optimal tracing strategies for branch-heavy computations. Traces capturing all conditional decisions inevitably become highly specialized and less reusable, making them impractical unless identical execution paths—dictated by specific input patterns—are frequently taken, which is typically unrealistic. Conversely, frequently executed paths that are intentionally not traced—such as those omitted due to our proposed techniques—may save the overhead associated with tracing but progressively degrade runtime performance as they remain interpreted rather than JIT-

compiled. Nevertheless, our evidence indicates that addressing the unique tracing issues arising specifically from self-hosting through targeted approaches, such as those described here, indeed offers a viable path toward efficient self-hosting of functional languages on meta-tracing JIT compilers.

7.2 Mitigating Memory Pressure in Branch-Heavy Computation

A prominent contributor to the performance overhead in self-hosting via meta-tracing JIT compilers is the considerable memory pressure arising from frequent heap allocations, especially continuations and environments. Recall from Chapter 6 that the explicit continuations allocated on the heap are fundamental to the CEK machine’s ability to implement complex control operations, including proper tail-calls, which inherently leads to frequent continuation allocations in Pycket.

To address memory issues without altering the underlying runtime system or garbage collector installed automatically by the RPython framework, we propose incorporating a stackful interpreter alongside the existing CEK machine in Pycket. This hybrid approach leverages the stack to handle certain computation phases, thereby significantly reducing the number of continuation allocations on the heap. The stackful interpreter is realized as a simple recursive interpreter compiled by the RPython framework into a recursive C program utilizing the native stack. For example, when interpreting a `let`-form, instead of creating a `LetCont` to evaluate its body after the right-hand-side like we do in the CEK machine, in the stackful interpreter we recursively evaluate the right-hand-side and then interpret the body, using the native stack for the natural continuation.

Despite its simplicity, the use of a stackful interpreter introduces notable challenges related to stack management, particularly concerning stack overflows due to arbitrarily deep recursion. Furthermore, entirely replacing the CEK interpreter with a stack-based alternative would forgo the performance benefits offered by explicit continuations in the CEK machine, particularly in managing complex control operations and proper tail-calls.

Consequently, we advocate for an approach where the two interpreters operate in tandem, strategically switching between them to optimize overall performance. The primary goal of this dual-interpreter strategy is to judiciously balance heap and stack usage—leveraging rapid stack allocations and improved locality while retaining the sophisticated control and optimization capabilities provided by the heap-based CEK interpreter.

7.2.1 CEK Stackful Prototype on Pycket

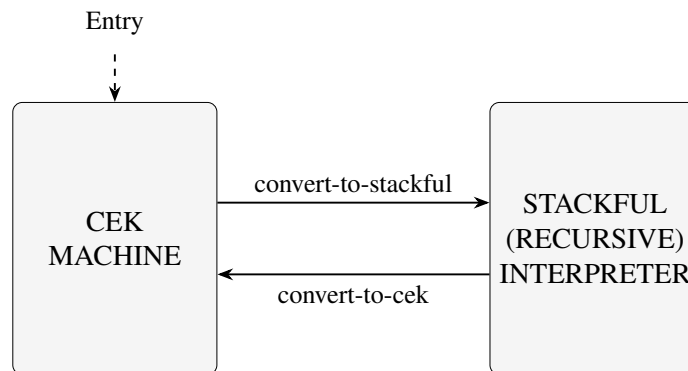


Figure 7.4: Stackful & CEK Switch

We implemented a practical prototype of the hybrid interpreter approach on Pycket. This prototype aligns closely with the conceptual framework illustrated in Figure 7.4, though it

primarily initiates computations using the CEK interpreter. This design choice accommodates Pycket’s need to load and instantiate the bootstrapping linklets at startup, subsequently beginning computations via the `read`, `expand`, and evaluation processes.

In the implemented prototype, the CEK interpreter transfers control to the stackful interpreter upon encountering computational constructs such as `let` or `begin`. The switching mechanism itself is straightforward, via direct invocation of the stackful interpreter. The stackful interpreter is augmented with a trampoline mechanism to efficiently manage tail-call optimizations. Furthermore, it automatically hands control back to the CEK interpreter under three key conditions: (i) completion of computation and return, (ii) execution involving CPS-transformed primitives (e.g. `call/cc`), or (iii) detection of potential stack overflow situations. The return to CEK is handled by unwinding the stack and installing necessary continuations via a `ConvertStack` exception mechanism, effectively allowing seamless continuation of the computation in CEK mode.

To evaluate the runtime performance and demonstrate the effectiveness of our hybrid Stackful+CEK interpreter in reducing memory overhead, we conducted experiments using the Branchy program from Chapter 6 (see Figure 6.4). We ran Branchy on Pycket under two configurations: the hybrid interpreter and the original CEK-only interpreter. The experimental inputs matched those used previously in Section 7.1: one exclusively taking the shallow Exit A path, one exclusively taking the deeper Exit B path, and one with randomized inputs, each consisting of 2000 numbers. As before, the experiments followed the setup detailed in Section 6.3.1, with reported runtimes averaged over 1000 runs and using bootstrapped 95% confidence intervals [39]. To isolate the measurement strictly to runtime-allocated continuations,

the Branchy program intentionally avoids continuation primitives. Computation begins with the CEK interpreter and switches to the stackful interpreter upon encountering the first `let` form, proceeding from there.

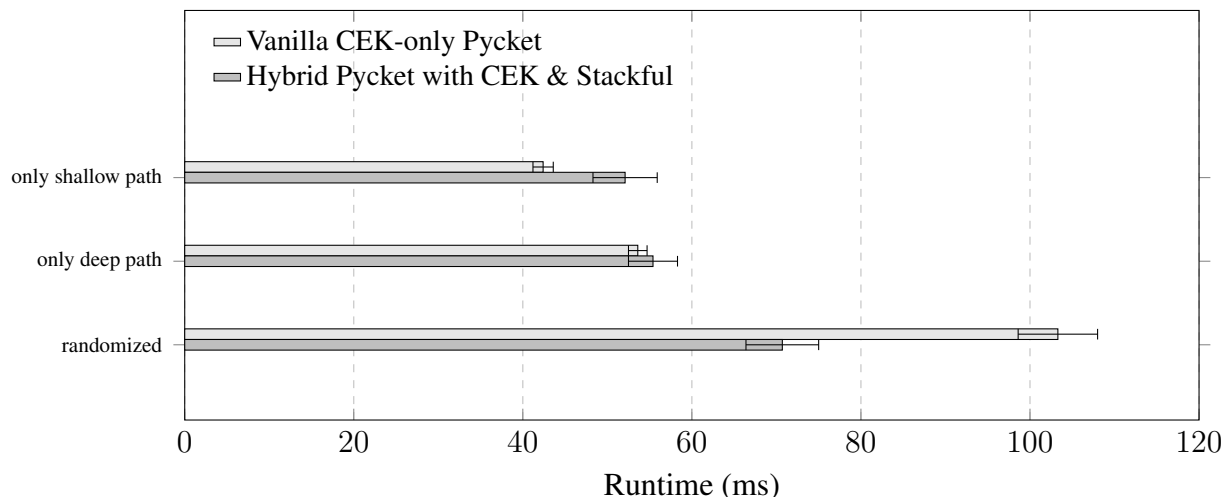


Figure 7.5: Experiment results of CEK-only Pycket vs CEK+Stackful Pycket running Branchy. Lower is better.

Figure 7.5 presents the runtime results for these experiments. We observe slightly longer warmup times for the hybrid Stackful+CEK interpreter in all input scenarios. This is expected because RPython’s meta-tracing JIT is not optimized specifically for recursive interpreter structures, unlike Pycket’s highly-tuned CEK interpreter. Despite this initial overhead, the hybrid interpreter achieves performance comparable to the original CEK for both shallow and deep branch inputs. Notably, in the scenario with randomized inputs, the hybrid interpreter consistently improves overall runtime performance. This improvement arises primarily from the reduced memory footprint, as will be confirmed through the memory analysis shortly, highlighting the viability of using the hybrid Stackful+CEK interpreter in reducing memory

pressure. These preliminary results clearly motivate further research into this hybrid interpreter approach.

	Collections	Bytes copied (KiB)	Old-gen growth (MiB)	GC wall-time (ms)
CEK-only Pycket	32	1161.3	17.8	70.3
Hybrid Pycket	10	2713.4	2.7	20.2

Table 7.1: Memory footprint of CEK-only Pycket, Hybrid Pycket with CEK & Stackful, running Branchy with randomized input. Nursery size: 32M

Table 7.1 presents the memory footprint comparison between the CEK-only Pycket and the hybrid Stackful+CEK Pycket, both running the Branchy program with randomized input. All values in the table represent averages over 1000 runs and correspond to the same experiments presented previously in Figure 7.5. We observe a notable reduction in the number of minor garbage collections, dropping from an average of 32 collections in CEK-only Pycket to just 10 in the hybrid model. While the hybrid interpreter increases the bytes copied—expected due to the need to reconstruct continuations in the heap when switching from stackful mode—it substantially reduces growth in the old-generation heap. This improvement occurs because computations largely take place on the native stack rather than the heap, thus avoiding long-lived continuation allocations. This reduction in memory pressure clearly explains the runtime improvements observed in the randomized-input scenario (Figure 7.5) and further supports the viability of our hybrid interpreter approach.

These preliminary results suggest that the hybrid Stackful+CEK interpreter might offer a potential solution to mitigate memory pressure in self-hosting scenarios. While the evidence is promising, it comes from an isolated experiment with a small prototype. Developing this

prototype further into a complete hybrid interpreter could allow a more thorough evaluation of performance, particularly for larger programs and those involving frequent use of continuations that trigger switches back to the CEK interpreter. However, significant engineering effort would be required to handle interpreter switching robustly, preserve proper tail-calls, and ensure overhead remains bounded. Nonetheless, given these initial promising observations, further investigation into this hybrid approach is indeed warranted.

7.2.2 Formalism for CEK Stackful

To better understand and reason about the hybrid Stackful+CEK interpreter approach, we developed a formal model integrating both interpreters using PLT Redex. The model, illustrated in Figure 7.4, evaluates a simplified subset of Racket similar to the Linklet Kernel language described in Section 3.2, excluding linklet variable forms. Additionally, it introduces special convert forms that explicitly trigger interpreter switches directly from the source code. The grammar for this simplified language is shown in Figure D.1, clearly depicting how deliberate transitions between the interpreters are controlled. This formal model enables further experimentation and analysis, helping to clarify the conditions under which interpreter switching can effectively reduce memory pressure.

The primary goal of this formal model is to allow precise reasoning about reducing heap memory pressure from large, long-lived continuations. Developed as an executable formalism using PLT Redex, the model supports rigorous experimentation with interpreter transitions. The complete reduction semantics for the CEK and stackful interpreters are presented in Appendix D.

```

 $e ::= x \mid v \mid (e \ e \ \dots) \mid (\mathbf{if} \ e \ e \ e) \mid (op \ e \ \dots)$ 
 $\mid (\mathbf{set!} \ x \ e) \mid (\mathbf{begin} \ e \ e \ \dots) \mid (\mathbf{lambda} \ (x_ \ \dots) \ e)$ 
 $\mid (\mathbf{let-values} \ (((x_ ) \ e) \ \dots) \ e) \mid (\mathbf{letrec-values} \ (((x_ ) \ e) \ \dots) \ e)$ 
 $\mid \mathit{internal-letrec} \mid (\mathbf{raises} \ e) \mid (\mathbf{raise-depth}) \mid (\mathbf{convert-stack} \ e)$ 
 $\mid \mathit{convert} \mid \mathit{stuck}$ 

 $v ::= n \mid b \mid c \mid (\mathbf{void})$ 
 $c ::= (\mathbf{closure} \ x \ \dots \ e \ \rho)$ 
 $n ::= \mathit{number}$ 
 $b ::= \mathbf{true} \mid \mathbf{false}$ 
 $x, \mathit{cell} ::= \mathit{variable-not-otherwise-mentioned}$ 
 $op ::= \mathbf{add1} \mid + \mid * \mid < \mid \mathbf{sub1}$ 
 $\rho ::= ((x \ \mathit{any}) \ \dots)$ 
 $\Sigma ::= ((x \ \mathit{any}) \ \dots)$ 

 $\mathit{internal-letrec} ::= (\mathbf{letrec-values-cell-ready} \ (((x_ ) \ e) \ \dots) \ e)$ 
 $\mathit{convert} ::= (\mathbf{convert-to-stackful} \ e) \mid (\mathbf{convert-to-cek} \ e)$ 
 $\mid (\mathbf{convert-stack-to-heap} \ \rho \ \Sigma \ x)$ 
 $\mathit{exception} ::= (\mathbf{stack-depth-exn} \ n) \mid (\mathbf{convert-to-cek-exn} \ e \ \rho \ \Sigma)$ 
 $\mathit{rc-result} ::= v \mid \mathit{stuck}$ 

 $\kappa ::= (x \ \dots)$ 
 $\mid (\mathbf{if-}\kappa \ e \ e)$ 
 $\mid (\mathbf{arg-}\kappa \ (e \ \dots))$ 
 $\mid (\mathbf{fun-}\kappa \ c \ (e \ \dots) \ (v \ \dots))$ 
 $\mid (\mathbf{set-}\kappa \ x)$ 
 $\mid (\mathbf{seq-}\kappa \ e \ \dots)$ 
 $\mid (\mathbf{op-}\kappa \ op \ (v \ \dots) \ (e \ \dots) \ \rho)$ 
 $\mid (\mathbf{let-}\kappa \ (((x) \ e) \ \dots) \ (x \ \dots) \ (v \ \dots) \ e)$ 
 $\mid (\mathbf{letrec-}\kappa \ (((x) \ e) \ \dots) \ x \ e)$ 

```

Figure 7.6: Source Language for CEK & Stackful Hybrid Model

Taking a step towards addressing the fundamental challenges in self-hosting on meta-tracing frameworks—namely, intelligent meta-tracing of programs with complex control flows and effective management of heap memory pressure—we demonstrate viable strategies for achieving efficient self-hosting of Racket on Pycket. We believe the approaches explored herein hold broad applicability, potentially serving as foundational methods for self-hosting on meta-tracing frameworks more generally.

8. CONCLUSION & SIGNIFICANCE

CHAPTER SYNOPSIS

Thesis:

Efficient self-hosting of full-scale functional languages on meta-tracing JIT compilers is achievable.

We:

1. provided a full-scale, self-hosting functional programming language on a meta-tracing JIT compiler as evidence.
2. provided, for the first time, a formal specification of the operational semantics of linklets.
3. exposed performance issues fundamental to self-hosting on meta-tracing JIT compilers, and provided solution approaches.
4. argued that guiding the tracer away from branch-heavy computations can improve performance of tracing interpreter-style, data-dependent code.
5. presented and detailed a technique that uses hybrid computational models in language run-times to improve performance.

Sections:

- Related Work
- Future Work

Thesis:

Efficient self-hosting of a full-scale functional language on a meta-tracing JIT compiler is achievable.

In this dissertation, we provided reproducible evidence demonstrating the feasibility of efficiently implementing a full-scale, self-hosting functional programming language on a meta-tracing JIT compiler. This claim was substantiated through multiple key contributions, which we summarize below.

We developed Pycket, a practical research vehicle, transforming it from a rudimentary interpreter for Racket’s `kernel` language into a complete runtime system capable of supporting Racket. Pycket’s evolution and the critical implementation details are discussed comprehensively in Chapter 4, and the correctness and completeness of this transformation are validated in Chapter 5.

We provided, for the first time, a formal specification of the operational semantics of linklets, a core compilation unit that facilitates improved interaction between the compiler and the runtime, thereby enhancing the portability of Racket. This formalization is described in Chapter 3, with the complete model in PLT Redex presented in Appendix C.

Although a definitive proof of efficiency remains an open problem, we identified and analyzed performance issues fundamental to self-hosting on meta-tracing JIT compilers in Chapter 6. In Chapter 7, we proposed solution approaches to these significant issues, acknowledging that fully resolving them would require extensive further work. We provided initial evidence indicating that these approaches warrant additional investigation.

We contributed to the performance analysis of tracing JITs by proposing more fine-grained control over tracing loops involving deep branching. Specifically, in Chapter 7, we argued that guiding the tracer away from computations heavy with branches can improve the overall performance of tracing interpreter-style, data-dependent code.

Additionally, we presented a new perspective on computational models in language run-times. In Chapter 7, we argued for combining different kinds of reducers rather than committing to a single evaluation model. By leveraging the complementary strengths of multiple evaluation strategies, this hybrid approach has the potential to improve the overall performance of language VMs.

8.1 Related Work

Higher-order dynamic VMs Dynamic VM implementations have gained popularity due to their potential for rapid prototyping, given the high effort involved in implementing new VMs from scratch. Instead of building a VM manually in low-level languages like C, researchers frequently utilize existing general-purpose object-oriented VMs or dynamically generate VMs from high-level specifications. A prominent example is the *GraalVM*, which is a modified Java HotSpot VM that leverages the Truffle framework and the method-based Graal JIT compiler to implement languages such as JavaScript, Ruby, and Python [45]. In contrast, the RPython project introduced automatic VM generation from interpreter specifications via meta-tracing [13]. PyPy, built using RPython, has demonstrated better performance than CPython itself. Other notable meta-tracing systems include *SPUR*, a tracing JIT for CIL bytecode [49, 50].

Tracing JIT VMs Pycket, similar to PyPy, is built upon the RPython meta-tracing framework. Our research focuses on efficient self-hosting within Pycket. Trace-based compilation, initially introduced by the Dynamo project [9], has seen successful application in several commercial VMs such as Mozilla’s *TraceMonkey* JavaScript VM [47] and Adobe’s *Tamarin* ActionScript VM [51], as well as in research-focused implementations like *LuaJIT* [43], *Converge* [52], *Lambdamachine* [53], and *PyHaskell* [54].

Optimizing VM performance in a meta-tracing context As we mentioned in Chapter 2, the *RPython* backend dynamically performs various optimizations, which require alignment with the language interpreter. VM authors typically optimize such systems by either improving interpreter performance directly or by modifying interpreters to produce traces that are more easily optimized by the JIT. Both methods act on the interpreter itself, so improvements typically focus on language-specific patterns. Alongside general JIT improvements such as value promotion and elidable functions, projects like *PyPy* and *Converge* specifically optimize object, class, and module handling to enhance their performance [55].

Self-hosting Bolz et al. argue that achieving good performance in general-purpose VMs requires semantic alignment between the implemented language and its compiler [56, 23]. Our investigation in Chapter 6 similarly studies the relationship between program semantics specific to self-hosting and interpreter performance on tracing JITs. To our knowledge, no other work specifically examines self-hosting in meta-tracing frameworks. However, existing self-hosting systems for dynamic languages, such as the *Tachyon* JavaScript VM, implement bootstrapping compilers entirely by hand-coding in their own languages without automatic generation techniques [2]. Although *Tachyon* differs significantly from *Pycket*, it shares some

similarities, such as the use of CFGs in Static Single Assignment (SSA) form as IR. Moreover, techniques like conditional constant propagation through abstract interpretation [57] and low-level optimizations addressing indirect branching [58] could potentially be adapted to tracing contexts, though our approach specifically targets the language interpreter level, rather than modifications to the underlying *RPython* framework.

Managing abstraction levels A substantial part of the problems addressed in this thesis arises due to the added abstraction level introduced by self-hosting *Racket*, requiring careful semantic alignment with the underlying meta-tracing VM. Related research has focused on reducing complexity in multi-level abstraction systems, using techniques such as type-directed partial evaluation [59] and multi-stage programming [60]. Amin and Rompf, for instance, explored how interpreter towers can be collapsed into compilers to eliminate interpretation overhead, introducing a multi-level lambda calculus and the meta-circular evaluator *Pink* to demonstrate this idea [61]. Another relevant approach, explored by Yermolovich et al., involves trace-based optimization of hierarchically layered VMs, such as running the guest *Lua VM*[43] atop the host *Tamarin VM*[51]. Their approach allowed the guest VM to communicate hints about interpreter loops, specifying code sections the host VM should avoid tracing independently, thus considering the workload of the guest VM.

Space concerns & heap allocated continuations Many studies investigate the relationship between space complexity and performance optimizations in language implementations. Some utilize *repurposed* JITs (RJIT) to leverage dynamic language optimizations, such as simplifying CFGs[62] or applying run-time type feedback[63], particularly when adding dynamic language support to existing static-language VMs.

Higher-level approaches also address these concerns. For instance, *Pycket* employs hidden classes [8], which are conceptually similar to maps introduced by the *Self* project [64] and adopted by *PyPy* as well [23].

In our work, garbage collection overhead caused by long chains of heap-allocated continuations was identified as a significant issue in self-hosting. Allocating activation records on the heap has been extensively studied within compiler contexts, both with explicit continuations [65] and without them [66, 67].

Perhaps most relevant to our investigation involving both stackful and CEK interpreters is the work by Hieb et al. [68], who proposed allocating activation records on the stack without requiring extensive copying when continuations are created. Their approach bounds the amount of copying, which is otherwise potentially unbounded, and also offers techniques for stack overflow and underflow recovery, equivalent to continuation creation and re-instantiation.

To our knowledge, no comprehensive synthesis addressing the challenges and opportunities of self-hosting on a meta-tracing VM exists in the literature. Consequently, direct comparative evaluations with systems identical to *Pycket* are currently impossible. Nonetheless, existing Racket implementations, such as its previous generic JIT-based VM and the current *Chez*-based implementation [5], offer suitable targets for performance comparison involving computations emphasizing different language features across programs with varying characteristics.

8.2 Future Work

Dynamic programming languages typically come with performance overheads introduced by features such as dynamic typing, higher-order functions, and runtime reflection. Meta-tracing JITs hold great potential for minimizing or even eliminating these overheads. The methods and insights presented in this dissertation could be generalized and applied to optimize a wide range of dynamic language features. While we demonstrated this approach using Racket on *Pycket*, future studies might systematically explore similar optimizations in other languages and runtimes. Evaluating and adapting these techniques across diverse language implementations could lead to substantial performance improvements with relatively modest engineering effort.

Having multiple implementations of the same language or system facilitates valuable comparative research on language runtimes and compiler strategies. Our demonstration, Racket on *Pycket*, alongside implementations like *RacketCS*, could serve as the basis for extensive comparative studies across diverse computational patterns. Such studies could help identify complementary strengths and weaknesses of different compilation techniques, runtime architectures, and performance optimizations. Insights drawn from these comparisons could guide future research on run-time systems for dynamically-typed functional languages in general, influencing both theoretical understanding and practical development practices.

This dissertation demonstrated self-hosting on a meta-tracing JIT, using Racket and *Pycket* as a concrete implementation example. However, the general methodology could inspire further studies in various directions. For instance, future research might explore different self-hosting strategies, investigate trade-offs associated with varying abstraction levels, or systematically

assess how different language design choices influence performance when using meta-tracing. Additionally, experimenting with other dynamic languages or diverse interpreter architectures could further refine our understanding of self-hosting dynamics, enabling more systematic approaches for rapid and efficient language implementation.

The presented study also opens interesting avenues for deeper research on meta-tracing techniques themselves. In particular, we touched upon one of the classical problems inherent in tracing JITs: the asymmetric penalty incurred during branch-heavy computations, where the cost of occasional slowdowns can significantly outweigh frequent fast-path optimizations. Future work might investigate methods for predicting, detecting, and effectively managing these cases, including the development of adaptive tracing heuristics, and specialized hybrid evaluation strategies. Such research could lead to meta-tracing compilers that better handle data-dependent and branch-intensive workloads, improving the overall applicability and robustness of tracing-based optimization.

Appendices

APPENDIX A. LINKLET KERNEL LANGUAGE SPECIFICATION

$$\begin{aligned}
 e &::= x \mid v \mid (e \ e \ \dots) \mid (\mathbf{if} \ e \ e \ e) \mid (p1 \ e) \mid (p2 \ e \ e) \\
 &\quad \mid (\mathbf{set!} \ x \ e) \mid (\mathbf{begin} \ e \ e \ \dots) \mid (\mathbf{lambda} \ (x_ \ \dots) \ e) \\
 &\quad \mid (\mathbf{let-values} \ (((x_) \ e) \ \dots) \ e) \mid (\mathbf{raises} \ e) \\
 v &::= n \mid b \mid c \mid (\mathbf{void}) \\
 c &::= (\mathbf{closure} \ x \ \dots \ e \ \Sigma) \\
 n &::= \textit{number} \\
 b &::= \mathbf{true} \mid \mathbf{false} \\
 x &::= \textit{variable-not-otherwise-mentioned} \\
 p1 &::= \mathbf{add1} \\
 p2 &::= + \mid * \mid < \\
 o &::= p1 \mid p2 \\
 E &::= [] \mid (v \ \dots \ E \ e \ \dots) \mid (o \ v \ \dots \ E \ e \ \dots) \mid (\mathbf{if} \ E \ e \ e) \\
 &\quad \mid (\mathbf{begin} \ v \ \dots \ E \ e \ \dots) \mid (\mathbf{set!} \ x \ E) \\
 &\quad \mid (\mathbf{let-values} \ (((x) \ v) \ \dots \ ((x) \ E) \ ((x) \ e) \ \dots) \ e) \\
 \Sigma &::= ((x \ \textit{any}) \ \dots) \\
 \sigma &::= ((x \ \textit{any}) \ \dots)
 \end{aligned}$$

Figure A.1: Linklet Kernel Language Grammar

$[E[(\text{raises } e)] \Sigma \sigma] \longrightarrow [(\text{raises } e) \Sigma \sigma]$	[error]
$[E[x] \Sigma \sigma] \longrightarrow [E[\text{lookup}[\sigma, x_i]] \Sigma \sigma]$ where $x_i = \text{lookup}[\Sigma, x]$	[lookup]
$[E[(\text{lambda } (x \dots) e)] \Sigma \sigma] \longrightarrow [E[(\text{closure } x \dots e \Sigma)] \Sigma \sigma]$	[closure]
$[E[(\text{set! } x \ v)] \Sigma \sigma] \longrightarrow [E[(\text{void})] \Sigma \text{extend}[\sigma, (c_i), (v)]]]$ where $(\text{lookup}[\Sigma, x] \neq (\text{raises } x)), c_i = \text{lookup}[\Sigma, x]$	[set!]
$[E[(\text{begin } v_1 \dots v_n)] \Sigma \sigma] \longrightarrow [E[v_n] \Sigma \sigma]$	[begin]
$[E[(\text{let-values } (((x) \ v) \dots) e)] \Sigma \sigma] \longrightarrow [E[e] \text{extend}[\Sigma, (x \dots), (x_2 \dots)] \text{extend}[\sigma, (x_2 \dots), (v \dots)]]]$ where $(x_2 \dots) = (\text{variables-not-in } e \ (x \dots))$	[let]
$[E[(\text{if } v_0 \ e_1 \ e_2)] \Sigma \sigma] \longrightarrow [E[e_1] \Sigma \sigma]$ where $(v_0 \neq \text{false})$	[if-true]
$[E[(\text{if false } e_1 \ e_2)] \Sigma \sigma] \longrightarrow [E[e_2] \Sigma \sigma]$	[if-false]
$[E[(\text{o } v_1 \ v_2 \dots)] \Sigma \sigma] \longrightarrow [E[\delta[(\text{o } v_1 \ v_2 \dots)]] \Sigma \sigma]$	[δ]
$[E[(\text{closure } x \dots e \Sigma_1) \ v \dots_n] \Sigma_2 \sigma] \longrightarrow [E[e] \text{extend}[\Sigma_1, (x \dots), (c_2 \dots)] \text{extend}[\sigma, (c_2 \dots), (v \dots)]]]$ where $(c_2 \dots) = (\text{variables-not-in } e \ (x \dots))$	[βv]

Figure A.2: Linklet Kernel Language standard reduction relation

$\text{run-rc}[(n \ \Sigma \ \sigma)] = n$
 $\text{run-rc}[(b \ \Sigma \ \sigma)] = b$
 $\text{run-rc}[(c \ \Sigma \ \sigma)] = \text{closure}$
 $\text{run-rc}[(\text{void}) \ \Sigma \ \sigma] = (\text{void})$
 $\text{run-rc}[(\text{raises } e) \ \Sigma \ \sigma] = \text{stuck}$
 $\text{run-rc}[any_i] = \text{run-rc}[any_{again}]$
 where $(any_{again}) = (\text{apply-reduction-relation } \rightarrow \beta \text{ s (term } any_i))$
 $\text{run-rc}[any_i] = \text{stuck}$

Figure A.3: Linklet Kernel Language evaluator

APPENDIX B. COMPLETE REDUCTION STEPS FOR TOP-LEVEL EXAMPLE PROGRAM

	program	ρ	σ
	<pre>(program ([l1 (linklet () (a k) (define-values (k) (lambda () a)) (void))) [l2 (linklet () (a) (define-values (a) 10) (void))] [l3 (linklet () (k) (k))]) (let-inst t (make-instance) (seq (ϕ^I l1 #:t t) (ϕ^I l2 #:t t) (ϕ^I l3 #:t t))))</pre>	[]	[]
$\xrightarrow{\tau_{\beta p}^*}$	<pre>(program () (let-inst t (make-instance) (seq (ϕ^I (Lα () ((Export a1 a a) (Export k1 k k)) (define-values (k) (lambda () (var-ref a1))) (var-set! k1 k) (void)) #:t t) (ϕ^I (Lα () ((Export a1 a a)) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))))</pre>	[]	[]
$\xrightarrow{\tau_{\beta p}^*}$	<pre>(program () (seq (ϕ^I (Lβ t (define-values (k) (lambda () (var-ref a1))) (var-set! k1 k) (void))) (ϕ^I (Lα () ((Export a1 a a)) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))))</pre>	$[k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[var_a, var_k \rightarrow \text{uninit},$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$

	program	ρ	σ
$\longrightarrow_{\beta p}^*$	<pre> (program () (seq (ϕ^I (Lβ t (var-set! k1 k) (void))) (ϕ^I (Lα () ((Export a1 a a)) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))) </pre>	$[k \rightarrow cell_1,$ $k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a, var_k \rightarrow \text{uninit},$ $t \rightarrow (LI$ $(a var_a) (k var_k))]$
$\longrightarrow_{\beta p}^*$	<pre> (program () (seq (void) (ϕ^I (Lα () ((Export a1 a a)) (define-values (a) 10) (var-set! a1 a) (void)) #:t t) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))) </pre>	$[]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow \text{uninit},$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a var_a) (k var_k))]$
$\longrightarrow_{\beta p}^*$	<pre> (program () (seq (void) (ϕ^I (Lβ t (define-values (a) 10) (var-set! a1 a) (void))) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))) </pre>	$[a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow \text{uninit},$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a var_a) (k var_k))]$
$\longrightarrow_{\beta p}^*$	<pre> (program () (seq (void) (ϕ^I (Lβ t (var-set! a1 a) (void))) (ϕ^I (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t))) </pre>	$[a \rightarrow 10,$ $a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow \text{uninit},$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a var_a) (k var_k))]$

	program	ρ	σ
$\longrightarrow^*_{\beta p}$	<pre>(program () (seq (void) (void) (ϕ^I) (Lα () ((Export k1 k k)) ((var-ref k1))) #:t t)))</pre>	[]	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow 10,$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$
$\longrightarrow^*_{\beta p}$	<pre>(program () (seq (void) (void) (ϕ^I) (Lβ t ((var-ref k1))))))</pre>	$[k1 \rightarrow var_k]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow 10,$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$
$\longrightarrow^*_{\beta p}$	<pre>(program () (seq (void) (void) (ϕ^I) (Lβ t ((lambda () (var-ref a1))))))</pre>	$[k1 \rightarrow var_k]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow 10,$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$
$\longrightarrow^*_{\beta p}$	<pre>(program () (seq (void) (void) ((lambda () (var-ref a1))))))</pre>	$[k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow 10,$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$
$\longrightarrow^*_{\beta p}$	<pre>(program () (seq (void) (void) 10))</pre>	$[k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow 10,$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$
$\longrightarrow^*_{\beta p}$	10	$[k1 \rightarrow var_k,$ $a1 \rightarrow var_a]$	$[cell_1 \rightarrow \text{closure},$ $var_a \rightarrow 10,$ $var_k \rightarrow cell_1$ $t \rightarrow (LI$ $(a\ var_a)\ (k\ var_k))]$

APPENDIX C. PLT REDEX MODEL FOR LINKLET SEMANTICS

The whole model is available online at <https://github.com/cderici/linklets-redex-model/tree/master>.

```
.....  
;; lang.rkt  
.....  
  
(define-language RC  
  [e ::= x v (e e ...) (if e e e) (o e e)  
    (begin e e ...) (lambda (x!_ ...) e)  
    (raises e) (set! x e)  
    (var-ref x) (var-ref/no-check x)  
    (var-set! x e) (var-set/check-undef! x e)] ;; expressions  
  [v ::= n b c (void) unit] ;; values  
  [c ::= (closure (x ...) e  $\rho$ )]  
  [n ::= number]  
  [b ::= true false]  
  [x cell ::= variable-not-otherwise-mentioned] ;; variables  
  [o ::= + * <]  
  [E ::= hole (v ... E e ...) (o E e) (o v E)  
    (var-set! x E) (var-set/check-undef! x E)  
    (begin v ... E e ...) (set! x E) (if E e e)] ;; eval context  
  [ $\rho$  ::= ((x any) ...)] ;; environment  
  [ $\sigma$  ::= ((x any) ...)] ;; store  
  
  [e-test ::= x n b (void)  
    (e-test e-test ...) (lambda (x!_ ...) e-test) (if e-test e-test e-test)  
    (p1 e-test e-test) (p1 e-test) (set! x e-test) (begin e-test e-test ...)  
    (raises e-test)] ;; to be used to generate test cases (i.e. exclude closures)  
)
```

```

(define-extended-language LinkletSource RC
  [L ::= (linklet ((imp-id ...) ...) (exp-id ...) l-top ... e)]

  [l-top ::= (define-values (x) e) e] ; linklet body expressions

  ;; (external-imported-id internal-imported-id)
  [imp-id ::= x (x x)]
  ;; (internal-exported-id external-exported-id)
  [exp-id ::= x (x x)]

(define-extended-language Linklets LinkletSource
  ;; compile
  [CL ::= (compile-linklet L)]
  [L-obj ::= (L $\alpha$  c-imps c-exps l-top l-top ...) (L $\beta$  x l-top ...)]
  [c-imps ::= ((imp-obj ...) ...)]
  [c-exps ::= (exp-obj ...)]
  ;; import & export objects
  [imp-obj ::= (Import n x x x)] ; group-index id(<-gensymed) int_id ext_id
  [exp-obj ::= (Export x x x)] ; int_gensymed int_id ext_id

  ;; instantiate
  [LI ::= x (linklet-instance (x cell) ...)] ; note that an instance have no exports
  [I ::= (make-instance)
    (instantiate-linklet linkl-ref x ...)
    (instantiate-linklet linkl-ref x ... #:target x)]

  [linkl-ref ::= x L-obj (raises e)]

  [v ::= .... (v x)]
  ;; program-stuff
  [p ::= (program (use-linklets (x!_ L) ...) p-top)]
  [p-top ::= v I (let-inst x p-top p-top) (seq p-top ...)
    (instance-variable-value x x)]

```

```

;; evaluation-context for the programs
[EP ::= hole
  (instantiate-linklet (L $\beta$  x v ... EP l-top ...) x ...) ;; instantiate
  (define-values (x) EP)
  (let-inst x EP p-top)
  (seq v ... EP p-top ...)

  (program (use-linklets) EP)]

;; evaluation-context for the linklet body
[EI ::= hole (L $\alpha$  ((imp-obj ...) ...) (exp-obj ...) v ... EI l-top ...)]
)

(define-extended-language LinkletProgramTest Linklets
  [p-test ::= (program (use-linklets (x!_ L) ...) p-top-test ...)]
  [p-top-test ::= (instantiate-linklet x x ... #:target I-test)
    (instantiate-linklet x x ...)
    (let-inst x (instantiate-linklet x x ...) p-top-test)
    (instance-variable-value x x)
    v-test]
  [I-test ::= x (linklet-instance)]
  [v-test ::= n b (void)])

.....
;; racket-core.rkt
.....

(define-metafunction RC
  [(let-values (((x) e) ...) e_body)
    ((lambda (x ...) e_body) e ...)])

(define-metafunction RC
   $\delta$  : (o any any) -> v or true or false or (raises e)
  [( $\delta$  (< n_1 n_2)) ,(if (< (term n_1) (term n_2))
    (term true) (term false))]
  [( $\delta$  (+ n_1 n_2)) ,(+ (term n_1) (term n_2))])

```



```

[(δ (* n_1 n_2)) ,(* (term n_1) (term n_2))]
[(δ (o any_1 any_2)) (raises (o any_1 any_2))]

(define-metafunction RC
  extend : ((x any) ...) (x ...) (any ...) -> ((x any) ...)
  [(extend ((x any) ...) (x_1 ...) (any_1 ...))
   ((x_1 any_1) ... (x any) ...)]

(define-metafunction RC
  lookup : ((x any) ...) x -> any
  [(lookup ((x_1 any_1) ... (x any_t) (x_2 any_2) ...) x)
   any_t
   (side-condition (not (member (term x) (term (x_1 ...)))))]
  [(lookup any_1 any_2)
   (raises any_1)]

;; standard reduction
(define -->βr
  (reduction-relation
   Linklets
   #:domain (p ρ σ)
   (--> [(in-hole EP (in-hole E (raises e))) ρ σ]
        [(in-hole EP (raises e)) ρ σ] "error")

   (--> [(in-hole EP (in-hole E x)) ρ σ]
        [(in-hole EP (in-hole E (lookup σ x_1))) ρ σ] "lookup"
        (where x_1 ,(term (lookup ρ x))))

   (--> [(in-hole EP (in-hole E (var-ref x))) ρ σ]
        [(in-hole EP (in-hole E v)) ρ σ]
        (where v (lookup σ (lookup ρ x)))
        "var-ref")

   (--> [(in-hole EP (in-hole E (var-ref/no-check x))) ρ σ]
        [(in-hole EP (in-hole E v)) ρ σ]
        (where v (lookup σ (lookup ρ x)))

```

```

"var-ref/no-check") ; for now the same with var-ref

(--> [(in-hole EP (in-hole E (var-set! x v)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E (void)))  $\rho$  (extend  $\sigma$  (cell_var) (v))]
      (where cell_var (lookup  $\rho$  x))
      "var-set!")

(--> [(in-hole EP (in-hole E (var-set/check-undef! x v)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E (void)))  $\rho$  (extend  $\sigma$  (cell_var) (v))]
      (where cell_var (lookup  $\rho$  x))
      (where v_var (lookup  $\sigma$  cell_var)) ; to make sure it's there
      "var-set/check-undef!") ; for now the same with var-set!

(--> [(in-hole EP (in-hole E (lambda (x ...) e)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E (closure (x ...) e  $\rho$ )))  $\rho$   $\sigma$ ] "closure")

(--> [(in-hole EP (in-hole E (set! x v)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E (void)))  $\rho$  (extend  $\sigma$  (x_1) (v))]
      (side-condition (not (equal? (term (raises ,(term x))) (term (lookup  $\rho$  x)))))
      (where x_1 ,(term (lookup  $\rho$  x)) "set!")

(--> [(in-hole EP (in-hole E (begin v_1 ... v_n)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E v_n))  $\rho$   $\sigma$ ] "begin")

(--> [(in-hole EP (in-hole E (if v_0 e_1 e_2)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E e_1))  $\rho$   $\sigma$ ]
      (side-condition (not (equal? (term v_0) (term false)))) "if-true")

(--> [(in-hole EP (in-hole E (if false e_1 e_2)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E e_2))  $\rho$   $\sigma$ ] "if-false")

(--> [(in-hole EP (in-hole E (o v_1 v_2)))  $\rho$   $\sigma$ ]
      [(in-hole EP (in-hole E ( $\delta$  (o v_1 v_2))))  $\rho$   $\sigma$ ] " $\delta$ ")

(--> [(in-hole EP (in-hole E ((closure (x ..._n) e  $\rho_1$ ) v ..._n)))  $\rho_2$   $\sigma$ ]
      [(in-hole EP (in-hole E e)) (extend  $\rho_1$  (x ...) (x_2 ...)) (extend  $\sigma$  (x_2 ...) (v ...))] " $\beta v$ "
      (where (x_2 ...) ,(variables-not-in (term e) (term (x ...))))))

```

```

.....
;; compile-linklets.rkt
.....

(define-extended-language LinkletsCompile Linklets
  [exprs ::= (l-top ...)]
  [lex-ids ::= (x ...)]
  [mut-ids ::= (x ...)]
  [top-ids ::= (x ...)])

; A separate pass
(define-metafunction LinkletsCompile
  all-toplevels : (l-top ...) (x ...) -> (x ...)
  [(all-toplevels () (x ...)) (x ...)]
  [(all-toplevels ((define-values (x) e) l-top ...) (x_tops ...))
   (all-toplevels (l-top ...) (x_tops ... x)))]
  [(all-toplevels (l-top_1 l-top ...) (x ...))
   (all-toplevels (l-top ...) (x ...))])

; A separate (slightly deeper) pass
(define-metafunction LinkletsCompile
  get-mutated-vars-expr : l-top (x ...) -> (x ...)
  [(get-mutated-vars-expr (set! x v) (x_muts ...)) (x x_muts ...)]
  [(get-mutated-vars-expr (begin l-top ...) (x_muts ...))
   (get-all-mutated-vars (l-top ...) (x_muts ...))]
  [(get-mutated-vars-expr l-top (x ...)) (x ...)])

(define-metafunction LinkletsCompile
  get-all-mutated-vars : (l-top ...) (x ...) -> (x ...)
  [(get-all-mutated-vars () (x ...)) (x ...)]
  [(get-all-mutated-vars (l-top_1 l-top ...) (x_muts ...))
   (get-all-mutated-vars (l-top ...) (x_muts ... x_new_muts ... ))
   (where (x_new_muts ...) (get-mutated-vars-expr l-top_1 ())))])

; Process Imports

```

```

(define-metafunclion LinkletsCompile
  process-import : n (imp-id ...) (imp-obj ...) -> (imp-obj ...)
  [(process-import n () (imp-obj ...)) (imp-obj ...)]
  [(process-import n (x imp-id ...) (imp-obj ...))
   (process-import n (imp-id ...) (imp-obj ... (Import n x_gen x x) ))
   (where x_gen ,(variable-not-in (term (x imp-id ...)) (term x)))]
  [(process-import n ((x_ext x_int) imp-id ...) (imp-obj ...))
   (process-import n (imp-id ...) (imp-obj ... (Import n x_gen x_int x_ext)))
   (where x_gen ,(variable-not-in (term ((x_ext x_int) imp-id ...)) (term x)))]])

(define-metafunclion LinkletsCompile
  process-importss : n ((imp-id ...) ...) ((imp-obj ...) ...) -> ((imp-obj ...) ...)
  [(process-importss n () ((imp-obj ...) ...)) ((imp-obj ...) ...)]
  [(process-importss n ((imp-id_1 ...) (imp-id ...) ...) ((imp-obj ...) ...))
   (process-importss ,(add1 (term n))
                      ((imp-id ...) ...)
                      ((imp-obj ...) ... (imp-obj_1 ...)))
   (where (imp-obj_1 ...) (process-import n (imp-id_1 ...) ())))])

; Process Exports
(define-metafunclion LinkletsCompile
  process-exports : (exp-id ...) (exp-obj ...) -> (exp-obj ...)
  [(process-exports () (exp-obj ...)) (exp-obj ...)]
  [(process-exports (x exp-id ...) (exp-obj ...))
   (process-exports (exp-id ...) (exp-obj ... (Export x_gen x x)))
   (where x_gen ,(variable-not-in (term (x exp-id ...)) (term x)))]
  [(process-exports ((x_int x_ext) exp-id ...) (exp-obj ...))
   (process-exports (exp-id ...) (exp-obj ... (Export x_gen x_int x_ext)))
   (where x_gen ,(variable-not-in (term ((x_int x_ext) exp-id ...)) (term x)))]])

```

```

#|
---- Define-values

if we see a define-values, we look at exported vars.
if the defined id is exported, we add a new linklet variable for that.
export's internal id should be the same with the defined id.
we create the variable with the exports internal gensym.
|#

(define-metafunction LinkletsCompile
  c-def-val : x e c-exps -> exprs
  [(c-def-val x e_body (exp-obj_before ... (Export x_gen x_ext) exp-obj_after ...))
   ((define-values (x) e_body) (var-set! x_gen x))]
  [(c-def-val x e_body c-exps)
   ((define-values (x) e_body))])

```

```

#|
---- Symbol

Important parts are if the symbol is one of exports or imports.
If it's not, then it's either a toplevel defined (within linklet)
or a primitive (op)(which is handled in a separate case below)
|#

(define-metafunction LinkletsCompile
  c-symbol : x mut-ids top-ids c-imps c-exps -> l-top
  ; 1) if it's one of imports
  [(c-symbol x_current mut-ids top-ids
            ((imp-obj_before ...) ...
             ((Import n_bef x_gen_bef x_int_bef x_ext_bef) ...
              (Import n_cur x_gen_cur x_current x_ext_cur)
              (Import n_aft x_gen_aft x_int_aft x_ext_aft) ...)
             (imp-obj_after ...) ...) c-exps)
   (var-ref/no-check x_gen_cur)]
  ; 2-a) if it's one of exports, and it is mutated
  [(c-symbol x_current mut-ids top-ids c-imps
            ((Export x_gen_bef x_int_bef x_ext_bef) ...
             (Export x_gen_cur x_current x_ext_cur)
             (Export x_gen_aft x_int_aft x_ext_aft) ...))

```

```

    (var-ref x_gen_cur)

    (side-condition (member (term x_current) (term mut-ids))))]
; 2-b) if it's one of exports, and not defined in the toplevel
; (within the linklet)
[(c-symbol x_current mut-ids top-ids c-imps
  ((Export x_gen_bef x_int_bef x_ext_bef) ...
   (Export x_gen_cur x_current x_ext_cur)
   (Export x_gen_aft x_int_aft x_ext_aft) ...))

  (var-ref x_gen_cur)

  (side-condition (not (member (term x_current) (term top-ids)))))]
; 3) symbol is neither import nor export, treat normal
[(c-symbol x_current mut-ids top-ids c-imps c-exps) x_current]]

;; ---- Set!
(define-metafunction LinkletsCompile
  c-set-bang : x c-exps e -> l-top
  [(c-set-bang x ((Export x_gen_bef x_int_bef x_ext_bef) ...
    (Export x_gen_cur x x_ext_cur)
    (Export x_gen_aft x_int_aft x_ext_aft) ...))
    e_rhs)
  (var-set/check-undef! x_gen_cur e_rhs)]
  [(c-set-bang x c-exps e_rhs) (set! x e_rhs)])

;; c-body == sexp_to_ast (in Pycket)
(define-metafunction LinkletsCompile
  c-body : exprs lex-ids c-imps c-exps mut-ids top-ids exprs -> exprs
  ; base case
  [(c-body () lex-ids c-imps c-exps mut-ids top-ids exprs) exprs]
  ; define-values
  [(c-body ((define-values (x) e) l-top ...) (x_lex ...) c-imps c-exps mut-ids top-ids (l-top_compiled ...))
    (c-body (l-top ...) (x_lex ...) c-imps c-exps mut-ids top-ids (l-top_compiled ... l-top_def_val ...))
    (where (e_new) (c-body (e) (x x_lex ...) c-imps c-exps mut-ids top-ids ()))
    (where (l-top_def_val ...) (c-def-val x e_new c-exps)))]
  ; symbols
  [(c-body (x_current l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))

```

```

(c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... l-top_sym))
(when (l-top_sym (c-symbol x_current mut-ids top-ids c-imps c-exps)))
; set!
[(c-body ((set! x e) l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
(c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... l-top_set_bang))
(when (e_new) (c-body (e) lex-ids c-imps c-exps mut-ids top-ids ()))
(when (l-top_set_bang (c-set-bang x c-exps e_new)))]
; others
[(c-body exprs lex-ids c-imps c-exps mut-ids top-ids exprs_compiled)
(c-expr exprs lex-ids c-imps c-exps mut-ids top-ids exprs_compiled)]

(define-metafunction LinkletsCompile
  c-expr : exprs lex-ids c-imps c-exps mut-ids top-ids exprs -> exprs
  ; values
  [(c-expr (v l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... v)))]
  ; begin
  [(c-expr ((begin e ...) l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... (begin e_new ...)))
  (when (e_new ...) (c-body (e ...) lex-ids c-imps c-exps mut-ids top-ids ()))]]
  ; op
  [(c-expr ((o e ...) l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... (o e_new ...)))
  (when (e_new ...) (c-body (e ...) lex-ids c-imps c-exps mut-ids top-ids ()))]]
  ; if
  [(c-expr ((if e_tst e_thn e_els) l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... (if e_tst_new e_thn_new
    e_els_new)))
  (when (e_tst_new) (c-body (e_tst) lex-ids c-imps c-exps mut-ids top-ids ()))
  (when (e_thn_new) (c-body (e_thn) lex-ids c-imps c-exps mut-ids top-ids ()))
  (when (e_els_new) (c-body (e_els) lex-ids c-imps c-exps mut-ids top-ids ()))]]
  ; lambda
  [(c-expr ((lambda (x ...) e_body) l-top ...) (x_lex ...) c-imps c-exps mut-ids top-ids (l-top_compiled
    ...))
  (c-body (l-top ...) (x_lex ...) c-imps c-exps mut-ids top-ids (l-top_compiled ... (lambda (x ...)

```

```

    e_body_new)))

  (where (e_body_new) (c-body (e_body) (x ... x_lex ...) c-imps c-exps mut-ids top-ids ()))]

; let-values
[(c-expr ((let-values (((x_rhs) e_rhs) ...) e_body) l-top ...) (x_lex ...)
          c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) (x_lex ...) c-imps c-exps mut-ids top-ids
          (l-top_compiled ... (let-values (((x_rhs) e_rhs_new) ...) e_body_new)))
  (where (e_rhs_new ...) (c-body (e_rhs ...) (x_lex ...) c-imps c-exps mut-ids top-ids ()))
  (where (e_body_new) (c-body (e_body) (x_rhs ... x_lex ...) c-imps c-exps mut-ids top-ids ()))])

; raises
[(c-expr ((raises e) l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... (raises e)))]

; app
[(c-expr ((e ...) l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ...))
  (c-body (l-top ...) lex-ids c-imps c-exps mut-ids top-ids (l-top_compiled ... (e_new ...)))
  (where (e_new ...) (c-body (e ...) lex-ids c-imps c-exps mut-ids top-ids ()))])

(define-metafunction LinkletsCompile
  compile-linklet : L -> L-obj or stuck
  [(compile-linklet (linklet ((imp-id ...) ...) (exp-id ...) l-top ...))
   (L $\alpha$  c-imps c-exps l-top_compiled ...)
   ; where
   (where c-imps (process-importss 0 ((imp-id ...) ...) ()))
   (where c-exps (process-exports (exp-id ...) ()))
   (where mut-ids (get-all-mutated-vars (l-top ...) ()))
   (where top-ids (all-toplevels (l-top ...) ()))
   (where (l-top_compiled ...)
    (c-body (l-top ...) () c-imps c-exps mut-ids top-ids ()))])

```



```

.....
;; linklets.rkt
.....

#|
Instantiating a linklet is basically getting the imported vars into
the env and evaluating all the forms in the body in the presence of
a "target" linklet instance.

If a target instance is not provided to the instantiation (as an
initial argument), then it's a regular instantiation, we will create a
new instance and evaluate all the forms in the linklet body and the
instantiation will return the created linklet instance (the variables
inside the created instance depends on the evaluated forms within the
linklet body).

If a target is provided to the instantiation, then the instantiation
will take place similarly, but the result will be the result of
evaluating the last expression in the linklet body, i.e. the
instantiation will return a value instead of an instance. This is what
we call "evaluating a linklet".
|#

(define -->βp
  (reduction-relation
    Linklets
    #:domain (p ρ σ)
    #:(--> [(in-hole EP (raises e)) ω Ω ρ σ]
      [(raises e) ω Ω ρ σ] "error")
    (--> [(program (use-linklets (x_1 L_1) (x L) ...) p-top) ρ σ]
      [(program (use-linklets (x L) ...) p-top_new) ρ σ]
      (where L-obj_1 (compile-linklet L_1))
      (where (p-top_new) (substitute-linklet x_1 L-obj_1 (p-top))) "compile and load")

    (--> [(in-hole EP (make-instance)) ρ σ]
      [(in-hole EP ((void) x_li)) ρ σ_1]

```

```

    (where x_li ,(variable-not-in (term  $\sigma$ ) (term li)))
    (where  $\sigma_1$  (extend  $\sigma$  (x_li) ((linklet-instance)))) "make-instance")
(--> [(in-hole EP (instance-variable-value x_li x))  $\rho$   $\sigma$ ]
      [(in-hole EP (v x_li))  $\rho$   $\sigma$ ]
      (where v (lookup  $\sigma$  (get-var-from-instance x x_li  $\sigma$ ))) "instance variable value")
(--> [(in-hole EP (let-inst x (v x_i) p-top))  $\rho$   $\sigma$ ]
      [(in-hole EP p-top)  $\rho$  (extend  $\sigma$  (x) (LI))]]
      (where LI (lookup  $\sigma$  x_i)) "let-inst")
(--> [(in-hole EP (seq v_1 ... v_n))  $\rho$   $\sigma$ ]
      [(in-hole EP v_n)  $\rho$   $\sigma$ ] "seq")

(--> [(in-hole EP (instantiate-linklet (L $\beta$  x_target v ... v_last)))  $\rho$   $\sigma$ ]
      [(in-hole EP (v_last x_target))  $\rho$   $\sigma$ ] "return instance/value")

(--> [(in-hole EP (instantiate-linklet (L $\beta$  x_target v_prev ... (define-values (x) v) l-top ...)))  $\rho$   $\sigma$ ]
      [(in-hole EP (instantiate-linklet (L $\beta$  x_target v_prev ... l-top ...)))  $\rho_1$   $\sigma_1$ ]
      (where cell ,(variable-not-in (term (x  $\rho$   $\sigma$ )) (term cell_1)))
      (where ( $\rho_1$   $\sigma_1$ ) ((extend  $\rho$  (x) (cell)) (extend  $\sigma$  (cell) (v)))))

(--> [(in-hole EP (instantiate-linklet (L $\alpha$  c-imps c-exps l-top ...) LI ...))  $\rho$   $\sigma$ ]
      [(in-hole EP (instantiate-linklet (L $\alpha$  c-imps c-exps l-top ...) LI ... #:target x_target))  $\rho$   $\sigma_1$ ]
      (where x_target ,(variable-not-in (term  $\sigma$ ) (term li)))
      (where  $\sigma_1$  (extend  $\sigma$  (x_target) ((linklet-instance))))))

(--> [(in-hole EP (instantiate-linklet (L $\alpha$  c-imps c-exps l-top ...) LI ... #:target x_target))  $\rho$   $\sigma$ ]
      [(in-hole EP (instantiate-linklet (L $\beta$  x_target l-top ...)))  $\rho_2$   $\sigma_1$ ]
      ; set the stage for target/imports/exports
      (where  $\rho_1$  (instantiate-imports c-imps (LI ...)  $\rho$   $\sigma$ ))
      (where ( $\rho_2$   $\sigma_1$ ) (instantiate-exports c-exps x_target  $\rho_1$   $\sigma$ ))
      "set the stage for evaluation"))

```

```
.....
.....
```

```
;; Instantiation Utilities
```

```
.....
.....
```

```
;; Utils for Imports
```

```
(define-metafunction Linklets
  get-var-from-instance : x x  $\sigma$  -> cell
  [(get-var-from-instance x x_li  $\sigma$ )
   cell
   (where (linklet-instance (x_bef cell_bef) ... (x cell) (x_aft cell_aft) ...)
    (lookup  $\sigma$  x_li))]]
  [(get-var-from-instance x LI  $\sigma$ ) (raises not-found)]]
```

```
(define-metafunction Linklets
  get-instance : n n (LI ...) -> LI
  [(get-instance n n (LI LI_rest ...)) LI]
  [(get-instance n_1 n_2 (LI LI_rest ...))
   (get-instance ,(add1 (term n_1)) n_2 (LI_rest ...))]]
```

```
(define-metafunction Linklets
  process-import-group : (imp-obj ...) (LI ...)  $\rho$   $\sigma$  ->  $\rho$ 
  [(process-import-group () (LI ...)  $\rho$   $\sigma$ )  $\rho$ ]
  [(process-import-group ((Import n x_id x_int x_ext) imp-obj_rest ...)
   (LI ...)  $\rho$   $\sigma$ )
   (process-import-group (imp-obj_rest ...) (LI ...)  $\rho_{-1}$   $\sigma$ )
   (where LI_inst (get-instance 0 n (LI ...)))
   (where cell_var (get-var-from-instance x_ext LI_inst  $\sigma$ ))
   (where  $\rho_{-1}$  (extend  $\rho$  (x_id) (cell_var))))]]
```

```
(define-metafunction Linklets
  instantiate-imports : c-imps (LI ...)  $\rho$   $\sigma$  ->  $\rho$ 
  [(instantiate-imports () (LI ...)  $\rho$   $\sigma$ )  $\rho$ ]
  [(instantiate-imports ((imp-obj ...) (imp-obj_rest ...) ...)
   (LI ...)  $\rho$   $\sigma$ )
   (instantiate-imports ((imp-obj_rest ...) ...) (LI ...)  $\rho_{-1}$   $\sigma$ )
```

```

    (where  $\rho_1$  (process-import-group (imp-obj ...) (LI ...)  $\rho$   $\sigma$ )))

; Utils for Exports
(define-metafunction Linklets
  process-one-export : exp-obj x  $\rho$   $\sigma \rightarrow (\rho$   $\sigma)$ 
  ; target has it
  [(process-one-export (Export x_gen x_id x_ext) x_target  $\rho$   $\sigma$ )
   ( $\rho_1$   $\sigma$ ) ; <- same store ( $\sigma$ ) and instances ( $\Omega$ ), i.e. don't create new variable
   (where (linklet-instance (x_bef cell_bef) ... (x_ext cell) (x_aft cell_aft) ...)
           (lookup  $\sigma$  x_target))
   (where  $\rho_1$  (extend  $\rho$  (x_gen) (cell))))]
  ; target doesn't have it
  [(process-one-export (Export x_gen x_id x_ext) x_target  $\rho$   $\sigma$ )
   ( $\rho_1$  (extend  $\sigma_1$  (x_target) ((linklet-instance (x cell) ... (x_ext cell_new))))))
   ; create a new variable and put a reference to it within the target
   (where (linklet-instance (x cell) ...) (lookup  $\sigma$  x_target))
   (where cell_new ,(variable-not-in (term ( $\rho$   $\sigma$  x ... cell ...)) (term cell_1)))
   (where ( $\rho_1$   $\sigma_1$ ) ((extend  $\rho$  (x_gen) (cell_new)) (extend  $\sigma$  (cell_new) (uninit)))))]

(define-metafunction Linklets
  instantiate-exports : c-exps x  $\rho$   $\sigma \rightarrow (\rho$   $\sigma)$ 
  [(instantiate-exports () x  $\rho$   $\sigma$ ) ( $\rho$   $\sigma$ )]
  [(instantiate-exports ((Export x_gen x_id x_ext) exp-obj ...) x_target  $\rho$   $\sigma$ )
   (instantiate-exports (exp-obj ...) x_target  $\rho_1$   $\sigma_1$ )
   (where ( $\rho_1$   $\sigma_1$ ) (process-one-export (Export x_gen x_id x_ext) x_target  $\rho$   $\sigma$ )))]

.....
;; util.rkt
.....

(define-metafunction Linklets
  substitute-one : x L-obj p-top  $\rightarrow$  p-top
  [(substitute-one x L-obj (instantiate-linklet x x_imp_inst ...))
   (instantiate-linklet L-obj x_imp_inst ...)]
  [(substitute-one x L-obj (instantiate-linklet x x_imp_inst ... #:target x_t))

```

```

(instantiate-linklet L-obj x_imp_inst ... #:target x_t)]
[(substitute-one x L-obj (let-inst x_1 I p-top))
 (let-inst x_1 I_s p-top_new)
 (where I_s (substitute-one x L-obj I))
 (where p-top_new (substitute-one x L-obj p-top))]
[(substitute-one x L-obj (let-inst x_1 LI p-top))
 (let-inst x_1 LI p-top_new)
 (where p-top_new (substitute-one x L-obj p-top))]
[(substitute-one x L-obj (seq p-top ...))
 (seq p-top_new ...)
 (where (p-top_new ...) ((substitute-one x L-obj p-top) ...))]
[(substitute-one x L-obj p-top) p-top]

```

```

(define-metafunction Linklets
  [(substitute-linklet x L-obj (p-top ...))
   (p-top_new ...)
   (where (p-top_new ...) ((substitute-one x L-obj p-top) ...))])

```

```

.....
;; main.rkt
.....

```

```

(define-metafunction Linklets
  ;; return
  [(run-prog ((program (use-linklets) (n _))  $\rho$   $\sigma$ )) n] ;; number
  [(run-prog ((program (use-linklets) (b _))  $\rho$   $\sigma$ )) b] ;; boolean
  [(run-prog ((program (use-linklets) ((void _))  $\rho$   $\sigma$ )) (void))] ;; void
  [(run-prog ((raises e)  $\rho$   $\sigma$ )) stuck] ;; stuck

  ;; problem in intermediate steps
  [(run-prog ((program (use-linklets (x L) ...) stuck)  $\rho$   $\sigma$ )) stuck]

  ;; reduce
  [(run-prog any_1)
   (run-prog any_again)]

```

```

    (where (any_again) ,(apply-reduction-relation --> $\beta$ p (term any_1))))]
[(run-prog any_1)
 (run-prog any_again)
 (where (any_again) ,(apply-reduction-relation --> $\beta$ r (term any_1))))]

[(run-prog any_1) stuck]

```

```

(define-metafunction Linklets
  ;eval-prog :Linklets-> v or closure or stuck or void
  [(eval-prog (program (use-linklets (x_L L) ...) p-top))
   (run-prog ((program (use-linklets (x_L L) ...) p-top) () ()))
   (where ((x_L L-obj) ...) ((x_L (compile-linklet L)) ...))
   (side-condition (and (term (check-free-varss L ...))
                        (term (no-exp/imp-duplicates L ...))
                        (term (no-export-rename-duplicates L ...))
                        (term (no-non-definable-variables L ...))
                        (term (no-duplicate-binding-namess L ...))
                        (term (linklet-refs-check-out
                              (p-top ...)
                              (x_L ...)
                              (get-defined-instance-ids (p-top ...) ()))))))
   [(eval-prog p) stuck])

```

APPENDIX D. SEMANTICS FOR THE CEK & STACKFUL HYBRID MODEL

$$\begin{aligned}
e ::= & x \mid v \mid (e \ e \ \dots) \mid (\mathbf{if} \ e \ e \ e) \mid (op \ e \ \dots) \\
& \mid (\mathbf{set!} \ x \ e) \mid (\mathbf{begin} \ e \ e \ \dots) \mid (\mathbf{lambda} \ (x_ \ \dots) \ e) \\
& \mid (\mathbf{let-values} \ (((x_) \ e) \ \dots) \ e) \mid (\mathbf{letrec-values} \ (((x_) \ e) \ \dots) \ e) \\
& \mid \mathit{internal-letrec} \mid (\mathbf{raises} \ e) \mid (\mathbf{raise-depth}) \mid (\mathbf{convert-stack} \ e) \\
& \mid \mathit{convert} \mid \mathit{stuck} \\
v ::= & n \mid b \mid c \mid (\mathbf{void}) \\
c ::= & (\mathbf{closure} \ x \ \dots \ e \ \rho) \\
n ::= & \mathit{number} \\
b ::= & \mathbf{true} \mid \mathbf{false} \\
x, \mathit{cell} ::= & \mathit{variable-not-otherwise-mentioned} \\
op ::= & \mathbf{add1} \mid + \mid * \mid < \mid \mathbf{sub1} \\
\rho ::= & ((x \ \mathit{any}) \ \dots) \\
\Sigma ::= & ((x \ \mathit{any}) \ \dots) \\
\mathit{internal-letrec} ::= & (\mathbf{letrec-values-cell-ready} \ (((x_) \ e) \ \dots) \ e) \\
\mathit{convert} ::= & (\mathbf{convert-to-stackful} \ e) \mid (\mathbf{convert-to-cek} \ e) \\
& \mid (\mathbf{convert-stack-to-heap} \ \rho \ \Sigma \ x) \\
\mathit{exception} ::= & (\mathbf{stack-depth-exn} \ n) \mid (\mathbf{convert-to-cek-exn} \ e \ \rho \ \Sigma) \\
\mathit{rc-result} ::= & v \mid \mathit{stuck} \\
\kappa ::= & (x \ \dots) \\
& \mid (\mathbf{if-}\kappa \ e \ e) \\
& \mid (\mathbf{arg-}\kappa \ (e \ \dots)) \\
& \mid (\mathbf{fun-}\kappa \ c \ (e \ \dots) \ (v \ \dots)) \\
& \mid (\mathbf{set-}\kappa \ x) \\
& \mid (\mathbf{seq-}\kappa \ e \ \dots) \\
& \mid (\mathbf{op-}\kappa \ op \ (v \ \dots) \ (e \ \dots) \ \rho) \\
& \mid (\mathbf{let-}\kappa \ (((x) \ e) \ \dots) \ (x \ \dots) \ (v \ \dots) \ e) \\
& \mid (\mathbf{letrec-}\kappa \ (((x) \ e) \ \dots) \ x \ e)
\end{aligned}$$

Figure D.1: Source Language for CEK & Stackful Hybrid Model

$[x, \rho, \Sigma, \kappa] \rightarrow_{cek} [\text{lookup}(\Sigma, \text{lookup}(\rho, x)), \rho, \Sigma, \kappa]$	[LOOKUP]
$[(\lambda(x \dots) e), \rho, \Sigma, \kappa] \rightarrow_{cek} [(\text{closure } x \dots e \rho), \rho, \Sigma, \kappa]$	[CLOSURE]
$[v_1, \rho, \Sigma, ((\text{op-}\kappa \text{ op } (v \dots) ()) \rho_{op}) \kappa \dots] \rightarrow_{cek} [\delta(\text{op } v \dots v_1), \rho_{op}, \Sigma, (\kappa \dots)]$	[OP-PLUG]
$[v, \rho, \Sigma, ((\text{if-}\kappa \text{ e}_1 \text{ e}_2) \kappa \dots)] \rightarrow_{cek} [e, \rho, \Sigma, (\kappa \dots)] \text{ where } e = (\text{if } (\text{equal? } v \text{ false}) \text{ e}_2 \text{ e}_1)$	[IF-TRUE-PLUG]
$[v, \rho, \Sigma, ((\text{set-}\kappa \text{ x}) \kappa \dots)] \rightarrow_{cek} [(void), \rho, \text{overwrite}(\Sigma, x, v), (\kappa \dots)]$	[SET-PLUG]
$[v, \rho, \Sigma, ((\text{let-}\kappa ()) (x_{rhs} \dots) (v_{rhs} \dots) e_{body}) \kappa \dots] \rightarrow_{cek} [e_{body}, \text{extend}(\rho, (x_{rhs} \dots), (cell_{addr} \dots)),$ $\text{extend}(\Sigma, (cell_{addr} \dots), (v_{rhs} \dots)), (\kappa \dots)]$ where $(cell_{addr} \dots) = \text{variables-not-in}(e_{body}, (x_{rhs} \dots))$	[LET-PLUG]
$[v, \rho, \Sigma, ((\text{letrec-}\kappa ()) x_{prev} e_{body}) \kappa \dots] \rightarrow_{cek} [e_{body}, \rho, \text{extend}(\Sigma, (\text{lookup}(\rho, x_{prev})), (v)), (\kappa \dots)]$	[LETREC-PLUG]
$[(\text{letrec-values } ()) e_{body}], \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e_{body}, \rho, \Sigma, (\kappa \dots)]$	[LETREC-NO-RHS]
$[v_1, \rho, \Sigma, ((\text{op-}\kappa \text{ op } (v \dots) (e_1 e_{\dots})) \rho_{op}) \kappa \dots] \rightarrow_{cek} [e_1, \rho_{op}, \Sigma, ((\text{op-}\kappa \text{ op } (v \dots) v_1) (e_{\dots}) \rho_{op}) \kappa \dots]$	[OP-SWITCH]
$[(\text{op } e_1 e_{\dots}), \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e_1, \rho, \Sigma, ((\text{op-}\kappa \text{ op } ()) (e_{\dots}) \rho) \kappa \dots]$	[OP-PUSH]
$[(\text{if } e_{test} \text{ e}_1 \text{ e}_2), \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e_{test}, \rho, \Sigma, ((\text{if-}\kappa \text{ e}_1 \text{ e}_2) \kappa \dots)]$	[IF-PUSH]
$[(\text{set! } x \text{ e}), \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e, \rho, \Sigma, ((\text{set-}\kappa(\text{lookup}(\rho, x))) \kappa \dots)]$	[SET-PUSH]
$[(\text{begin } e_1 e_{\dots}), \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e_1, \rho, \Sigma, ((\text{seq-}\kappa \text{ e}_{\dots}) \kappa \dots)]$	[BEGIN-PUSH]
$[v, \rho, \Sigma, ((\text{seq-}\kappa \text{ e}_1 e_{\dots}) \kappa \dots)] \rightarrow_{cek} [e_1, \rho, \Sigma, ((\text{seq-}\kappa \text{ e}_{\dots}) \kappa \dots)]$	[BEGIN-SWITCH]
$[v, \rho, \Sigma, ((\text{seq-}\kappa) \kappa \dots)] \rightarrow_{cek} [v, \rho, \Sigma, (\kappa \dots)]$	[BEGIN-PLUG]
$[v_{rhs}, \rho, \Sigma, ((\text{let-}\kappa(((x_1) e_{rhs_next}))((x_2) e_2) \dots)$ $(x \dots) (v \dots) e_{body}) \kappa \dots] \rightarrow_{cek} [e_{rhs_next}, \rho, \Sigma, ((\text{let-}\kappa(((x_2) e_2) \dots)$ $(x_1 x \dots) (v_{rhs} v \dots) e_{body}) \kappa \dots)]$	[LET-RHS-SWITCH]
$[(\text{let-values } (((x_{rhs}) e_{rhs}))((x_2) e_2) \dots) e_{body}], \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e_{rhs}, \rho, \Sigma, ((\text{let-}\kappa(((x_2) e_2) \dots)(x_{rhs})()) e_{body}) \kappa \dots]$	[LET-PUSH]
$[v_{rhs}, \rho, \Sigma, ((\text{letrec-}\kappa$ $((x_{rhs_next}) e_{rhs_next})((x_{rhs_rest}) e_{rhs_rest}) \dots)$ $x_{prev} e_{body}) \kappa \dots] \rightarrow_{cek} [e_{rhs_next}, \rho, \text{extend}(\Sigma, (\text{lookup}(\rho, x_{prev})), (v_{rhs})),$ $((\text{letrec-}\kappa(((x_{rhs_rest}) e_{rhs_rest}) \dots) x_{rhs_next} e_{body}) \kappa \dots)]$	[LETREC-RHS-SWITCH]
$[(\text{letrec-values } (((x_{rhs}) e_{rhs}))((x_{rhs_next}) e_{rhs_next}) \dots) e_{body}],$ $\rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [e_{rhs}, \text{extend}(\rho, (x_{rhs} x_{rhs_next} \dots), (cell_{addr} \dots)), \Sigma,$ $((\text{letrec-}\kappa(((x_{rhs_next}) e_{rhs_next}) \dots) x_{rhs} e_{body}) \kappa \dots)]$ where $(cell_{addr} \dots) = \text{variables-not-in}(e_{body}, (x_{rhs} x_{rhs_next} \dots))$	[LETREC-PUSH]
$[(\text{erator } e_{rands} \dots), \rho, \Sigma, (\kappa \dots)] \rightarrow_{cek} [erator, \rho, \Sigma, ((\text{arg-}\kappa(e_{rands} \dots)) \kappa \dots)]$	[APP-RATOR-PUSH]
$[(\text{closure } x \dots e_{body} \rho_{clo}), \rho, \Sigma, ((\text{arg-}\kappa()) \kappa \dots)] \rightarrow_{cek} [e_{body}, \rho_{clo}, \Sigma, (\kappa \dots)]$	[APP-NO-RAND-PLUG]
$[v_{clo}, \rho, \Sigma, ((\text{arg-}\kappa(e_{rand_1} e_{rands} \dots)) \kappa \dots)] \rightarrow_{cek} [e_{rand_1}, \rho, \Sigma, ((\text{fun-}\kappa v_{clo}(e_{rands} \dots)()) \kappa \dots)]$	[APP-RAND-PUSH]
$[v_{rand}, \rho, \Sigma, ((\text{fun-}\kappa v_{clo}(e_{rand_1} e_{rands} \dots)(v \dots)) \kappa \dots)] \rightarrow_{cek} [e_{rand_1}, \rho, \Sigma, ((\text{fun-}\kappa v_{clo}(e_{rands} \dots)(v \dots v_{rand})) \kappa \dots)]$	[APP-RAND-SWITCH]
$[v_{rand}, \rho, \Sigma, ((\text{fun-}\kappa(\text{closure } x \dots e_{body} \rho_{clo})) (v \dots)) \kappa \dots] \rightarrow_{cek} [e_{body}, \text{extend}(\rho_{clo}, (x \dots), (cell_{addr} \dots)),$ $\text{extend}(\Sigma, (cell_{addr} \dots), (v \dots v_{rand})), (\kappa \dots)]$ where $(cell_{addr} \dots) = \text{variables-not-in}(e_{body}, (x \dots))$	[APP-PLUG]

Figure D.2: CEK Reduction Relation


```

(define-metafunction RC
  ; (expr env store stack-depth) -> (result store stack-depth)
  interpret-stack : e  $\rho$   $\Sigma$  n -> (rc-result  $\Sigma$  n) or exception or convert
  [(interpret-stack (raises e)  $\rho$   $\Sigma$  n) (stuck  $\Sigma$  n)] ; for intermediate errors
  [(interpret-stack (raise-depth)  $\rho$   $\Sigma$  n) (stack-depth-exn n)]
  ; stack overflow
  #,[(interpret-stack e  $\rho$   $\Sigma$  n)
    (rc-result  $\Sigma_{\text{new}}$  n)
    (side-condition
      (and (not (redex-match? RC convert (term e)))
            (not (redex-match? RC x (term e)))
            (not (redex-match? RC v (term e)))
            (>= (term n) 10)
            (begin (printf "overflow converting to cek for ~a -- n : ~a\n" (term e) (term n)) #t)))]
    (where (rc-result  $\Sigma_{\text{new}}$ ) (run-cek (e  $\rho$   $\Sigma$  ()))))]
  [(interpret-stack e  $\rho$   $\Sigma$  n)
    (convert-stack-to-heap e  $\rho$   $\Sigma$  ())
    (side-condition
      (and (not (redex-match? RC convert (term e)))
            (not (redex-match? RC x (term e)))
            (not (redex-match? RC v (term e)))
            (>= (term n) 10)
            (begin (printf "overflow converting to cek for ~a -- n : ~a\n" (term e) (term n)) #t)))]
    ; convert to cek (for a single expression)
    [(interpret-stack (convert-to-cek e)  $\rho$   $\Sigma$  n)
      (rc-result  $\Sigma_{\text{new}}$  n)
      (where (rc-result  $\Sigma_{\text{new}}$ ) (run-cek (e  $\rho$   $\Sigma$  ()))))]
  ; convert to heap
  [(interpret-stack (convert-stack e)  $\rho$   $\Sigma$  n) (convert-stack-to-heap e  $\rho$   $\Sigma$  ())]
  ; if
  [(interpret-stack (if e_test e_1 e_2)  $\rho$   $\Sigma$  n)
    (convert-stack-to-heap e_ast  $\rho_{\text{ast}}$   $\Sigma_{\text{ast}}$  ( $\kappa$  ... (if- $\kappa$  e_1 e_2)))
    (where (convert-stack-to-heap e_ast  $\rho_{\text{ast}}$   $\Sigma_{\text{ast}}$  ( $\kappa$  ...)))]

```

```

      (interpret-stack e_test  $\rho$   $\Sigma$  ,(add1 (term n))))]
;op
[(interpret-stack (op v ... e_1 e ...)  $\rho$   $\Sigma$  n)
 (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ... (op- $\kappa$  op (v ...) (e ...)  $\rho$ )))
 (where (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ...))
        (interpret-stack e_1  $\rho$   $\Sigma$  ,(add1 (term n))))]
;set!
[(interpret-stack (set! x e)  $\rho$   $\Sigma$  n)
 (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ... (set- $\kappa$  (lookup  $\rho$  x))))
 (where (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ...))
        (interpret-stack e  $\rho$   $\Sigma$  ,(add1 (term n))))]
;begin
[(interpret-stack (begin v ... e_1 e_2 e ...)  $\rho$   $\Sigma$  n)
 (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ... (seq- $\kappa$  e_2 e ...)))
 (where (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ...))
        (interpret-stack e_1  $\rho$   $\Sigma$  ,(add1 (term n))))]
;let-values
[(interpret-stack (let-values (((x_1) v_1) ... ((x) e) ((x_r) e_r) ...) e_body)  $\rho$   $\Sigma$  n)
 (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast
      ( $\kappa$  ... (let- $\kappa$  (((x_r) e_r) ...)
                     (x_1 ... x) (v_1 ...) e_body)))
 (where (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ...))
        (interpret-stack e  $\rho$   $\Sigma$  ,(add1 (term n))))]
;letrec-values
[(interpret-stack (letrec-values-cell-ready
                  (((x) e) ((x_r) e_r) ...) e_body)  $\rho$   $\Sigma$  n)
 (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast
      ( $\kappa$  ...
        (letrec- $\kappa$  (((x_r) e_r) ...)
                    x
                    e_body)))
 (where (convert-stack-to-heap e_ast  $\rho$ _ast  $\Sigma$ _ast ( $\kappa$  ...))
        (interpret-stack e  $\rho$   $\Sigma$  ,(add1 (term n))))]
;app
[(interpret-stack ((closure x ... e_body  $\rho$ _closure) v_args ... e_arg_1 e_args ...)  $\rho$   $\Sigma$  n)

```

```

(convert-stack-to-heap e_ast ρ_ast Σ_ast
  (κ ... (fun-κ (closure x ... e_body ρ_closure) (e_args ...) (v_args ...))))
(when (convert-stack-to-heap e_ast ρ_ast Σ_ast (κ ...))
  (interpret-stack e_arg_1 ρ Σ ,(add1 (term n))))]
[(interpret-stack (e_f e_args ...) ρ Σ n)
  (convert-stack-to-heap e_ast ρ_ast Σ_ast (κ ... (arg-κ (e_args ...))))]
(when (convert-stack-to-heap e_ast ρ_ast Σ_ast (κ ...))
  (interpret-stack e_f ρ Σ ,(add1 (term n))))]

; value
[(interpret-stack rc-result ρ Σ n) (rc-result Σ n)]

; id lookup
[(interpret-stack x ρ Σ n) ((lookup Σ (lookup ρ x)) Σ n)]

; lambda
[(interpret-stack (lambda (x ...) e) ρ Σ n) ((closure x ... e ρ) Σ n)]

; set!
[(interpret-stack (set! x e) ρ Σ n)
  ((void) (overwrite Σ_1 (lookup ρ x) v) n)
  (when (v Σ_1 n_1) (interpret-stack e ρ Σ ,(add1 (term n))))]

; op
[(interpret-stack (op v ...) ρ Σ n) ((δ (op v ...)) Σ n)]

[(interpret-stack (op v ... e_1 e ...) ρ Σ n)
  (interpret-stack (op v ... v_1 e ...) ρ Σ_1 n)
  (side-condition (not (redex-match? RC v (term e_1))))]
(when (v_1 Σ_1 n_1) (interpret-stack e_1 ρ Σ ,(add1 (term n))))]

[(interpret-stack (op v ... e_1 e ...) ρ Σ n)
  (stuck Σ_1 n_1)
  (when (stuck Σ_1 n_1) (interpret-stack e_1 ρ Σ ,(add1 (term n))))]
[(interpret-stack (op v ... e_1 e ...) ρ Σ n)
  exception
  (when exception (interpret-stack e_1 ρ Σ ,(add1 (term n))))]

; begin

```

```

[(interpret-stack (begin v ... e_1 e_2 e ...)  $\rho$   $\Sigma$  n)
 (interpret-stack (begin v ... v_1 e_2 e ...)  $\rho$   $\Sigma_{-1}$  n)
 (side-condition (not (redex-match? RC v (term e_1))))
 (where (v_1  $\Sigma_{-1}$  n_1) (interpret-stack e_1  $\rho$   $\Sigma$  ,(add1 (term n)))))
[(interpret-stack (begin v ... e_1 e_2 e ...)  $\rho$   $\Sigma$  n)
 exception
 (where exception (interpret-stack e_1  $\rho$   $\Sigma$  ,(add1 (term n)))))
[(interpret-stack (begin v ... e)  $\rho$   $\Sigma$  n) ; tail
 (interpret-stack e  $\rho$   $\Sigma$  n)]
; if
[(interpret-stack (if e_test e_1 e_2)  $\rho$   $\Sigma$  n)
 ,(if (equal? (term v_1) (term false))
 (term (interpret-stack e_2  $\rho$   $\Sigma_{-1}$  n)) ;; tail
 (term (interpret-stack e_1  $\rho$   $\Sigma_{-1}$  n)))
 (where (v_1  $\Sigma_{-1}$  n_1) (interpret-stack e_test  $\rho$   $\Sigma$  n))]
; let-values
[(interpret-stack (let-values (((x) v) ...) e_body)  $\rho$   $\Sigma$  n)
 (interpret-stack e_body (extend  $\rho$  (x ...) (cell_addr ...)) (extend  $\Sigma$  (cell_addr ...) (v ...)) n)
 (where (cell_addr ...) ,(variables-not-in (term e_body) (term (x ...))))) ; tail
[(interpret-stack (let-values (((x_1) v_1) ... ((x) e) ((x_r) e_r) ...) e_body)  $\rho$   $\Sigma$  n)
 (interpret-stack (let-values (((x_1) v_1) ... ((x) v) ((x_r) e_r) ...) e_body)  $\rho$   $\Sigma_{-1}$  n)
 (side-condition (not (redex-match? RC v (term e))))
 (where (v  $\Sigma_{-1}$  n_1) (interpret-stack e  $\rho$   $\Sigma$  ,(add1 (term n)))))
[(interpret-stack (let-values (((x_1) v_1) ... ((x) e) ((x_r) e_r) ...) e_body)  $\rho$   $\Sigma$  n)
 exception
 (where exception (interpret-stack e  $\rho$   $\Sigma$  ,(add1 (term n)))))
; letrec-values
[(interpret-stack (letrec-values (((x) e) ...) e_body)  $\rho$   $\Sigma$  n)
 (interpret-stack (letrec-values-cell-ready (((x) e) ...) e_body)
 (extend  $\rho$  (x ...) (cell_addr ...))  $\Sigma$  n)
 (where (cell_addr ...) ,(variables-not-in (term (e_body x ...)) (term (x ...)))))
[(interpret-stack (letrec-values-cell-ready () e_body)  $\rho$   $\Sigma$  n)
 (interpret-stack e_body  $\rho$   $\Sigma$  n)]

```

```

[(interpret-stack (letrec-values-cell-ready (((x_rhs) e_rhs) ((x) e) ...) e_body)  $\rho$   $\Sigma$  n)
 (interpret-stack (letrec-values-cell-ready (((x) e) ...) e_body)
   $\rho$  (extend  $\Sigma_1$  ((lookup  $\rho$  x_rhs)) (v_rhs)) n)
;
v-- don't need to extend, the cell is already there
(when (v_rhs  $\Sigma_1$  n_1) (interpret-stack e_rhs  $\rho$   $\Sigma$  ,(add1 (term n)))))]

[(interpret-stack (letrec-values-cell-ready (((x) e) ((x_r) e_r) ...) e_body)  $\rho$   $\Sigma$  n)
 exception
 (when cell_addr (lookup  $\rho$  x))
 (when exception (interpret-stack e  $\rho$   $\Sigma$  ,(add1 (term n)))))]

;app
[(interpret-stack (x_func e ...)  $\rho$   $\Sigma$  n)
 (interpret-stack (v_func e ...)  $\rho$   $\Sigma$  n)
 (when v_func (lookup  $\Sigma$  (lookup  $\rho$  x_func)))]

[(interpret-stack ((closure x ... e_body  $\rho$ _closure) v_args ...)  $\rho$   $\Sigma$  n)
 (interpret-stack e_body (extend  $\rho$ _closure (x ...) (cell_addr ...)) (extend  $\Sigma$  (cell_addr ...) (v_args ...)
 ) n)
 (when (cell_addr ...) ,(variables-not-in (term e_body) (term (x ...)))))]

[(interpret-stack ((closure x ... e_body  $\rho$ _closure) v_args ... e_arg_1 e_args ...)  $\rho$   $\Sigma$  n)
 (interpret-stack ((closure x ... e_body  $\rho$ _closure) v_args ... v_arg_1 e_args ...)  $\rho$   $\Sigma_1$  n)
 (side-condition (not (redex-match? RC v (term e_arg_1))))
 (when (v_arg_1  $\Sigma_1$  n_1) (interpret-stack e_arg_1  $\rho$   $\Sigma$  ,(add1 (term n)))))]

[(interpret-stack ((closure x ... e_body  $\rho$ _closure) v_args ... e_arg_1 e_args ...)  $\rho$   $\Sigma$  n)
 exception
 (when exception (interpret-stack e_arg_1  $\rho$   $\Sigma$  ,(add1 (term n)))))
[(interpret-stack (e_f e_args ...)  $\rho$   $\Sigma$  n)
 (interpret-stack (v_func e_args ...)  $\rho$   $\Sigma_1$  n)
 (when (v_func  $\Sigma_1$  n_1) (interpret-stack e_f  $\rho$   $\Sigma$  ,(add1 (term n)))))]
```

```

[(interpret-stack (e_f e_args ...)  $\rho$   $\Sigma$  n)
 exception
 (where exception (interpret-stack e_f  $\rho$   $\Sigma$  ,(add1 (term n))))]
)

```

```

.....
;; CEK Evaluator
.....

```

```

(define-metafunction RC
  eval-cek : e -> rc-result or exception
  [(eval-cek e) rc-result
   (where (rc-result  $\Sigma$ ) (run-cek (e ()) () ())))]
  [(eval-cek e) exception
   (where exception (run-cek (e ()) () ())))]
)

```

```

(define-metafunction RC
  run-cek : (e  $\rho$   $\Sigma$   $\kappa$ ) -> (rc-result  $\Sigma$ ) or exception
  [(run-cek (rc-result  $\rho$   $\Sigma$  ())) (rc-result  $\Sigma$ )]
  [(run-cek ((convert-to-stackful e)  $\rho$   $\Sigma$   $\kappa$ ))
   (run-cek (e_again  $\rho$   $\Sigma$ _again  $\kappa$ ))
   (where (e_again  $\Sigma$ _again)
    (dynamic-eval-stackful e  $\rho$   $\Sigma$ ))]
  [(run-cek any_1)
   (run-cek (e_again  $\rho$ _again  $\Sigma$ _again  $\kappa$ _again))
   (where ((e_again  $\rho$ _again  $\Sigma$ _again  $\kappa$ _again))
    ,(apply-reduction-relation -->cek (term any_1)))]
  [(run-cek any) (stuck ()))]
)

```

REFERENCES

- [1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, July 25, 1996. ISBN: 978-0-262-51087-5 978-0-262-31091-8. URL: <https://library.oapen.org/handle/20.500.12657/26092>.
- [2] Maxime Chevalier-Boisvert et al. “Bootstrapping a Self-Hosted Research Virtual Machine for JavaScript: An Experience Report”. In: *Proceedings of the 7th Symposium on Dynamic Languages*. DLS ’11. New York, NY, USA: Association for Computing Machinery, Oct. 24, 2011, pp. 61–72. ISBN: 978-1-4503-0939-4. URL: <https://dl.acm.org/doi/10.1145/2047849.2047858>.
- [3] Carl Friedrich Bolz. “Meta-Tracing Just-in-Time Compilation for RPython”. In: ().
- [4] Carl Friedrich Bolz et al. “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* (Genova, Italy). IC00OLPS ’09. New York, NY, USA: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3. URL: <http://doi.acm.org/10.1145/1565824.1565827>.
- [5] Matthew Flatt et al. “Rebuilding Racket on Chez Scheme (Experience Report)”. In: *Proc. ACM Program. Lang.* 3 (ICFP July 2019), 78:1–78:15. URL: <http://doi.acm.org/10.1145/3341642>.
- [6] Carl Friedrich Bolz et al. “Meta-Tracing Makes a Fast Racket”. In: (2014), p. 7.

- [7] Spenser Bauman et al. “Pycket: A Tracing JIT for a Functional Language”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada). ICFP 2015. New York, NY, USA: ACM, 2015, pp. 22–34. ISBN: 978-1-4503-3669-7. URL: <http://doi.acm.org/10.1145/2784731.2784740>.
- [8] Spenser Bauman et al. “Sound Gradual Typing: Only Mostly Dead”. In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 1–24. URL: <http://dl.acm.org/citation.cfm?doid=3152284.3133878>.
- [9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Dynamo: A Transparent Dynamic Optimization System”. In: (), p. 12.
- [10] M. Arnold et al. “A Survey of Adaptive Optimization in Virtual Machines”. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 449–466.
- [11] John Aycock. “A Brief History of Just-in-time”. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. URL: <http://doi.acm.org/10.1145/857076.857077>.
- [12] Armin Rigo and Samuele Pedroni. “PyPy’s Approach to Virtual Machine Construction”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA). OOPSLA ’06. New York, NY, USA: ACM, 2006, pp. 944–953. ISBN: 978-1-59593-491-8. URL: <http://doi.acm.org/10.1145/1176617.1176753>.
- [13] Davide Ancona et al. “RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages”. In: *Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, Quebec, Canada). DLS ’07. New York, NY, USA: ACM, 2007, pp. 53–64. ISBN: 978-1-59593-868-8. URL: <http://doi.acm.org/10.1145/1297081.1297091>.

- [14] Camillo Bruni and Toon Verwaest. “PyGirl: Generating Whole-System VMs from High-Level Models Using PyPy”. In: vol. 33. June 29, 2009, pp. 328–347.
- [15] *The Architecture of Open Source Applications (Volume 2): PyPy*. URL: <https://www.osabook.org/en/pypy.html> (visited on 11/12/2019).
- [16] Sam Tobin-Hochstadt et al. “Languages as Libraries”. In: (), p. 10.
- [17] Matthias Felleisen and Daniel P. Friedman. “Control Operators, the SECD-machine, and the λ -Calculus”. In: *Formal Description of Programming Concepts*. 1987.
- [18] Matthew Flatt. “Composable and Compilable Macros: You Want It When?” In: *SIGPLAN Not.* 37.9 (Sept. 17, 2002), pp. 72–83. URL: <https://dl.acm.org/doi/10.1145/583852.581486>.
- [19] Julia L. Lawall and Olivier Danvy. “Separating Stages in the Continuation-Passing Style Transformation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’93. New York, NY, USA: Association for Computing Machinery, Mar. 1, 1993, pp. 124–136. ISBN: 978-0-89791-560-1. URL: <https://dl.acm.org/doi/10.1145/158511.158613>.
- [20] Cormac Flanagan et al. “The Essence of Compiling with Continuations”. In: *SIGPLAN Not.* 28.6 (June 1, 1993), pp. 237–247. URL: <https://dl.acm.org/doi/10.1145/173262.155113>.
- [21] *Contracts for Higher-Order Functions | Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. URL: <https://dl-acm-org.proxyiub.uits.iu.edu/doi/10.1145/581478.581484> (visited on 06/23/2025).

- [22] Andreas Gal et al. “Trace-Based Just-in-Time Type Specialization for Dynamic Languages”. In: *SIGPLAN Not.* 44.6 (June 15, 2009), pp. 465–478. URL: <https://dl.acm.org/doi/10.1145/1543135.1542528>.
- [23] Carl Friedrich Bolz et al. “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* (Lancaster, United Kingdom). ICPOOLPS ’11. New York, NY, USA: ACM, 2011, 9:1–9:8. ISBN: 978-1-4503-0894-6. URL: <http://doi.acm.org/10.1145/2069172.2069181>.
- [24] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. “Loop-Aware Optimizations in PyPy’s Tracing JIT”. In: *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA). DLS ’12. New York, NY, USA: ACM, 2012, pp. 63–72. ISBN: 978-1-4503-1564-7. URL: <http://doi.acm.org/10.1145/2384577.2384586>.
- [25] *Incremental Dynamic Code Generation with Trace Trees | Request PDF*. ResearchGate. URL: https://www.researchgate.net/publication/213877755_Incremental_Dynamic_Code_Generation_with_Trace_Trees (visited on 06/24/2025).
- [26] David A. Terei and Manuel M.T. Chakravarty. “An LLVM Backend for GHC”. In: *SIGPLAN Not.* 45.11 (Sept. 30, 2010), pp. 109–120. URL: <https://dl.acm.org/doi/10.1145/2088456.1863538>.
- [27] Christian A. Schafmeister and Alex Wood. “Clasp Common Lisp Implementation and Optimization”. In: *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*. ELS2018. Marbella, Spain: European Lisp Scientific Activities Association, Apr. 16, 2018, pp. 59–64. ISBN: 978-2-9557474-2-1.

- [28] Fred Chow. “Intermediate Representation”. In: *Commun. ACM* 56.12 (Dec. 1, 2013), pp. 57–62. URL: <https://dl.acm.org/doi/10.1145/2534706.2534720>.
- [29] Jack J. Garzella et al. “Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification”. In: *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Jan. 16, 2020, pp. 90–111. ISBN: 978-3-030-39321-2. URL: https://doi.org/10.1007/978-3-030-39322-9_5.
- [30] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st ed. The MIT Press, July 2009. 528 pp. ISBN: 978-0-262-06275-6.
- [31] Casey Klein, Matthew Flatt, and Robert Bruce Findler. “Random Testing for Higher-Order, Stateful Programs”. In: *SIGPLAN Not.* 45.10 (Oct. 17, 2010), pp. 555–566. URL: <https://dl.acm.org/doi/10.1145/1932682.1869505>.
- [32] Dan Ingalls et al. “Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself”. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’97. New York, NY, USA: Association for Computing Machinery, Oct. 9, 1997, pp. 318–326. ISBN: 978-0-89791-908-1. URL: <https://dl.acm.org/doi/10.1145/263698.263754>.
- [33] Alceste Scalas, Giovanni Casu, and Piero Pili. “High-Performance Technical Computing with Erlang”. In: *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG ’08. New York, NY, USA: Association for Computing Machinery, Sept. 27, 2008,

- pp. 49–60. ISBN: 978-1-60558-065-4. URL: <https://dl.acm.org/doi/10.1145/1411273.1411281>.
- [34] Linda G. DeMichiel and Richard P. Gabriel. “The Common Lisp Object System: An Overview”. In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP ’87. Berlin, Heidelberg: Springer-Verlag, June 15, 1987, pp. 151–170. ISBN: 978-3-540-18353-2.
- [35] Christopher T. Haynes and Daniel P. Friedman. “Engines Build Process Abstractions”. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP ’84. New York, NY, USA: Association for Computing Machinery, Aug. 6, 1984, pp. 18–24. ISBN: 978-0-89791-142-9. URL: <https://dl.acm.org/doi/10.1145/800055.802018>.
- [36] Matthew Flatt. “Creating Languages in Racket”. In: *Commun. ACM* 55.1 (Jan. 1, 2012), pp. 48–56. URL: <https://dl.acm.org/doi/10.1145/2063176.2063195>.
- [37] Andreas Gal, Christian W. Probst, and Michael Franz. “HotpathVM: An Effective JIT Compiler for Resource-constrained Devices”. In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada). VEE ’06. New York, NY, USA: ACM, 2006, pp. 144–153. ISBN: 978-1-59593-332-4. URL: <http://doi.acm.org/10.1145/1134760.1134780>.
- [38] Carl Friedrich Bolz et al. “Allocation Removal by Partial Evaluation in a Tracing JIT”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Austin, Texas, USA). PEPM ’11. New York, NY, USA: ACM, 2011,

- pp. 43–52. ISBN: 978-1-4503-0485-6. URL: <http://doi.acm.org/10.1145/1929501.1929508>.
- [39] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*. USA: Cambridge University Press, May 2013. ISBN: 978-0-511-80284-3.
 - [40] David Schneider and Carl Friedrich Bolz. “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*. SPLASH ’12: Conference on Systems, Programming, and Applications: Software for Humanity. Tucson Arizona USA: ACM, Oct. 21, 2012, pp. 3–12. ISBN: 978-1-4503-1633-0. URL: <https://dl.acm.org/doi/10.1145/2414740.2414743>.
 - [41] *You Lose More When Slow than You Gain When Fast – Nicholas Nethercote*. URL: <https://blog.mozilla.org/nnethercote/2011/05/31/you-lose-more-when-slow-than-you-gain-when-fast/> (visited on 06/12/2025).
 - [42] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st ed. Chapman & Hall/CRC, July 2011. ISBN: 978-1-4200-8279-1.
 - [43] *LuaJIT Language Toolkit*. URL: <https://github.com/franko/luajit-lang-toolkit>.
 - [44] Andrew W. Appel. *Compiling with Continuations*. USA: Cambridge University Press, Jan. 2007. 272 pp. ISBN: 978-0-521-03311-4.
 - [45] Thomas Würthinger et al. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Program-*

- ming & Software - Onward! '13*. The 2013 ACM International Symposium. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 187–204. ISBN: 978-1-4503-2472-4. URL: <http://dl.acm.org/citation.cfm?doid=2509578.2509581>.
- [46] Thomas Würthinger et al. “Practical Partial Evaluation for High-Performance Dynamic Language Runtimes”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '17: ACM SIGPLAN Conference on Programming Language Design and Implementation. Barcelona Spain: ACM, June 14, 2017, pp. 662–676. ISBN: 978-1-4503-4988-8. URL: <https://dl.acm.org/doi/10.1145/3062341.3062381>.
- [47] Andreas Gal et al. “Trace-Based Just-in-time Type Specialization for Dynamic Languages”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland). PLDI '09. New York, NY, USA: ACM, 2009, pp. 465–478. ISBN: 978-1-60558-392-1. URL: <http://doi.acm.org/10.1145/1542476.1542528>.
- [48] Stefan Marr and Stéphane Ducasse. “Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015*. The 2015 ACM SIGPLAN International Conference. Pittsburgh, PA, USA: ACM Press, 2015, pp. 821–839. ISBN: 978-1-4503-3689-5. URL: <http://dl.acm.org/citation.cfm?doid=2814270.2814275>.
- [49] Michael Bebenita et al. “SPUR: A Trace-Based JIT Compiler for CIL”. In: (), p. 18.

- [50] Jim S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. USA: Addison-Wesley Longman Publishing Co., Inc., Sept. 2003. 928 pp. ISBN: 978-0-321-15493-4.
- [51] Mason Chang et al. “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*. VEE '09. New York, NY, USA: Association for Computing Machinery, Mar. 11, 2009, pp. 71–80. ISBN: 978-1-60558-375-4. URL: <https://dl.acm.org/doi/10.1145/1508293.1508304>.
- [52] Laurence Tratt. “Compile-Time Meta-programming in a Dynamically Typed OO Language”. In: *Proceedings of the 2005 Symposium on Dynamic Languages* (San Diego, California). DLS '05. New York, NY, USA: ACM, 2005, pp. 49–63. URL: <http://doi.acm.org/10.1145/1146841.1146846>.
- [53] Thomas Schilling. “Trace-Based Just-in-time Compilation for Lazy Functional Programming Languages”. In: (), p. 213.
- [54] Wiik Thomassen. “Trace-Based Just-Intime Compiler for Haskell with RPython”. In: 2013.
- [55] Carl Friedrich Bolz and Laurence Tratt. “The Impact of Meta-Tracing on VM Design and Implementation”. In: *Science of Computer Programming*. Special Issue on Advances in Dynamic Languages 98 (Feb. 1, 2015), pp. 408–421. URL: <http://www.sciencedirect.com/science/article/pii/S0167642313000269>.
- [56] Carl Friedrich Bolz and Armin Rigo. “How to Not Write Virtual Machines for Dynamic Languages”. In: (), p. 11.

- [57] Mark N. Wegman and F. Kenneth Zadeck. “Constant Propagation with Conditional Branches”. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. URL: <http://doi.acm.org/10.1145/103135.103136>.
- [58] Kevin Casey, M. Anton Ertl, and David Gregg. “Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters”. In: *ACM Trans. Program. Lang. Syst.* 29.6 (Oct. 1, 2007), 37–es. URL: <https://dl.acm.org/doi/10.1145/1286821.1286828>.
- [59] Andrzej Filinski. “A Semantic Account of Type-Directed Partial Evaluation”. In: *Proceedings of the International Conference PPDP’99 on Principles and Practice of Declarative Programming*. PPDP ’99. Berlin, Heidelberg: Springer-Verlag, Sept. 29, 1999, pp. 378–395. ISBN: 978-3-540-66540-3.
- [60] Rui Ge and Ronald Garcia. “Refining Semantics for Multi-Stage Programming”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2017. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2017, pp. 2–14. ISBN: 978-1-4503-5524-7. URL: <https://dl.acm.org/doi/10.1145/3136040.3136047>.
- [61] Nada Amin and Tiark Rompf. “Collapsing Towers of Interpreters”. In: *Proc. ACM Program. Lang.* 2 (POPL Dec. 2017), 52:1–52:33. URL: <http://doi.acm.org/10.1145/3158140>.
- [62] Kazuaki Ishizaki et al. “Adding Dynamically-typed Language Support to a Statically-typed Language Compiler: Performance Evaluation, Analysis, and Tradeoffs”. In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environ-*

- ments* (London, England, UK). VEE '12. New York, NY, USA: ACM, 2012, pp. 169–180. ISBN: 978-1-4503-1176-2. URL: <http://doi.acm.org/10.1145/2151024.2151047>.
- [63] Jose Castanos et al. “On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages”. In: *SIGPLAN Not.* 47.10 (Oct. 19, 2012), pp. 195–212. URL: <https://dl.acm.org/doi/10.1145/2398857.2384631>.
- [64] C. Chambers, D. Ungar, and E. Lee. “An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes”. In: *SIGPLAN Not.* 24.10 (Sept. 1, 1989), pp. 49–70. URL: <https://dl.acm.org/doi/10.1145/74878.74884>.
- [65] Andrew W. Appel. *Compiling with Continuations*. USA: Cambridge University Press, 1992. 262 pp. ISBN: 978-0-521-41695-5.
- [66] Luke Maurer et al. “Compiling Without Continuations”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain). PLDI 2017. New York, NY, USA: ACM, 2017, pp. 482–494. ISBN: 978-1-4503-4988-8. URL: <http://doi.acm.org/10.1145/3062341.3062380>.
- [67] Youyou Cong et al. “Compiling with Continuations, or Without? Whatever.” In: *Proc. ACM Program. Lang.* 3 (ICFP July 2019), 79:1–79:28. URL: <http://doi.acm.org/10.1145/3341643>.
- [68] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. “Representing Control in the Presence of First-class Continuations”. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA).

PLDI '90. New York, NY, USA: ACM, 1990, pp. 66–77. ISBN: 978-0-89791-364-5.

URL: <http://doi.acm.org/10.1145/93542.93554>.

VITA

Vita may be provided by doctoral students only. The length of the vita is preferably one page. It may include the place of birth and should be written in third person. This vita is similar to the author biography found on book jackets.