Thesis Proposal: Towards Rapid and Efficient Implementation for Self-Hosting Functional Programming Languages

Caner Derici

December 3, 2019

Abstract

This thesis presents solution approaches for some of the fundamental performance issues of a self-hosting functional language implementation on a meta-tracing framework and argues that a competitive performance is achievable.

I will start by discussing how meta-tracing enables rapid prototyping of efficient dynamic language implementations. To establish the backbone of our further discussion, we will discuss the use of interpreter hints in meta-tracing to allow discovering and tracing the loops in a user program, rather than in the interpeter evaluating it. I will also discuss how meta-tracing makes a fast Racket interpreter, namely Pycket.

Next I will introduce the linklet form to study how Racket made its implementation more independent from the run-time, thereby achieving a greater flexibility to target different virtual machines. I will develop a formal model of linklets to investigate how they work, and study how the communication between the front and back-end of Racket is established via the linklets.

Then I will demonstrate the process of incorporating linklets into the Pycket to eventually turn it into an implementation of the self-hosting Racket. We will discuss how Racket's self-hosting works with bootstrapping linklets, how loading and evaluating a Racket module works on Pycket from start to end, and demonstrate with examples how functionalities like the top-level repl implemented with linklets.

Finally I will study the performance issues that are fundamental to implementing a self-hosting interpreter on a meta-tracing framework. I will provide preliminary experiments and discussion to reveal the key points of these issues that we have observed during the improvement of Pycket, and propose approaches to addressing the issues with additional experiments and formalism, as well as demonstration on the new Pycket, to improve the performance enough to justify using meta-tracing to implement VMs for self-hosting functional languages.

1 Introduction

To contextualize the thread of work I propose in this thesis, this section gives a general introduction to the rapid implementation of dynamic programming languages using meta-tracing, and then discusses the increased portability in Racket and provides a general introduction on the problem of self-hosting on meta-tracing.

Meta-tracing & Pycket Over the past several decades the tracing *just-in-time (JIT)* virtual machines (VM) gained a significant popularity for their substantial performance benefits. On another front, instead of tracing a program, tracing an interpreter evaluating that program, namely the meta-tracing, allowed developers and researchers to rapidly prototype fast VMs for dynamic languages [17]. Given only the interpreter of the language (possibly annotated with some hints), meta-tracing can automatically produce an efficient executable for the interpreter (by effectively compiling the interpreter to C), and also build a JIT compiler into the binary as well. For example, the PyPy, a Python interpreter built via the RPython meta-tracing framework has been a big success, achieved a speedup by a factor of 6.54 over the interpreter (in C), outperforming even the CPython itself [17]. In 2014, Bolz and collaborators discovered that meta-tracing can increase the performance of dynamic functional programming languages as well [15], and created Pycket, a high-level interpreter for Racket based on the control, environment, and continuation (CEK) abstract machine [27]. Built on the RPtyhon framework, Pycket was able to outperform Racket's own VM and other highly-optimized Scheme compilers, while also demonstrating a significant reduction on the overhead of contracts and gradual typing [8, 9].

More Portability for Racket Racket's original interpreter, as well as the compiler and run-time (including the built-in primitives and data structures) has been in C since its first design in 1995. Porting the GUI layer in 2010 and the expander in 2016 from C to Racket has verified that Racket is easier to maintain and modify than C. This started the effort of porting more of Racket from C to Racket, which in turn, created the need of a more maintainable host run-time system. In 2017, Racket decided to adopt the Chez Scheme as its run-time. To to that end, two important implementation decisions (among others) allowed Racket to be more flexible in re-targeting different VMs: ***i)*** separated the expansion and the compilation that were fused together in the original C implementation ***ii)*** export some functionality that's essential for self-hosting (e.g. reader, expander etc.) for the run-time to utilize (e.g. to read and expand Racket modules). This relieved any targeted run-time from implementing those functionalities that are quite large in terms of both code size and semantics, thereby allowing an easier hosting of the language. This approach is used to adopt the Chez Scheme and it also made it easier for potential hosting run-times for Racket [29].

Meta-tracing & Self Hosting With Racket being easier to host in different VMs and already having an interpreter for it in Pycket, it is logical to consider self-hosting Racket on RPython, thereby turning Pycket into an implementation of Racket. However, self-hosting on a meta-tracing framework poses challanges in terms of performance. The most common problem is the inconsistency in the performance improvements, especially for programs that has frequent changes in the control flow. In 2011, Mozilla reported having this problem in their TraceMonkey JIT where they observed a massive speed-up in performance for the parts of the programs that has tight[1] loops, while suffering substantially in performance for other parts, stating "You lose more when slow than you gain when fast" [51]. This problem becomes more apparent when meta-tracing is involved, because the program that is being traced is another interpreter, which inherently involves many different control flow paths at each iteration in its dispatch loop [13]. Increasing the meta level of the VM by tracing a self-hosting interpreter, therefore, poses even a greater challenge. Moreover, self-hosting a modern functional programing language like Racket requires additional functionalities that naturally entail dispatch-loops, such as expander, reader, regular expressions and fast-load serialization (FASL).

2 Thesis Statement

My thesis statement is:

> An efficient self-hosting functional programming language on a meta-tracing JIT is achievable.

I will support this thesis statement with the following research contributions:

- *An implementation of self-hosting Racket on RPython.* I will demonstrate that Pycket has all the necessary run-time support (e.g. primitives, data structures, error handling etc.) to load and evaluate Racket code without requiring any external support.
- *Approaches in achieving good performance* I will identify the fundamental issues in self-hosting a language such as Racket on a meta-tracing JIT compiler, and provide approaches in addressing them. I will vet the run-time performance of the proposed approaches on the improved Pycket and argue why these approaches enable a full and efficient implementation.

[1] having only a single exit/loop point

3 Technical Overview

In this section, I will expand on the introduction, and start with discussing in Section 3.1 the RPython framework and how meta-tracing works. Then in Section 3.2, I will study the Racket's improved portability via linklets. Next in Section 3.3 I will describe how the Racket's improved portability is used to turn Pycket into a run-time for a self-hosting Racket. Finally in Section 3.4, I will introduce the performance problems that we identified as fundamental to self-hosting and meta-tracing, and propose approaches on addressing them.

3.1 Tracing JITs & RPython Framework

JIT compilers allow programs to be compiled dynamically at run-time during the evaluation via an interpreter. As opposed to the ahead-of-time (AOT) compilers, JITs interleave the compilation with execution, aiming to compile and optimize only the parts of the program that are used the most. JIT compilers employ a low-overhead profiling to dynamically detect a frequent (i.e. hot) sequence of instructions during interpretation. Starting with evaluating a given program with an interpreter, a JIT compiler stops the evaluation when a frequently evaluated instruction path is detected, compiles the instructions and starts using the compiled path whenever the same path needs to be evaluated again [7].

JITs have been shown to be significantly effective in implementing efficient VMs for dynamic languages. There are two main approaches in JIT compilation. Method-based compilation work on the method level, and compile the most frequently used methods in the program, while trace-based JIT compilers compile the most frequently evaluated loops [5, 6]. In this thesis, we're interested in the trace-based JIT compilation, as we work with meta-tracing,

Trace-based JIT compilation works on two principal assumptions:

i) Programs spend most of their running time in loops.

ii) Several iterations of the same loop are likely to take similar code paths.

```
# start of the trace (preamble)
label(p0, p1, descr="64723312")
guard_not_invalidated()
guard_class(p0, ConsEnv)
p3 = getfield_gc_r(p0, descr="ConsEnv.prev")
guard_class(p3, ConsEnv)
i5 = getfield_gc_i(p3, descr="Fixnum")
i6 = getfield_gc_i(p0, descr="Fixnum")
i7 = int_add_ovf(i5, i6)
guard_no_overflow()
guard_class(p1, LetCont)
p9 = getfield_gc_r(p1, descr="LetCont.ast")
guard_value(p9, ConstPtr(ptr10))
p11 = getfield_gc_r(p1, descr="LetCont.env")
p12 = getfield_gc_r(p1, descr="LetCont.prev")
# peeled-iteration (inner loop)
label(p11, i7, p12, descr="64723392")
guard_not_invalidated()
guard_class(p11, ConsEnv)
i14 = getfield_gc_i(p11, descr="Fixnum")
i15 = int_add_ovf(i14, i7)
guard_no_overflow()
guard_class(p12, LetCont)
p17 = getfield_gc_r(p12, descr="LetCont.ast")
guard_value(p17, ConstPtr(ptr18))
p19 = getfield_gc_r(p12, descr="LetCont.env")
p20 = getfield_gc_r(p12, descr="LetCont.prev")
# jump
jump(p19, i15, p20, descr="64723392")
```

Figure 1: An example trace

To exploit these assumptions, a tracing JIT classifies certain execution paths to be *hot loops* during the evaluation. When a hot loop is detected, as described before the evaluation stops, and loop is compiled and optimized into a *trace*, and then the execution continues, using the compiled trace whenever the same path needs to be executed. A trace is a linear sequence of instructions with an entry point and one or more exit points. Figure 1 shows a simplified example of a typical trace, taking two arguments $p0$ and $p1$ as inputs, having a preamble (because of unrolling) and an inner loop that jumps to itself. The *guards* within a trace make up the trace's exit points, where the trace jumps out if certain conditions are not satisfied. They are often used for **i)** making sure in the preamble the conditions are the same when this sequence is first traced, and **ii)** to finish and exit the loop.

Meta-tracing Evaluating a byte-code interpreter for a language in place of a regular program on a tracing JIT naturally forms an implementation of a language. However, without any treatment the loops that are traced during the execuion of the interpreter not necessarily correspond to the loops implemented in the user program that is being interpreted. In particular, since a byte-code interpreter is basically a dispatch loop for the byte-code instructions in the language, a trace of an execution path within such an interpretation loop consists of instructions implementing only one particular byte-code. However, it is rarely the case where a single byte-code is repeatedly executed when evaluating a program on an interpreter. Additionally the loops in the user program, in which most of the running time will be spent are never being traced. Therefore this approach leads to compiling traces that most likely will be used only once at a time (exit out at the end of the trace, as the next byte-code will not be the same), resulting in a worse performance than even just interpreting.

Meta-tracing introduces certain hints and annotations for the tracer to be utilized within the executed program (in our case the interpreter) to capture the useful traces that will be executed repeatedly, e.g. in the case of an interpreter the ones that will correspond to the loops in the

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

def interpret(bcode, a):
  regs = [0] * 256
  pc = 0

  while True:
    driver.jit_merge_point(bcode=bytecode,
                           pc=pc,a=a,
                           regs=regs)
    opcode = ord(bytecode[pc])
    pc += 1

    if opcode == JUMP_IF_A:
      target = ord(bcode[pc])
      pc += 1
      if a:
        if target < pc:
          driver.can_enter_jit(bytecode=bcode,
                               pc=target,
                               a=a, regs=regs)
        pc = target
    elif opcode == MOV_A_R:
      ...
```

Figure 2: An interpreter annotated with hints. Figure adapted from [17]

user program. The loop detection is simply finding a spot where the program jumped backwards. Figure 2 shows an example interpreter decorated with such annotations and hints. The annotations allow the implementer to set the parameters (*greens*) that constitute what can be considered as a compund program counter (pc), by which the tracer will detect that the program jumped backwards. Additionally the hints allow the points where the program might jump backwards to be stated. The reader is referred to [17] for a detailed discussion.

RPython framework RPython (Restricted Python) is a statically typed, object-oriented proper subset of Python that is designed during PyPy's development [42]. It restricts Python in a way that enables type-inference on RPython programs allowing an efficient conversion to a lower language such as C. The restrictions are mainly the dynamic features of Python and commonly listed as [2, 18]; no mix types at the same location (all variables need to be type consistent and infereable), run-time reflection is not supported (i.e. changing method of classes at run-time), no closures, global and class bindings are assumed to be constants etc.

The RPython framework takes RPython code and converts it to a chosen lower-level language, most commonly C. Given an interpreter written in RPython, the toolchain translates it to the target language by lowering it in numerous phases, where each phase has its own type system and a generic type inference engine. Because the RPython is a subset of Python, the entire process can use a Python run-time. The general translation process can be described as follows. It starts with loading the program into the run-time and get the function objects in memory as inputs. Using these function objects, the framework generates by abstract intepretation the control flow graphs for these functions that will be processed at further transformation steps. Then the annotator acts as a high-level type inference engine and assigns "annotations" to each variable at each flow graph. These annotations basically denotes the possible Python objects a variable might contain at the run-time. After the whole program is annotated, RTyper (RPython typer) takes control and starts lowering the annotations and some operations into the lower types and operations that would make sense to the targeted environment (e.g. structs, pointers, arrays in case of C), acting as a bridge between the high-level annotator and the low-level code generation. After RTyping, some optional backend optimizations are applied, such as inlining, malloc removal and escape analysis. Note that like Python, RPython is garbage collected, however, C is not. Therefore at this point a garbage collector is inserted into the program as well. Finally the typed and lowered flow-graphs are inputted into the code generation and the binary is generated. [2, 42, 41]

3.2 Making Racket More Portable With Linklets

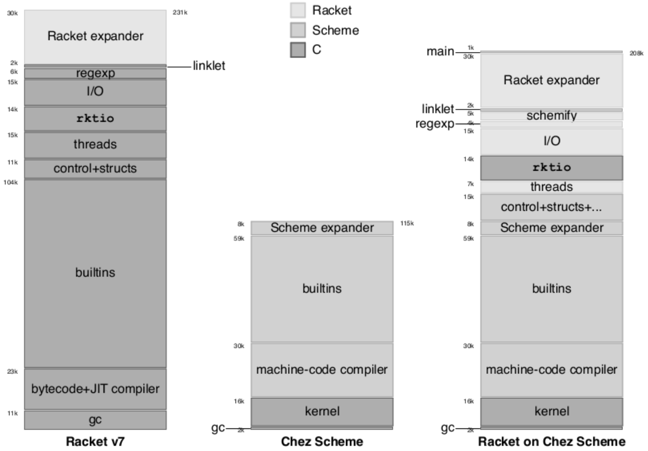
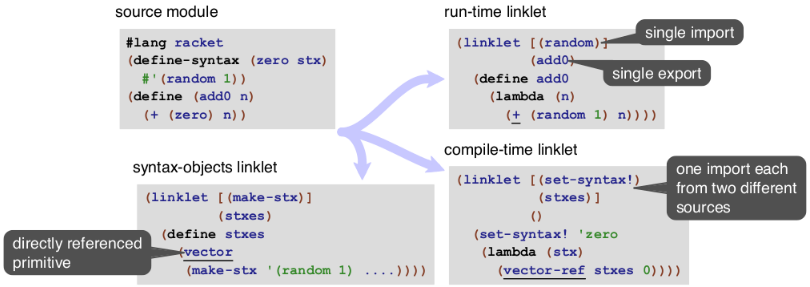


Figure 3: Figure used from [29]

In this section, we study how Racket became more portable to easily re-target different virtual machines. I will present the general idea and introduce the use of linklets as an essential part of a communication language between the language and the run-time. Then I will provide and example that demonstrates how functionalities such as the top-level are implemented in this setup. Next I will provide the formalism for linklets that allows us to easily reason about and implement them on VMs.

As described in Section 1, part of the effort of targeting a new run-time for Racket was making it more portable by separating the parts that are essential for hosting Racket from the run-time implementation. Figure 3 summarizes how different parts fit together. In the previous versions of Racket, each segment of the leftmost figure was in C. The first step was to port the expander from C to Racket, as the Racket expander implements most of the essential parts for the front-end such as the module system, macro system, read, expand etc. Aside from being ported from C to Racket, the expander is also made to elaborate given Racket code into a core language for the hosting compiler to consume. As we will discuss in more detail in the remainder of this section, the essential forms of this language, acting as units of compilation in the run-time, are called the ”linklets”, which are lambda-like forms that consume and produce potentially mutable variables (instead of values).

The way that the expander elaborates a given Racket program is to produce a bundle of linklets for the different parts of this program. Figure 4 shows an example where a Racket module is expanded into such a bundle of linklets. As the expander strictly separates the expansion and run-time phases, three linklets are produced for the given source module; one for the expansion-time (compile-time) and one for the run-time and one for the literal syntax objects to bridge the two worlds. Linklets can directly refer to the primitive functions as well. As we will detail in Section 3.3, they assume roughly 1500 primitive functions that are implemented in the run-time, such as *vector-ref* and + as it is seen in Figure 4.



```
#lang racket
(define-syntax (zero stx)
  #'(random 1))
(define (add0 n)
  (+ (zero) n))
```

```
(linklet [(random)]
         (add0)
  (define add0
    (lambda (n)
      (+ (random 1) n))))
```

```
(linklet [(set-syntax!)
          (stxes)]
         ()
  (set-syntax! 'zero
   (lambda (stx)
     (vector-ref stxes 0))))
```

```
(linklet [(make-stx)]
         (stxes)
  (define stxes
    (vector
     (make-stx '(random 1) ....))))
```

Figure 4: Expansion of a Racket source module into a linklet bundle. Figure used from [29]

Using this, just like any other Racket module, when the expander is given to itself as input, we get an implementation of the expander as a bundle of linklets that a compiler can consume. Along with the expander itself, this process (applying the expander) is also used to export several different functionalities as independent linklets for the run-time. These linklets are called the *bootstrapping linklets*. Currently Racket exports four such linklets: *expander*, *io*, *threads*, *regexp* [2]. Among them, the *expander* linklet is the key linklet for making Racket more portable, as it provides the implementations of essential functionalities such as the module system, macro system, read, eval etc.

A hosting compiler that implements a thin API layer to process linklets (the *linklets* layer in Figure 3) can then consume these bootstrapping linklets to load the essential functionalities into its run-time, for example loading the expander linklet to read, expand and evaluate any given Racket code. That API is usually accommodated with an adapter layer in the run-time. For example in Chez Scheme, the *schemify* layer in the rightmost figure of the Figure 3 converts the produced linklets into Chez Scheme lambda forms. We will discuss the implementation and use of this API in more detail in Section 3.3.

3.2.1 Top-level REPL via Linklets

Let’s take a look at how a VM that implements linklets can easily implement functionalities such as the Racket top-level repl[3] by loading the expander linklet into its run-time. Note that the top-level itself is quite complicated and beyond the scope of our study. The example here aims to provide a general overview of the interaction between the run-time and the functionalities provided by the language through the linklets, for this exercise in particular the expander linklet. Recall that the expander implements functions such as read, expand, eval, and outputs a bundle of linklets for a given Racket code. Let’s first briefly describe what linklets are and how linklets operate. In Section 3.2.2 I will provide linklets’ operational semantics with a formalism in more detail.

A linklet consists of a set of variable definitions and expressions, an exported subset of the defined variable names, a set of variables to export from the linklet despite having no corresponding definition, and a set of imports that provide other variables for the linklet to use. Some linklet examples can be seen in Figure 4. To run a linklet, it is instantiated as a *linklet instance*, which is a collection of linklet variables. When a linklet is instantiated, it receives other linklet instances for its imports, and it extracts a specified set of variables that are exported from each of the given instances. The newly created linklet instance provides its exported variables for use by other linklets. Moreover, an instantiation may be given an additional linklet instance as an input, namely the target instance. In this case, the variables in the target instance is used and modified for the linklet definitions and expressions during the evaluation of the body expressions, and the result is the value of the last expression in the linklet (instead of an instance). In

[2] there’s also the *schemify* and *known* linklets, but they’re specifically for Chez Scheme integration

[3] read-eval-print loop

the remainder of this document we will use the names “targeted instantiation” and “regular instantiation” for these two modes.

The minimal API for a virtual machine to implement the linklets consists of the functions **i)** `compile-linklet` for preparing a linklet for instantiation and **ii)** `instantiate-linklet` for running a compiled linklet. In particular, the VM uses the `compile-linklet` to take an s-expression for a linklet as input and produce a linklet object in a representation of its choice, containing the run-time AST for the codes inside the linklet. Therefore, loading a linklet into its run-time for a VM means that the functions provided by the linklet are *callable* during the run-time. For example after loading the expander linklet, a VM can *call* the functions from Racket’s expander such as read, expand, eval to use Racket’s module system, macro system etc.

As our example, let’s consider the interactions in Figure 5. The key point here is that the variable ”a” is defined *after* it is used within the function ”k”. To implement this repl, the VM will keep a dedicated linklet instance that we will refer to as the *top-level* instance. Recall that a linklet instance is a collection of linklet variables. In our study we assume that it is implemented as a finite mapping of identifiers to linklet variables. Initially the top-level instance will be an empty instance.

```
> (define k (lambda () a))
> (define a 10)
> (k)
10
```

Figure 5: Top-level Example

The key point is the interaction between the language and the run-time through the expander linklet. At each step in the repl, the VM gets a Racket expression to evaluate. Then it applies the `read` and `expand` functions imported from the expander linklet, on the given expression. For clarity, we will omit the non-essential details and focus only on the simplified run-time linklet for the expression.

Let’s see what happens at each interaction:

> **(define k (lambda () a))** As we briefly explained earlier, the VM will call the expander’s `expand` function, which will output a linklet containing (the expanded version of) this expression. Then the VM will use `instantiate-linklet` to instantiate it over the *top-level* instance, which is initially empty.

```
(linklet (...) (a k)
  (define-values (k)
    (lambda () a)))
```

(a) Linklet generated by `expand`

```
(linklet (...) ((a0 a) (k0 k))
  (define-values (k)
    (lambda ()
      (variable-ref (LinkletVar a0))))
  (variable-set! (LinkletVar k0) k))
```

(b) Linklet compiled by VM

$a_0 \rightarrow var_a$

$k_0 \rightarrow var_k$

(c) Environment after compile

$a \rightarrow var_a$

$k \rightarrow var_k$

(d) Target after instantiate

The important point about the top-level part in this example is the handling of the undefined identifier ”a”. When the expander generates a linklet containing an unidentified identifier, it puts the identifier among the exported variables of the linklet (despite having no corresponding definition for it), allowing it to be handled via the linklet variables. During the compilation of the linklet on the VM, a new linklet variable is created for it with the special value *uninitialized*, and when the binding for the identifier is needed, 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 we have the variable mappings for both ”a” and the ”k” within the target instance after the instantiation, containing the values *uninitialized* and a closure with no arguments and body with a reference to var_a respectively.

> **(define a 10)** When this new expression is inserted in the repl, the VM calls the `expand` again to produce a linklet, then it compiles and instantiates it over the same *top-level* target instance.

```
(linklet (...) (a)
  (define-values (a) 10))
```

(a) Linklet generated by `expand`

```
(linklet (...) ((a1 a))
  (define-values (a) 10)
  (variable-set! (LinkletVar a1) a))
```

(b) Linklet compiled by VM

$a_1 \rightarrow var_a$

(c) Environment after compile

$a \rightarrow var_a$

$k \rightarrow var_k$

(d) Target after instantiate

Note that the `compile-linklet` inserted the *variable-set!* expression in the body to set the value of the variable bound by the external name ”a”. During the instantiation, this variable exists in two possible ways. When the exported variables are processed at the start of the instantiation, if the target instance doesn’t have a variable mapping for ”a”, then a new variable will be created with the *uninitialized* value (as in the previous step), otherwise if the target does have a mapping for "a" (which it does in this case), then the target's variable will be used during the instantiation, and will be set by *variable-set!*. Note that because of this the binding for a_1 in the environment above is set to be the var_a (instead of a new variable), since the target does has a mapping for "a".

> **(k)** At this step, we have everything we need to evaluate this expression thanks to the *top-level* target instance.

```
(linklet (...) (k) (k))
```

(a) Linklet
generated by `expand`

```
(linklet (...) ((k1 k))
  ((variable-ref (LinkletVar k1))))
```

(b) Linklet
compiled by VM

$k_1 \rightarrow var_k$

(c) Environment
after compile

$a \rightarrow var_a$
$k \rightarrow var_k$

(d) Target
after instantiate

When instantiating the compiled linklet, because of the first step, the value for the variable var_k is resolved as a closure with no arguments and containing a reference to var_a in the body. Similarly when that closure is applied, the value for the var_a is resolved as 10, thanks to the second step.

3.2.2 Operational Semantics of Linklets

In this section I present a formalism to briefly describe the operational semantics of linklets, which is the basis of the implementation on Pycket. The formalism is developed in the PLT Redex language, therefore it is executable [26]. Randomized testing is used to verify the model [36], and a parallel test suite is used to verify the Pycket implementation and the model with each other.

Only the parts of the formalism that are essential to the discussion in this document are shown here, the presentation of the full formalism and the semantics are deferred to the dissertation.

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

Figure 9: Linklet Source Language [4]

Figure 9 shows the grammar of the linklet language, which is a minimal subset of the Fully Expanded Racket Programs extended with the `linklet` form and toplevel definitions. The sub-language that is used within the linklet body, namely the Racket Core (RC) is basically a λ-calculus with a handful of syntactic extensions. In addition to the basic forms such as `begin`, `set!`, `if` etc., RC has some additional forms to manipulate linklet variables (e.g. *var-ref*), as shown in **e** rule.

Before a linklet is ready for instantiation, as part of the `compile-linklet` process, several passes over the linklet are performed to ensure a correct information flow between the variables in the run-time. For each import and export identifier (and possibly their corresponding renamings), *Import* and *Export* objects are created with the fresh references that will be used to name the variables that are dynamically created during the instantiation. Additionally, a pass on the linklet body collects all the identifiers that are defined within the linklet body and all the mutated identifiers.

Given a set of *Imports* (C_I) and a set of *Exports* (C_E), a set of toplevel defined identifiers (X_T) and a set of mutated identifiers (X_M), the compilation starts to process each expression in the linklet body one by one, performing the following actions depending on the type of expression:

- For a '(**define-values** (x) e)' expression where ($\textbf{Export}\ x_{gen}\ x\ x_{ext}$) $\in C_E$ (i.e. x is exported), add an additional '(**var-set!** x_{gen} x)' expression.
- For a '(**set!** x e)' expression where ($\textbf{Export}\ x_{gen}\ x\ x_{ext}$) $\in C_E$ (i.e. x is exported), turn the expression into '(**var-set/check-undef!** x_{gen} e)'.
- For any identifier x where ($\textbf{Import}\ x_{gen}\ x\ x_{ext}$) $\in C_I$ (i.e. x is imported), turn the identifier into '(**var-ref/no-check** x_{gen})'.
- For any identifier x where ($\textbf{Export}\ x_{gen}\ x\ x_{ext}$) $\in C_E$ and either $x \in X_M$ or $x \notin X_T$, turn the identifier into '(**var-ref** x_{gen})'.
- For any other expression e, recurse if e has subforms, return otherwise.

[4] Some rules (e.g. evaluation contexts etc.) are omitted for space.

$$
\begin{aligned}
CL &::= \boldsymbol{\Phi^{C}}(L) \\
\textit{L-obj} &::= (\boldsymbol{L_{\alpha}}\ \textit{c-imps}\ \textit{c-exps}\ \textit{l-top} \ldots) \\
&\quad \mid (\boldsymbol{L_{\beta}}\ x\ \textit{l-top}\ \ldots) \\
LI &::= (\textbf{linklet-instance}\ (x\ var)\ \ldots) \\
\textit{c-imps} &::= ((\textit{imp-obj}\ \ldots)\ \ldots) \\
\textit{c-exps} &::= (\textit{exp-obj}\ \ldots) \\
\textit{imp-obj} &::= (\textbf{Import}\ x\ x\ x) && \text{[id int ext]} \\
\textit{exp-obj} &::= (\textbf{Export}\ x\ x\ x) && \text{[id int ext]}
\end{aligned}
$$

Φ^{C} : **compile-linklet**

Figure 10: Linklet Runtime Language [4]

Figure 10 shows the grammar of the run-time language for linklets, i.e. expressions that represent the run-time objects used during an instantiation of linklets. Given a linklet L, *compile-linklet* produces a linklet object $\boldsymbol{L_{\alpha}}$ that is ready to be instantiated. $\boldsymbol{L_{\alpha}}$ consists of *Import* and *Export* objects for the identifiers denoting the imported and exported variables, and the body expressions with potentially some additional expressions for variable manipulations in the run-time.

Each *Import* and *Export* object consists of two identifiers that are internal and external with respect to the linklet body (to accommodate renamings), and one fresh identifier for internal use during the instantiation. The *Import* and *Export* objects are processed as the first step of the instantiation. As we detail below, references for the imported variables are collected from the given import instances, and additional variables may be created for exported variables depending on the target instance as we have shown in the example in Section 3.2.1. $\boldsymbol{L_{\beta}}$ denotes a compiled linklet object after such a preparation step, with which the actual instantition may begin. Note that $\boldsymbol{L_{\beta}}$ contains only a reference (to the target instance that is used during the instantiation) and the body expressions, and doesn't contain the import and export objects, as the instantiation uses the variables that are already prepared for them. Finally a *linklet-instance* is a finite mapping from symbols to variables.

Since the linklets are lambda-like binding forms and not capable of starting a computation by themselves, we introduce a top-level form, namely `program`. A `program` consists of a set of given linklets to be loaded and a single top-level expression to perform a computation. Figure 11 shows part of the grammar for `program`s. We provided the `let-inst` form to name a linklet instance, and `seq` form to perform multiple computations in sequence.

$$
\begin{aligned}
p &::= (\textbf{program}\ ((x\ L)\ \ldots)\ \textit{p-top}) \\
\textit{p-top} &::= v \mid (\textbf{let-inst}\ x\ \textit{p-top}\ \textit{p-top}) \\
&\quad \mid (\textbf{seq}\ \textit{p-top}\ \ldots) \\
&\quad \mid \boldsymbol{\Phi^{I}}(\textit{l-ref}\ x\ \ldots) \mid \boldsymbol{\Phi^{I}}(\textit{l-ref}\ x\ \ldots\ \texttt{\#:t}\ x) \\
&\quad \mid \boldsymbol{\Phi^{V}}(x\ x) \\
\textit{l-ref} &::= x \mid \textit{L-obj} \\
v &::= \ldots. \mid (v\ x)
\end{aligned}
$$

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

Figure 11: Linklet Program Source Language [5]

As an example, we can represent the top-level repl example from the previous section as a linklet *program*, and simulate the process. The (multi-step) reduction below demonstrate the compilation of each linklet, representing the linklets produced by the expander at each repl interaction. After the compilation, the linklets are substitiuted in the body and the evaluation starts with reducing the body. Figure 12 shows the reduction relation, β_p, that implements the evaluation, together with the reduction relation for RC, β_r. For clarity, the reduction of the programs and the linklet instantiation is defined together by the same relation (β_p) so that the whole evaluation generating a single small-step sequence of reductions from programs to values.

	program	ρ	σ
	(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))))	[]	[]
$\longrightarrow_{\beta p}$ *	(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))))	[]	[]

After the compilation and loading phase, all the linklet objects are in $\boldsymbol{L_{\alpha}}$ form and ready to be instantiated. Recall that a linklet instantiation has two modes; **i)** regular instantiation, where a new instance is being created and returned, and **ii)** targeted instantiation, where the given target instance is used during the instantiation. Since the only difference between the two modes is the return value, for simplicity in the model we remove the distinction by turning the regular instantiation into a targeted instantiation with a fresh empty instance. The result of both the instantiation is then a pair of the last expression's value and the target instance being used.

[5] Some rules (e.g. evaluation contexts etc.) are omitted for space.

At the start of instantiation, the imported variables are collected from the given linklet instances by simply going through each set of *Import* objects in the $\boldsymbol{L_\alpha}$ and the corresponding linklet instance that provides the variables. The collected variable references are then put into the environment with the names in the corresponding *Import* objects that were generated for the variables during the compilation. Recall that a linklet may export a variable despite having no corresponding definition for it. This is achieved by for each *Export* object in $\boldsymbol{L_\alpha}$ grabbing the variable from the target instance if it has a mapping for it and making it available in the environment. Otherwise a fresh variable is created and its mapping is placed into the environment and into the target instance as well.

After processing the import and export variables, the $\boldsymbol{L_\alpha}$ object is reduced into $\boldsymbol{L_\beta}$, which triggers the evaluation of the expressions in the linklet body. The linklet compilation makes sure that every variable reference within the body uses the appropriate variable, and at the beginning of the instantiation the preparation of the import and export variables explained above provides the actual variables in the environment to be modified and used. The reduction then proceeds with the evaluation of the body expressions one-by-one. After all the expressions are processed, a pair of the last expression's value and the reference to the target instance is returned. The reduction of the program continues by similarly instantiating the other two linklets and finally returning the last value in the `seq` form.

$$
\begin{aligned}
EP\ \llbracket E\ \llbracket x \rrbracket\rrbracket, \rho, \sigma &\longrightarrow_{\beta r} EP\ \llbracket E\ \llbracket \sigma[\rho[x]] \rrbracket\rrbracket, \rho, \sigma \\
EP\ \llbracket E\ \llbracket(\textbf{var-ref}\ x))\rrbracket\rrbracket, \rho, \sigma &\longrightarrow_{\beta r} EP\ \llbracket E\ \llbracket v \rrbracket\rrbracket, \rho, \sigma\ \ \textbf{where}\ x \in \rho, \rho[x] \in \sigma, v = \sigma[\rho[x]], v \neq \mathit{uninit} \\
EP\ \llbracket E\ \llbracket(\textbf{var-ref/no-check}\ x))\rrbracket\rrbracket, \rho, \sigma &\longrightarrow_{\beta r} EP\ \llbracket E\ \llbracket v \rrbracket\rrbracket, \rho, \sigma\ \ \textbf{where}\ v = \sigma[\rho[x]] \\
EP\ \llbracket E\ \llbracket(\textbf{var-set!}\ x\ v)\rrbracket\rrbracket, \rho, \sigma &\longrightarrow_{\beta r} EP\ \llbracket E\ \llbracket(\mathit{void})\rrbracket\rrbracket, \rho, \sigma_1\ \textbf{where}\ \sigma_1 = \sigma[\rho[x] \to v] \\
EP\ \llbracket E\ \llbracket(\textbf{var-set/check-undef!}\ x\ v)\rrbracket\rrbracket, \rho, \sigma &\longrightarrow_{\beta r} EP\ \llbracket E\ \llbracket(\mathit{void})\rrbracket\rrbracket, \rho, \sigma_1\ \textbf{where}\ \sigma_1 = \sigma[\rho[x] \to v], x \in \rho \\
EP\ \llbracket E\ \llbracket(\textbf{set!}\ x\ v)\rrbracket\rrbracket, \rho, \sigma &\longrightarrow_{\beta r} EP\ \llbracket E\ \llbracket(\mathit{void})\rrbracket\rrbracket, \rho, \sigma_1\ \textbf{where}\ \sigma_1 = \sigma[\rho[x] \to v] \\[1em]
EP\ \llbracket \boldsymbol{\Phi^V}(x_{li},\ x)\rrbracket, \rho, \sigma &\longrightarrow_{\beta p} EP\ \llbracket v \rrbracket, \rho, \sigma\ \textbf{where}\ v\ =\ \sigma[(\sigma[x_{li}])[x]] \\
EP\ \llbracket(\boldsymbol{\Phi^I}\ (\boldsymbol{L_\beta}\ x\ v\ \ldots\ v_l))\rrbracket, \rho, \sigma &\longrightarrow_{\beta p} EP\ \llbracket(v_l\ x)\rrbracket, \rho, \sigma \\
EP\ \llbracket(\boldsymbol{\Phi^I}(\boldsymbol{L_\beta}\ x_t\ v_p\ \ldots\ (\textbf{define-values}\ (x)\ v)\ \mathit{l\text{-}top}\ \ldots))\rrbracket, &\longrightarrow_{\beta p} EP\ \llbracket \boldsymbol{\Phi^I}(\boldsymbol{L_\beta}\ x_t\ v_p\ \ldots\ \mathit{l\text{-}top}\ \ldots)\rrbracket, \rho_1, \sigma_1 \\
&\qquad \textbf{where}\ \rho_1 = \rho[x \to \mathit{cell}],\ \sigma_1 = \sigma[\mathit{cell} \to v] \\
EP\ \llbracket(\boldsymbol{\Phi^I}(\boldsymbol{L_\alpha}\ \mathit{c\text{-}imps}\ \mathit{c\text{-}exps}\ \mathit{l\text{-}top}\ldots)\ LI\ \ldots)\rrbracket, \rho, \sigma &\longrightarrow_{\beta p} EP\ \llbracket(\boldsymbol{\Phi^I}(\boldsymbol{L_\alpha}\ \mathit{c\text{-}imps}\ \mathit{c\text{-}exps}\ \mathit{l\text{-}top}\ \ldots)\ LI\ \ldots\ \textbf{\#:t}\ x_t)\rrbracket, \rho, \sigma_1 \\
&\qquad \textbf{where}\ x_t \notin \mathit{dom}(\sigma), \sigma_1 = \sigma[x_t \to (\mathit{linklet\text{-}instance})] \\
EP\ \llbracket(\boldsymbol{\Phi^I}(\boldsymbol{L_\alpha}\ \mathit{c\text{-}imps}\ \mathit{c\text{-}exps}\ \mathit{l\text{-}top}\ldots)\ LI\ \ldots\ \textbf{\#:t}\ x_t)\rrbracket, \rho, \sigma &\longrightarrow_{\beta p} EP\ \llbracket(\boldsymbol{\Phi^I}(\boldsymbol{L_\beta}\ x_t\ \mathit{l\text{-}top}\ \ldots))\rrbracket, \rho_2, \sigma_1 \\
&\qquad \textbf{where}\ \rho_1 = V^I(\mathit{c\text{-}imps}, (LI\ \ldots), \rho) \\
&\qquad\quad (\rho_2, \sigma_1) = V^E(\mathit{c\text{-}exps}, x_t, \rho_1, \sigma) \\
(\textbf{program}\ ((x\ L), (x_1\ L_1)\ \ldots)\ \mathit{p\text{-}top}), \rho, \sigma &\longrightarrow_{\beta p} (\textbf{program}\ ((x_1\ L_1)\ \ldots)\ \mathit{p\text{-}top}[x := \boldsymbol{\Phi^C}(L)]), \rho, \sigma \\[1em]
\mathit{eval}^{\longrightarrow_{\beta p}}(p) &=\ v\ \textbf{if}\ p, (), () \twoheadrightarrow_{\beta p \cup \beta r}\ (\textbf{program}\ ()\ (v\ \_))
\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 12: Reduction Relation [6]

	program	ρ	σ

$\longrightarrow_{\beta p}$	`(program ()`
	`(seq (`ϕ^I` (L`α` () ((Export a1 a a) (Export k1 k k))`
	`(define-values (k) (lambda () (var-ref a1))) (var-set! k1 k) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export a1 a a))`
	`(define-values (a) 10) (var-set! a1 a) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export k1 k k)) ((var-ref k1))) #:t t))))`
		[]	[t → (linklet-instance),
			li → (linklet-instance)]

$\longrightarrow_{\beta p}$	`(program ()`
	`(seq (`ϕ^I` (L`β` t (define-values (k) (lambda () (var-ref a1))) (var-set! k1 k) (void)))`
	`(`ϕ^I` (L`α` () ((Export a1 a a))`
	`(define-values (a) 10) (var-set! a1 a) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export k1 k k)) ((var-ref k1))) #:t t))))`
		[k1 → var_k,
		a1 → var_a]
			[var_a, var_k → uninit,
			t → (linklet-instance
			(a var_a) (k var_k)),
			li → (linklet-instance)]

$\longrightarrow_{\beta p}$	`(program ()`
	`(seq (`ϕ^I` (L`β` t (var-set! k1 k) (void)))`
	`(`ϕ^I` (L`α` () ((Export a1 a a))`
	`(define-values (a) 10) (var-set! a1 a) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export k1 k k)) ((var-ref k1))) #:t t))))`
		[k → $cell_1$,
		k1 → var_k,
		a1 → var_a]
			[$cell_1$ → closure,
			var_a, var_k → uninit,
			t → (linklet-instance
			(a var_a) (k var_k)),
			li → (linklet-instance)]

$\longrightarrow_{\beta r}$	`(program ()`
	`(seq (`ϕ^I` (L`β` t (var-set! k1 (closure () (var-ref a1) ((k1 `var_k`) (a1 `var_a`)))) (void)))`
	`(`ϕ^I` (L`α` () ((Export a1 a a))`
	`(define-values (a) 10) (var-set! a1 a) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export k1 k k)) ((var-ref k1))) #:t t))))`
		[k → $cell_1$,
		k1 → var_k,
		a1 → var_a]
			[$cell_1$ → closure,
			var_a, var_k → uninit,
			t → (linklet-instance
			(a var_a) (k var_k)),
			li → (linklet-instance)]

$\longrightarrow_{\beta r}$	`(program ()`
	`(seq (`ϕ^I` (L`β` t (void)))`
	`(`ϕ^I` (L`α` () ((Export a1 a a))`
	`(define-values (a) 10) (var-set! a1 a) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export k1 k k)) ((var-ref k1))) #:t t))))`
		[k → $cell_1$,
		k1 → var_k,
		a1 → var_a]
			[$cell_1$ → closure,
			var_a → uninit,
			var_k → closure,
			t → (linklet-instance
			(a var_a) (k var_k)),
			li → (linklet-instance)]

$\longrightarrow_{\beta p}$	`(program ()`
	`(seq ((void) t)`
	`(`ϕ^I` (L`α` () ((Export a1 a a))`
	`(define-values (a) 10) (var-set! a1 a) (void)) #:t t)`
	`(`ϕ^I` (L`α` () ((Export k1 k k)) ((var-ref k1))) #:t t))))`
		[k → $cell_1$,
		k1 → var_k,
		a1 → var_a]
			[$cell_1$ → closure,
			var_a → uninit,
			var_k → closure,
			t → (linklet-instance
			(a var_a) (k var_k)),
			li → (linklet-instance)]

3.3 Making Pycket an Independent Self-Hosting Racket

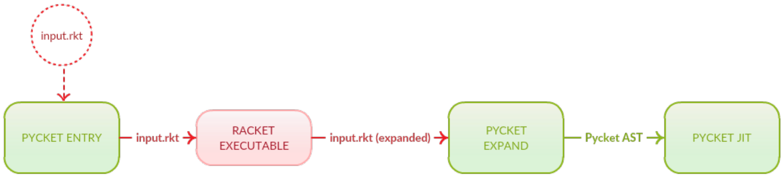


Figure 13: Pycket used to run Racket's executable to fully expand a given module before running it.

Pycket is first designed in 2014 as a high-performance JIT compiler for Racket, generated using the RPython meta-tracing framework [15]. The language interpreter is based on the CEK abstract machine and has the state $\langle e, \rho, \kappa \rangle$ (e : control (program AST), ρ : environment, κ : continuation) [27]. Figure 14 shows the transition rules and how the CEK-loop is implemented in Pycket. The interpreter loop continuously reduces the CEK triple until an empty continuation is reached, which triggers a *Done* exception that returns the results.

In its original design, as shown in Figure 13, Pycket relies on the Racket executable to read and fully expand a given module[46], and then generates RPython AST for it and evaluates it within the interpreter loop [8]. Exploiting the increased portability in Racket, here we turn the Pycket from being a rudimentary compiler to an actual implementation of Racket, independent of the Racket executable.

$$e ::= x \mid \lambda x.\, e \mid e\ e$$
$$\kappa ::= [] \mid \mathsf{arg}(e, \rho)::\kappa \mid \mathsf{fun}(v, \rho)::\kappa$$

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

```
try:
  while True:
    ast, env, cont = ast.interpret(env, cont)
except Done, e:
  return e.values
```

Figure 14: The CEK machine and the loop in Pycket. Figure taken from [8].

[6] Some reduction rules (e.g. closure application etc.) are omitted for space.

First step is building the *linklets* layer by implementing the *compile-linklet* and *instantiate-linklet* functions in RPython, thereby allowing Pycket to process and evaluate linklets. The *compile-linklet* takes an s-expression for a linklet to produce a linklet object as described in Section 3.2.2. It makes several passes on the s-expression to determine the identifiers that are defined within the linklet and the ones that are mutated. Then it processes the imported and exported identifiers to produce the Import and Export run-time objects, freshly generating identifiers (gensym) as needed. Then it makes a final recursive pass to produce the RPython AST for the body, and puts all the information within a linklet object. The *instantiate-linklet* on the other hand, carries out the steps for running a linklet object. First it processes the Import and Export objects and collects and creates variables as necessary, and prepares the target instance (i.e. creates a new instance if it's a regular instantiation) and interprets the body, effectively implementing the $\longrightarrow_{\beta p}$ reduction in the Figure 12 which will be transitively applied by the interpreter loop.

After implementing the *compile-linklet* and *instantiate-linklet*, the next step is to load the bootstrapping linklets into the run-time as described in Section 3.2. For simplicity, we are only concerned with loading the *expander* linklet, as it's the most essential one for bootstrapping (also it's the biggest one). Loading the other linklets are essentially the same process (modulo the run-time support they need).

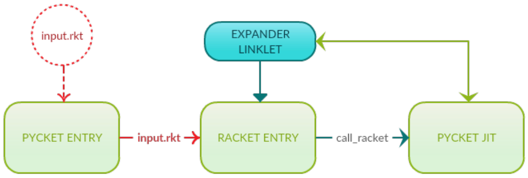


Figure 15: Pycket now uses the functionalities from the expander linklet to expand and run a given module.

As we described in Section 3.2, the expander linklet is generated offline by running the expander on itself using the Racket's executable, and serialized as an s-expression. Pycket, then reads this s-expression and runs `compile-linklet` to produce a linklet object, which is then instantiated using the `instantiate-linklet`, and the functions from the expander linklet is load into the run-time. At this point, Pycket has all the functionalities implemented and exported by the expander, such as read, expand, eval etc., in its run-time to call.

```
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 the expander and all the functions that it provides in its run-time allows Pycket to run a wide variety of Racket programs and easily implement run-time functionalities. For example, a very simple implementation of our running example top-level repl can directly use Racket's read, expand and eval functions, as shown in Figure 16. The **call_racket** is a wrapper that handles the setup to call Racket functions within the CEK loop.

Moreover, since the expander provides the implementations of Racket's macro-system and the module-system, Pycket is able to run programs that are in languages built on top of Racket core, such as *racket/base*. Therefore, instead of implementing the top-level repl itself, Pycket can directly run Racket's own repl, which is implemented as a *racket/base* program, through the `dynamic-require` function exported by the expander.

```
dynamic_require.call_racket(['racket/repl,'read-eval-print-loop])
```

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

Figure 16: A simple top-level REPL via expander functions on Pycket

These programs, including the bootstrapping linklets themselves often require additional run-time support such as *correlated* objects (which are syntax objects without the lexical-content information), Racket-level exception handling and primitive functions. For example, running Racket's repl requires delimited continuations. Also handling exceptions in Racket code in the run-time requires Racket-level exception handling on Pycket. The existing Pycket implementation has a rudimentary exception handling mechanism via transforming the Racket-level exceptions (which are implemented by Racket structs) into the RPython exceptions. However, having Racket-level exceptions requires the run-time to support installing Racket-level handlers via continuation-marks and dynamically pass the Racket-level exceptions to the appropriate handlers.

Moreover, the Racket itself (including the bootstrapping linklets) generally relies on a large number (∼1500) of primitives, aroud 900 of which we already have in the existing Pycket implementation. An additional ∼200 primitives need to be implemented, along with the addition of the run-time liblaray **#%linklet** that will contain 32 linklet related functions including the *compile-linklet*, *instantiate-linklet* and *instance-variable-value*.

3.3.1 On Racket's Portability

The ability to provide high-level functionalities as callable functions to the hosting virual machine is one of the key ideas in improving Racket's portability. A VM that wants to host Racket not only gets the high-level implementations such as the module system, macro system for free, but also gets highly re-usable functionalities that it can integrate the systems it implements, such as the top-level repl example in Figure 16.

Moreover, Racket implements and exports abstract functionalities that are tightly coupled with the hosting run-time as well. For instance the expander linklet provides the `eval` function, which is basically an abstract interpreter for Racket. A VM that hosts Racket can call `eval`, which interprets Racket code by using the run-time primitives and `compile-linklet` and `instantiate-linklet` that are implemented and provided by the VM itself.

Therefore, one of the essential nuances of self-hosting Racket with the expander linklet, is that the interaction between the run-time and the expander is a two-way street. The VM implements and uses `compile-linklet` and `instantiate-linklet` to import the functions provided by the expander, and the expander provides functions that uses the `compile-linklet` and `instantiate-linklet` functions, along with the other run-time primitives as well.

For example, the `dynamic-require` used above (provided by the expander linklet) is a Racket function that dynamically loads a Racket module (if it's not already loaded) by resolving the module path, finding the source code in the file system, reading and expanding the codes and modifying the namespace registry. The Racket code inside the expander that implements all that, calls run-time primitives such as for file-system support and also calls the `compile-linklet` and `instantiate-linklet` for expanding and instantiating all the required modules. `eval` works in a similar fashion as well.

This intertwined nature of the high-level language facilities and the low-level run-time support is central to the Racket's improved portability.

3.4 Performance Issues with Self-Hosting on JIT Compilers

With many micro and macro benchmarks and larger experiments, the existing Pycket implementation is shown to be significantly performant in evaluating Racket code. It benefits from the the existing generic optimizations in the RPython framework, including common subexpression elimination, copy propogation, constant folding, loop invariant code motion, malloc-removal and the inlining that naturally comes for free from tracing [4, 30, 14], as well as from many improvements on the interpreter such as environment pruning, data structure specialization, strategies, as well as some gradual typing related improvements such as the use of hidden-classes. However, it is also shown that in some cases Pycket is significantly slower than all the other systems, on "almost exclusively recursive programs with data-dependent control flow in the style of an interpreter over an AST" [8, 9]. In this section we identify the key points of these issues and propose solution approaches that will be essential in implementing efficient run-times for self-hosting functional langauges on meta-tracing JIT compilers.

Tracing interpreter-style programs with complex control-flow paths is a known weakness of JIT compilers. The large number of indirections not only cripple the JIT optimizations but also causes the loops to be segmented into a large number of highly data driven traces. For example, consider the following program in Figure 17, which is a very simple regular expression matcher. It is highly simplified and some of the rules are removed for space.

Tracing this program running with an input regexp, say #`rx"defg"`, trying to match it against an input string produces a trace that follows the control-flow path of the program for that input, making tracing quite wasteful because for any other input regexp, say #`rx"a*"` which follows an entirely different path on the program, the JIT produces, compiles and optimizes yet another trace for that input, unable to use the previously generated trace. This problem not only increases the warmup time but also produces traces that are unlikely to be frequently re-used.

This problem is immediately observable on Pycket self-hosting Racket through the expander linklet, because the interpreter style programs with complex control-flow paths are quite central in self-hosting a language (e.g. expand, fasl etc.). Additionally, since the Pycket's level of language abstraction is increased one step further, the generated traces are much larger, which creates a pressure for the JIT's compiler and optimizer. As a result, in the run-time the JIT spends a lot of time compiling and optimizing traces, but often bails out and interprets the code instead of using the traces, which defeats the purpose of using a tracing JIT.

Another major actor in this play is the loop invariant code motion optimization performed by the JIT [4]. This optimization prepends a trace to itself, moving all the loop-invariant operations (usually the operations for destructuring the interpreter state) into the preamble, keeping all the essential operations in the peeled-iteration (the inner loop). An example of this can be seen in the trace in Figure 1 in Section 3.1. Any trace or a side-trace (i.e. a *bridge*) that is able to jump to the peeled-iteration of another trace therefore saves time by not performing the loop-invariant operations. In order to do so, however, the program state needs to be consistent with the one that is expected by the peeled-iteration.

The program state consists of heap allocated objects (e.g. environment, continuation), the virtuals (malloc-removed parts of the interpreter state), and other loop information such as range values. In order to jump to a peeled-iteration, the state needs to match with the one that the iteration is expecting, e.g. the same state at the end

```
a      Matches the specified character literal     q      q
*      Matches 0 or more of the previous character a*     "", a, aa, aaa
.      Matches any character literal               .      a, b, c, d, e ...
^      Matches the start of a string               ^c     c, ca, caa, cbb ...

(define (match-star pattern p-pos str s-pos star-pos m)
  (or (and (< s-pos (string-length str)) (char=? (string-ref pattern p-pos) (string-ref str s-pos))
           (match-pat pattern p-pos str (add1 s-pos) (cons (string-ref str s-pos) m)))
       (match-pat pattern (add1 star-pos) str s-pos m)))

(define (match-pat pattern p-pos str s-pos m)
  (cond
    [(>= p-pos (string-length pattern)) m]
    [(and (< (+ p-pos 1) (string-length pattern)) (char=? (string-ref pattern (+ p-pos 1)) #\*))
      (match-star pattern p-pos str s-pos (+ p-pos 1) m)]
    [(char=? (string-ref pattern p-pos) #\.)
      (and (< s-pos (string-length str)) (match-pat pattern (add1 p-pos) str (add1 s-pos) (cons (string-ref str s-pos) m)))]
    [else
     (and (char=? (string-ref pattern p-pos) (string-ref str s-pos))
          (match-pat pattern (add1 p-pos) str (add1 s-pos) (cons (string-ref str s-pos) m)))]))

(define (reg-match pattern str)
  (let ([m (cond
             [(char=? (string-ref pattern 0) #\^) (match-pat pattern 1 str 0 '())]
             [(string=? str "") (match-pat pattern 0 str 0 '())] ; edge case
             [else (for/or ([i (in-range (string-length str))])
               (match-pat pattern 0 str i '()))])])
    (and m (list (list->string (reverse m))))))
```

Figure 17: A simple regular expression matcher (some cases are removed for space).

of its preamble. On the other hand, jumping to the preamble of a trace requires the allocation of all the unallocated parts of the interpreter state. If a bridge that is produced for a side-exit is used to jump back to the original trace, the JIT tries to make it jump to the peeled-iteration whenever possible to avoid performing the loop-invariant operations. However, this often fails on Pycket because the interpreter state often can't match the expected state, because the state is changed differently by different control flow paths (e.g. non-tail calls add a continuation frame). Note that on Pycket the major parts in the interpreter state is composed by the environment and the continuation, which are heap allocated objects.

This issue is partly addressed in the previous versions of Pycket by allowing the JIT to allocate a little bit more to force a match between the states. The malloc-removal and escape analysis in the trace optimizer often allows the JIT to remove parts of state and create virtuals for the objects that never escape the loop [14, 4]. The introduced heuristic on Pycket allows the JIT to allocate such objects elided by the optimizations, trading some space for jumping into the inner loop to avoid performing loop-invariant operations. It is shown that with this heuristic the performance in gradual typing is increased by 4% on Pycket, while adding no extra overhead [9].

Adding the linklets layer to the front-end allows Pycket to run large Racket programs including the expander. To give a perspective, the offline generated expander linklet that is processed in Pycket at boot itself is roughly 5MB file containing the s-expressions of 2642 functions. Another example is, to run a single Racket program written in the *racket/base* language (such as repl for instance), more than 100 Racket modules need to be expanded and instantiated first just to load the language before running the program itself. While Pycket's performance on the micro-benchmarks is not effected by the linklet layer (since nothing has changed in the back-end), these large programs that are now runnable on Pycket aggravate these issues observed before.

To understand the issue better, it is important to first see how the loop detection currently works on Pycket. While for the low-level languages such as in a bytecode interpreter a program counter

```
# start of the trace (preamble)
label(i0, p1, descr="124248176")
i3 = int_add(i0, 1)
p4 = getfield_gc_r(p1, descr="StrMatchContext.string")
i5 = strgetitem(p4, i0)
i7 = int_eq(i5, 100)
guard_false(i7)
i8 = getfield_gc_i(p1, descr="FixedMatchContext.end")
i9 = int_lt(i3, i8)
guard_true(i9)
# peeled-iteration (inner loop)
label(i3, p1, p4, i8, descr="124248256")
i11 = int_add(i3, 1)
i12 = strgetitem(p4, i3)
i14 = int_eq(i12, 100)
guard_false(i14)
i15 = int_lt(i11, i8)
guard_true(i15)
# jump
jump(i11, p1, p4, i8, descr="124248256")
```

```
#lang racket/base

(define l (make-string 4000 #\a))
(define r (make-string 4000 #\p))
(define str-big (string-append l "defg" r))

(define reg-r (regexp "defg"))

(define iter 1000000)

(time
 (for ([i (in-range iter)])
   (regexp-match reg-r str-big)))
```

Figure 18: Trace of Pycket's regexp (in RPython) matching #rx"defg"

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 loops is an application. 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 dynamically generated call-graph 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 [8, 9].

The reason that we chose the regular expression matcher as an example in Figure 17 is that Pycket already has an efficient regexp implementation in RPython to compare against. As a simple experiment to demonstrate the issue, we run both regexp implementations (the one in the Figure 17 and the RPython implementation) to match `#rx"defg"` against a large string containing the sequence `"defg"` in the middle and measure the time performances. Figure 18 shows both the trace we get from the RPython implementation and the program we run. To run the simple implementation in Figure 17, we modify this program to use the `reg-match` in instead of the `regexp-match`. When we run both with the same inputs, we observe 2x slowdown on the simple implementation. Part of the reason is the use of clever techniques that are specific to the regexp implementation on RPython such as caching and using contexts etc. The bigger part of the problem, however, lies in the difference between the traces that are generated for this computation.

The gist of this computation is the literal search in the string for the `"defg"` pattern. As can be seen in Figure 18, this is captured in a tight loop in the main trace for the RPython implementation. The RPython regexp implementation utilizes interpreter hints to control the unrolling and encourages the allocation removal optimizations through type specializations. On the other hand, tThe trace we get for the Racket implementation for the same program is quite large due to aggressive inlining comes with tracing and includes large amounts of allocation/deallocation. For space concerns in this document we defer presenting the large traces to the dissertation.

The main issue here is that any functionality that Pycket now imports and runs as Racket code, normally would’ve been implemented on the language interpreter in RPython and optimized using the interpreter hints that the RPython framework provides for interpreter authors, such as the regexp implementation. However, since Pycket now evaluates Racket code to extend the functionalities of its interpreter, it is unable to use the hints to optimize the implementation on the JIT. In other words, what gives Pycket the extendability at the language level via importing implementations as Racket code also cripples it by preventing the use of the interpreter hints that allow the language interpreter to communicate with the tracing interpreter (i.e. the JIT) to generate optimal traces.

To address this, we propose to develop and use the *meta-hints*. Meta-hints is a generalization of the RPython interpreter hints that extends further the communication between the language being implemented and the tracing-JIT run-time. Recall from Section 3.1, that a language interpreter written in RPython can utilize a `JitDriver` reflection provided by the RPython framework to define its own program counter (pc) and use hints such as `jit_merge_point` and `can_enter_jit` to indicate a starting point for a trace and where the interpreter might perform a backwards jump, respectively. An example can be seen in Figure 2 in Section 3.1. Pycket’s interpreter’s two-state tracking defines the program counter (green variables) as a pair of a lambda form and an application form (call-site) to determine the trace header.

For the `match-pat` function, in its original form in Figure 17 (without the meta-hints), in order for a trace to be formed for one of its looping branches while being evaluated on Pycket, the input needs to go into the same branch and make a recursive jump enough times to make that branch ”hot”. Therefore, until it becomes hot, the tracer continues to trace through the jumps, landing in the same lambda, but passing through the different branches along the way (because the input doesn’t necessarily go into the same branch everytime). By the time that a particular branch becomes hot, the traced instructions often contain codes for the other branches as well, generating a large and convoluted trace.

```
(define/jit-merge-point (match-pat pattern p-pos str s-pos m) #:greens pattern
  (cond
    [(>= p-pos (string-length pattern)) m]
    [(and (< (+ p-pos 1) (string-length pattern)) (char=? (string-ref pattern (+ p-pos 1)) #\*))
      (match-star pattern p-pos str s-pos (+ p-pos 1) m)]
    [(char=? (string-ref pattern p-pos) #\.)
      (and (< s-pos (string-length str))
           (can-enter-jit1 (match-pat pattern (add1 p-pos) str (add1 s-pos) (cons (string-ref str s-pos) m))))]
    [else
     (and (char=? (string-ref pattern p-pos) (string-ref str s-pos))
          (can-enter-jit2 (match-pat pattern (add1 p-pos) str (add1 s-pos) (cons (string-ref str s-pos) m))))]))
```

Figure 19: Regexp dispatch loop with meta-hint annotations.

The meta-hints approach aim to solve this problem in a number of ways. As an example, Figure 19 shows the main dispatch-loop of the regexp implementation in Figure 17. First, it adds additional components to the ”green” variables to extend the definition of pc, thereby contributing to the profiling performed by the language interpreter. For example, if this program was an actual interpreter (as in the case of self-hosting) instead of a regexp implementation, these green variables will be the pc for the language being implemented, combined with the pc of the underlying language interpreter being meta-traced. Additionally, marking the points where a backwards jump may occur at this level (by the `can-enter-jit` no-ops) helps the profiling on Pycket by considering only the marked spots, as opposed to every lambda application.

A more complicated idea here is to use the meta-hints to help reduce the branching effect. Currently the tracer follows the interpreter through the branches and calls and records a sequence of instructions for a hot loop. Note that the input is still a possibly non-repetitive sequence of data, so the tracer will still go in different branches at each step without stopping. Pausing and unpausing the tracer at certain spots would make forming a trace much more complicated, as it could easily invalidate a certain computation path and render the recorded instructions unusable. However, if the tracer is modified to label the instructions as it records, then it might be possible to filter out instructions and compose a trace that only includes the instructions with a certain label, thereby forming a bundle of coherent and concise loops involving only the paths for each branch.

3.4.1 Garbage Collection (GC) Pressure

The second issue that we identified as one of the essential problems in self-hosting Racket on meta-traced CEK is the GC pressure caused by the long chains of continuation allocations on the heap. As we described in Section 3.3, given a Racket module, Pycket now first runs the expander to fully expand the module before running it. Since the expansion of a given Racket module is included in Pycket’s run-time, the JIT suffers from a large number of heap allocations for the continuation chains often caused by the control indirections such as within the macro-system. This, in turn, creates a pressure for the garbage collection, as the GC is triggered by allocations. To understand the issue better, we start by briefly describing the GC utilized by the RPython framework.

As described in Section 3.1, the flow graphs and the C code that are generated during the translation in the RPython framework assume automatic memory management. Therefore the produced program is linked with a GC that is implemented within the framework (in RPython), by inspecting all the graphs and turning all `malloc` operations into calls to the GC. RPython’s garbage collector, namely the *minimark GC* is a three-generations semispace copying generational collector. Each semispace has a nursery where the young[7] objects are inserted first, i.e. allocations fill the nursery. And when the nursery is full then it is collected and the objects that are still alive are moved into the non-nursery part of the current semispace. This decreases the times that a full collection is needed. The approach is based on the idea that the lifetime of the objects that are created are short, and the amount of live objects that are used by the program fits in the nursery (set to be the size of CPU Level 2 cache). [42, 11, 35, 40]

Now that the Racket’s expander is used to expand a given module within the Pycket’s run-time, the life in the heap is much different than what a generational GC would like to see. The allocated objects are rather big and live very long. Even the expander itself is allocated as a linklet instance containing more than 2000 functions, and live during the whole run-time. Another example is the long continuation chains caused by the large depth of the real-world Racket programs such as the macro-system or the module-system. In the *minimark GC* the old and/or big objects are moved out of the semispace (”external”), where they will not be moved anymore and collected in a mark-and-sweep fashion to avoid costly moves. However, since our setup creates many objects (often large) that live long, not only we don’t benefit from the nursery approach enough but also the mark-and-sweep process creates a pressure on the run-time performance, as it is a serial blocking collector.

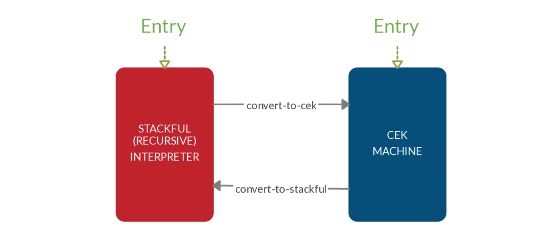


Figure 20: Stackful & CEK Switch

To address this issue (without modifying the underlying framework nor the GC), we propose to use a stackful interpreter to decrease the continuation allocations on the heap. The stackful interpreter here is a simple recursive interpreter for evaluating Racket forms, which is translated by the RPython framework into a recursive C program that uses the native stack. Recall that, however, the Pycket’s interpreter is based on the CEK machine, and the heap allocated explicit continuations makes it very convenient to implement complex control operations with continuations (as well as the proper handling of tail-calls). Therefore the proposed approach is to use a stackful interpreter *alongside* the CEK, rather than replacing the current CEK with a stackful interpreter. The idea is to use the stack and heap in balance to decrease the GC pressure and take advantage of using the stack, such as rapid allocations, locality etc.

[7] The age of an object is the number of collections they survived.

Using the stack to implement a language like Racket, however, understandibly introduces some challenges regarding the stack management, such as handling overflows due to arbitrarily deep recursion. Additionally, implementing a stackful interpreter on Pyket for the entire Racket would be illogical, since it would not only require implementing everything from scratch including the continuations, but also not benefit from the current CEK's performance. Therefore the logical approach is to use the two interpreters together, repeatedly switch between the two whenever appropriate, making sure that the switches don't induce a high performance overhead.

One of the most fundamental points here is the interaction between the two interpreters. Essentially, we want to use the stack to save some heap space (thereby reducing the GC pressure) and take advantage of faster allocations and locality. On the other hand, we don't want to lose the benefits of the JIT's performance optimizations on the CEK. Moreover, the stack management needs to be on point to avoid problems such as switching back to the CEK when the stack is very deep (e.g. stack overflow), since that would require allocating enough continuations on the heap to counterbalance the performance benefit we get from using the stackful interpreter.

$$
\begin{aligned}
e ::= {} & x \mid v \mid (e\ e\ \ldots) \mid (\textbf{if}\ e\ e\ e) \mid (o\ e\ e) \\
& \mid (\textbf{begin}\ e\ e\ \ldots) \mid (\textbf{lambda}\ (x\_!\_\ \ldots)\ e) \\
& \mid (\textbf{set!}\ x\ e) \mid (\textbf{raises}\ e) \\
& \mid (\textbf{let-values}\ (((x)\ e)\ \ldots)\ e) \\
& \mid (\textbf{convert-to-stackful}\ e) \\
& \mid (\textbf{convert-to-cek}\ e)
\end{aligned}
$$

Figure 21: Source Language for Stackful & CEK Models

To understand this problem better and to work on a simpler setup, I developed a formalism that includes both the CEK machine and the stackful interpreter, evaluating a very simple subset of Racket together, shown in Figure 21, which is similar to the Racket Core language used in Section 3.2.2 minus the forms for the linklet variables, plus some `convert` forms for controlling the interpreter switches from the source level. Figure 20 shows the overview of the interaction between the two models. We can start with either of the interpreters and switch back and forth between them using the `convert` forms manually within the source code.

```
(eval-stackful (let-values (((a) 3)) (convert-to-cek (let-values (((b) 1)) (+ a b)))))
(eval-cek (let-values (((a) 3)) (begin (convert-to-stackful (set! a 5)) a)))
```

In addition the formalism, in order to see how well this would perform in action, we also prototyped an implementation on Pycket. It works in a similar way to the model in Figure 20, except the starting point is the CEK, because Pycket needs to instantiate and import the bootstrapping linklets at the startup and start a computation by loading and running the program via the `read`, `expand` etc.

The CEK starts the computation and handles the control to the stackful whenever it encounters a `let` or a `begin` form. The switch happens by just calling the stackful interpreter. The stackful interpreter includes a trampoline to handle the tail-calls, and in the current implementation it automatically switches back to the CEK whenever **i)** it returns, or **ii)** a CPSed primitive (e.g. a higher order function) is used, or **iii)** a stack overflow is detected. The switch back to the CEK happens by rewinding the stack via a `ConvertStack` exception in which the forms in each activation record installs a continuation. The CEK takes the control and all the information at the previous switch-to-stack point and continues from there.

The main investigation here is about the increase in the overall performance with the following constraints: **i)** The number of points where a switch occurs between the two interpreters and the switching overhead should both be minimal, and **ii)** the amount of copying activation records from the stack into the continuations in the heap should be bounded. Since we're interested in the performance on self-hosting Racket on Pycket, we are still interested in interpreter-style dispatch loops. Therefore as a preliminary experiment, we synthesized a program by simplifying the Racket's `fasl->s-exp` function from the FASL[8] library, which deserializes a given byte stream into an s-expression. Figure 22 shows the simplified program, which takes a list of numbers and takes different branches depending on the number. Note that some of the recursive calls are not tail-calls, i.e. installs a continuation every time it is evaluated.

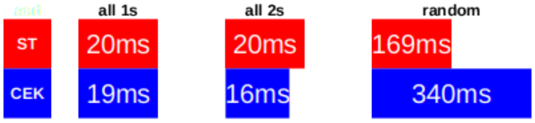
```
(define (branchy lst)
  (letrec
    ([loop
      (lambda (lst)
        (let ([index (if (null? lst) 3 (car lst))])
          (if (null? lst)
              -1
              (if (< index 3)
                  (if (< index 2)
                      (loop (cdr lst))
                      (+ 3 (loop (cdr lst))))
                  (if (< index 5)
                      (if (< index 4)
                          (loop (cdr lst))
                          (+ 5 (loop (cdr lst))))
                      (if (< index 6)
                          (loop (cdr lst))
                          (+ 1 (loop (cdr lst)))))))))])
   (loop lst)))
```

Figure 22: A simple program with non-trivial control flow.

We ran this program on Pycket using Stackful+CEK and only-CEK with different auto-generated inputs containing; only 1s, only 2s, and random [0-7], each having 1000 and 2000 numbers. The runtimes for the ones with only 1s

[8] fast-load serialization

and 2s are quite similar, except a larger warmup time for the stackful interpreter, which is expected since RPython's meta-tracing is not designed for recursive interpreters, therefore it needs to be further optimized. However, we observe a consistent increase in the run-time performance in the randomized experiments (both 1000 and 2000) which visits different branches. Below are the medians for the run-times for the experiments using 1000 numbers.



Because of the limitations in the variety of programs that the stackful interpreter can currently run (without large amounts of switching), I seek to further characterize the performance of the stackful evaluation along the CEK interpreter especially on programs with heavy use of continuations.

Taking a step towards solving the fundamental issues in self-hosting on meta-tracing, namely, intelligent meta-tracing of the programs with complex control-flow and proper handling of the heap pressure, will allow efficient self-hosting of Racket on meta-tracing Pycket. We believe these are broadly applicable solutions that will serve as principle approaches in self-hosting on meta-tracing frameworks in general.

4 Related Work

Higher-order dynamic VMs Dynamic VM implementations are becoming increasingly popular for their rapid prototyping potential, as the effort required to implement a new VM from scratch is often quite large. Therefore, instead of manually implementing a VM in a low-level language such as C, it is often argued that building on top of an existing object-oriented general-purpose VMs or dynamic integration via generating a VM using a ”specification” of a language allows easier and more maintainable implementations with competitive performance [12]. One of the major actors for the former approach is the *GraalVM*, which is a modified version of the Java HotSpot VM on the JVM (Java Virtual Machine). GraalVM uses a language implementation framework called Truffle, and a method-based JIT compiler called Graal to implement VMs on the JVM for dynamic languages such as Javascript, Ruby and Python [49]. As opposed to building on top of VM, the *RPython* project introduced the idea of automatically generating a VM from a language specification represented as an interpreter via meta-tracing [2]. For example, as mentioned in the introduction, PyPy is an implementation of Python that is built on the RPython meta-tracing framework that generated a VM including a tracing JIT for Python with a better performance than the CPython itself. In addition to the RPython framework, there are other meta-tracing systems as well, such as the SPUR, a tracing JIT compiler for CIL bytecode [38], meta-tracing languages that are implemented in C# [10].

Tracing JIT VMs Like PyPy, Pycket is built on the RPython meta-tracing framework as well, and this study tries to identify approaches on achieving efficient self-hosting on Pycket running a tracing JIT. Therefore it would be relevant to mention here some notable tracing JIT VMs too. Initially introduced by the Dynamo project [7], trace-based compilation is successfully utilized by many VMs including some commercial VMs such as Mozilla's TraceMonkey JavaScript VM [31], Adobe's Tamarin ActionScript VM [21], as well as some research VMs such as LuaJIT [39], Converge [47], Lambdamachine [43] and PyHaskell [45].

Optimizing VM performance in a meta-tracing context As we mentioned in Section 3.1, there are a lot of dynamic optimizations performed by the RPython back-end that the language interpreter should be in sync with. There are two main approaches for a VM author to optimize such a system: **i)** improve the interpreter performance, **ii)** modify the interpreter to produce traces that are easier to optimize by JIT. Both of these approaches operate on the interpreter, therefore the improvements should focus on the common patterns that are specific to the language being interpreted. For example, along with the general JIT improvements such as value promotion and edlibale functions, PyPy and Converge both focus on optimizing the objects, classes and modules [13].

Self-hosting Similarly, Bolz and contributors state that in general-purpose VMs, the compiler and the language being implemented should be semantically in sync to achieve a good performance [12, 16]. As we discussed in Section 3.4, we study the relationship between the semantics of the programs common to self-hosting and the interpreter & trace performance. Unfortunately, to the best of our knowledge, no studies of self-hosting on meta-tracing has been done so far. However, there are self-hosting VMs for dynamic languages, such as the Tachyon VM for JavaScript, which is a bootstrapping JIT compiler that is completely hand-coded in JavaScript and doesn't use any automatic generation techniques [22]. Although Tachyon is quite different than Pycket and bootstraps the VM itself, it uses control-flow graphs in SSA form as an intermediate representation (IR), similar to Pycket. Moreover, it uses interesting techniques that might prove useful for Pycket as well, such as conditional constant propogation via abstract interpretation [48]. Additionally, there are low-level techniques to deal with control-flow issues on VMs caused by large amount of branches, such as indirect branch prediction [24], which might be adapted to be useful in tracing as well. However, in our case implementing such techniques would require modifying the underlying RPython framework, where our approach is at the language (interpreter) level.

Managing abstraction levels A big part of the problems we tackle in this thesis is caused by the fact that adding self-hosting on Pycket adds another level of abstraction that needs to be semantically in-sync with the run-time meta-tracing VM. There has been related research studying to reduce complexity in systems involving multiple levels of abstractions, such as type-directed partial evaluation [28] and multi-stage programming [44, 32]. Amin and Rompf studied how to collapse a tower of interpreters interpreting each-other into a compiler to remove the interpretation overhead [1]. They developed a multi-level lambda calculus and a meta-circular evaluator, namely Pink, to demonstrate the collapse of arbitrarily many levels of self-interpretation.

Another work that is relevant to our investigation is optimizing hierarchically layered VMs using trace compilation, researched by Yermolovich and contributors [50]. Similar to the RPython framework, they propose to run a guest VM (Lua VM [39]) on top of a host VM with a trace-based JIT compiler (Tamarin [21]). Along with optimizing the guest VM, the host VM also allows the guest VM to provide hints about its interpreter loop that can specify which parts of the guest VM should not be traced for its own, thereby taking into account the guest VMs workload as well.

Space concerns & heap allocated continuations Many studies investigate the space complexity and optimizations aiming to increase performance. Some use *repurposed* JIT compilers (RJIT) to benefit from the optimizations that the dynamic languages enable, such simplification of control-flow graphs [34] and run-time type feedbacks [19] in the context of adding support for dynamic languages to an existing static-language VMs. Others use higher-level techniques such as hidden-classes in Pycket [9], similar to the maps introduced by Self [20] and used by PyPy [16].

As we discussed in the performance section, one of the big concerns we identified in self-hosting is the garbage collection overhead caused by the long-chains of heap allocated continuations. Allocating activation records on the heap is studied extensively in the context of compiling with [3] or without [37] continuations [23, 25]. With that said, perhaps the most relevant work to our investigation with the stackful and CEK interpreters working together is by Hieb and contributors [33], where they provide an approach to allocate activation records on the stack that doesn't require the stack to be copied when a continuation is created, thereby establishing an upper bound on the amount of copying which is otherwise unbounded. They also provide techniques for stack overflow/underflow recovery, which is isomorphic to continuation creation and re-instantiation.

To the best of our knowledge, no comprehensive synthesis of issues and opportunities of self-hosting on a meta-tracing VM has been done. Therefore a head-to-head comparison of Pycket with such a system is impossible. However, existing Racket implementations such as Racket's previous (generic JIT-based) and current (Chez [29]) run-time are suitable for performance comparison with Pycket implementing Racket.

5 Research Plan and Timeline

I have already made progress towards my thesis:

- I have created the formalism for describing the operational semantics of linklets.
- I have implemented the linklets in RPython and incorporated them into Pycket.
- I have implemented all the run-time support and primitives to incorporate the expander into Pycket and be able to implement `#racket`.
- I have created a formalism for a stackful and a CEK-based interpreter working together.
- I have prototyped the two interpreter approach where a stackful interpreter is working along with the existing CEK that is hosting Racket using linklets.

To complete my thesis, I plan to follow this timeline:

- **[December 2019]** Improve the stackful+CEK Pycket.
  - Improve the stackful interpreter enough to run more complex programs with linklets.
  - Investigate and improve the strategies for switching with the CEK.
  - Devise and implement experiments to show that it is a viable approach by demonstrating that it helps with the GC issue.
- **[January 2018]** Improve the performance of regexp implementation.

  - Implement the meta-hints approach to get the performance of the regexp implementation closer to the existing RPython regexp.
  - Plan and execute experiments to demonstrate that it is a viable approach to meta-trace programs with complex control flow, including Racket's expander etc.

- **[February-May 2020]** Write dissertation
- **[June 2020]** Defend

I plan to publish the following papers:

- **2020** Extension of *"Rebuilding Racket on Chez Scheme (Experience Report)"* [29] for JFP.

References

[1] Nada Amin and Tiark Rompf. "Collapsing Towers of Interpreters". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 52:1–52:33. ISSN: 2475-1421. DOI: 10.1145/3158140. URL: http://doi.acm.org/10.1145/3158140.

[2] Davide Ancona et al. "RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages". In: Jan. 2007. DOI: 10.1145/1297081.1297091.

[3] Andrew W. Appel. *Compiling with Continuations.* New York, NY, USA: Cambridge University Press, 2007. ISBN: 052103311X.

[4] Hkan Ard, Carl Friedrich Bolz, and Maciej FijaBkowski. "Loop-aware optimizations in PyPy's tracing JIT". en. In: *Proceedings of the 8th symposium on Dynamic languages - DLS '12.* Tucson, Arizona, USA: ACM Press, 2012, p. 63. ISBN: 978-1-4503-1564-7. DOI: 10.1145/2384577.2384586. URL: http://dl.acm.org/citation.cfm?doid=2384577.2384586 (visited on 03/26/2019).

[5] M. Arnold et al. "A Survey of Adaptive Optimization in Virtual Machines". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 449–466. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840305.

[6] John Aycock. "A Brief History of Just-in-time". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077. URL: http://doi.acm.org/10.1145/857076.857077.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* PLDI '00. event-place: Vancouver, British Columbia, Canada. New York, NY, USA: ACM, 2000, pp. 1–12. ISBN: 978-1-58113-199-4. DOI: 10.1145/349299.349303. URL: http://doi.acm.org/10.1145/349299.349303 (visited on 03/22/2019).

[8] Spenser Bauman et al. "Pycket: A Tracing JIT for a Functional Language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming.* ICFP 2015. event-place: Vancouver, BC, Canada. New York, NY, USA: ACM, 2015, pp. 22–34. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784740. URL: http://doi.acm.org/10.1145/2784731.2784740.

[9] Spenser Bauman et al. "Sound Gradual Typing: Only Mostly Dead". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 54:1–54:24. ISSN: 2475-1421. DOI: 10.1145/3133878. URL: http://doi.acm.org/10.1145/3133878 (visited on 03/22/2019).

[10] Michael Bebenita et al. "SPUR: a trace-based JIT compiler for CIL". en. In: (), p. 18.

[11] Carl Friedrich Bolz. "Meta-tracing just-in-time compilation for RPython". PhD thesis. Heinrich Heine University Düsseldorf, 2014.

[12] Carl Friedrich Bolz and Armin Rigo. "How to not write Virtual Machines for Dynamic Languages". en. In: (), p. 11.

[13] Carl Friedrich Bolz and Laurence Tratt. "The Impact of Meta-tracing on VM Design and Implementation". In: *Sci. Comput. Program.* 98.P3 (Feb. 2015), pp. 408–421. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.02.001. URL: http://dx.doi.org/10.1016/j.scico.2013.02.001.

[14] 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.* PEPM '11. event-place: Austin, Texas, USA. New York, NY, USA: ACM, 2011, pp. 43–52. ISBN: 978-1-4503-0485-6. DOI: 10.1145/1929501.1929508. URL: http://doi.acm.org/10.1145/1929501.1929508 (visited on 11/16/2019).

[15] Carl Friedrich Bolz et al. "Meta-tracing makes a fast Racket". en. In: (), p. 7.

[16] 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.* ICOOOLPS ’11. Lancaster, United Kingdom: ACM, 2011, 9:1–9:8. ISBN: 978-1-4503-0894-6. DOI: 10.1145/2069172.2069181. URL: http://doi.acm.org/10.1145/2069172.2069181.

[17] 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.* ICOOOLPS ’09. event-place: Genova, Italy. New York, NY, USA: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827. URL: http://doi.acm.org/10.1145/1565824.1565827.

[18] Camillo Bruni and Toon Verwaest. “PyGirl: Generating Whole-System VMs from high-level models using PyPy”. In: vol. 33. June 2009, pp. 328–347. DOI: 10.1007/978-3-642-02571-6_19.

[19] 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. 2012), pp. 195–212. ISSN: 0362-1340. DOI: 10.1145/2398857.2384631. URL: http://doi.acm.org/10.1145/2398857.2384631.

[20] C. Chambers, D. Ungar, and E. Lee. “An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications.* OOPSLA ’89. New Orleans, Louisiana, USA: ACM, 1989, pp. 49–70. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74884. URL: http://doi.acm.org/10.1145/74877.74884.

[21] Mason Chang et al. “Tracing for web 3.0: Trace compilation for the next generation web applications”. In: Jan. 2009, pp. 71–80. DOI: 10.1145/1508293.1508304.

[22] Maxime Chevalier-Boisvert et al. “Bootstrapping a Self-hosted Research Virtual Machine for JavaScript: An Experience Report”. In: *SIGPLAN Not.* 47.2 (Oct. 2011), pp. 61–72. ISSN: 0362-1340. DOI: 10.1145/2168696.2047858. URL: http://doi.acm.org/10.1145/2168696.2047858.

[23] Youyou Cong et al. “Compiling with Continuations, or Without? Whatever.” In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019), 79:1–79:28. ISSN: 2475-1421. DOI: 10.1145/3341643. URL: http://doi.acm.org/10.1145/3341643.

[24] M. Anton Ertl and David Gregg. “Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters”. In: *SIGPLAN Not.* 38.5 (May 2003), pp. 278–288. ISSN: 0362-1340. DOI: 10.1145/780822.781162. URL: http://doi.acm.org/10.1145/780822.781162.

[25] Kavon Farvardin and John H. Reppy. “Compiling with Continuations and LLVM”. In: *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016.* 2016, pp. 131–142. DOI: 10.4204/EPTCS.285.5. URL: https://doi.org/10.4204/EPTCS.285.5.

[26] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex.* Mit Press, 2009.

[27] Matthias Felleisen and Daniel P. Friedman. “Control operators, the SECD-machine, and the λ-calculus”. In: *Formal Description of Programming Concepts.* 1987.

[28] 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. London, UK, UK: Springer-Verlag, 1999, pp. 378–395. ISBN: 3-540-66540-4. URL: http://dl.acm.org/citation.cfm?id=645815.668894.

[29] Matthew Flatt et al. “Rebuilding Racket on Chez Scheme (Experience Report)”. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019), 78:1–78:15. ISSN: 2475-1421. DOI: 10.1145/3341642. URL: http://doi.acm.org/10.1145/3341642.

[30] 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.* VEE ’06. event-place: Ottawa, Ontario, Canada. New York, NY, USA: ACM, 2006, pp. 144–153. ISBN: 978-1-59593-332-4. DOI: 10.1145/1134760.1134780. URL: http://doi.acm.org/10.1145/1134760.1134780 (visited on 11/16/2019).

[31] 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.* PLDI ’09. event-place: Dublin, Ireland. New York, NY, USA: ACM, 2009, pp. 465–478. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542528. URL: http://doi.acm.org/10.1145/1542476.1542528 (visited on 11/28/2019).

[32] 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. Vancouver, BC, Canada: ACM, 2017, pp. 2–14. ISBN: 978-1-4503-5524-7. DOI: 10.1145/3136040.3136047. URL: http://doi.acm.org/10.1145/3136040.3136047.

[33] 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.* PLDI '90. White Plains, New York, USA: ACM, 1990, pp. 66–77. ISBN: 0-89791-364-7. DOI: 10.1145/93542.93554. URL: http://doi.acm.org/10.1145/93542.93554.

[34] Kazuaki Ishizaki et al. "Adding dynamically-typed language support to a statically-typed language compiler: Performance evaluation, analysis, and tradeoffs". In: *ACM SIGPLAN Notices* 47 (Sept. 2012), p. 169. DOI: 10.1145/2365864.2151047.

[35] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management.* Chapman and Hall/CRC, 2016.

[36] Casey Klein and Robert Bruce Findler. "Randomized testing in PLT Redex". In: *ACM SIGPLAN Workshop on Scheme and Functional Programming.* 2009.

[37] Luke Maurer et al. "Compiling Without Continuations". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 482–494. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062380. URL: http://doi.acm.org/10.1145/3062341.3062380.

[38] James S Miller and Susann Ragsdale. *The common language infrastructure annotated standard.* Addison-Wesley Professional, 2004.

[39] Mike Pall. "The luajit project". In: *Web site: http://luajit. org* (2008).

[40] Thomas Perl. "Python Garbage Collector Implementations CPython, PyPy and GaS". In: (2012).

[41] Benjamin Peterson, A Brown, and G Wilson. "PyPy". In: *The Architecture of Open Source Applications* 2 (2008), pp. 279–290.

[42] 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.* OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 944–953. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176753. URL: http://doi.acm.org/10.1145/1176617.1176753.

[43] Thomas Schilling. "Trace-based Just-in-time Compilation for Lazy Functional Programming Languages". en. In: (), p. 213.

[44] Walid Taha and Tim Sheard. "Multi-stage Programming with Explicit Annotations". In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation.* PEPM '97. Amsterdam, The Netherlands: ACM, 1997, pp. 203–217. ISBN: 0-89791-917-3. DOI: 10.1145/258993.259019. URL: http://doi.acm.org/10.1145/258993.259019.

[45] Wiik Thomassen. "Trace-based just-intime compiler for Haskell with RPython". In: 2013.

[46] Sam Tobin-Hochstadt et al. "Languages As Libraries". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '11. event-place: San Jose, California, USA. New York, NY, USA: ACM, 2011, pp. 132–141. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993514. URL: http://doi.acm.org/10.1145/1993498.1993514 (visited on 11/16/2019).

[47] Laurence Tratt. "Compile-time Meta-programming in a Dynamically Typed OO Language". In: *Proceedings of the 2005 Symposium on Dynamic Languages.* DLS '05. San Diego, California: ACM, 2005, pp. 49–63. DOI: 10.1145/1146841.1146846. URL: http://doi.acm.org/10.1145/1146841.1146846.

[48] 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. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.

[49] Thomas Wrthinger et al. "One VM to rule them all". en. In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! '13.* Indianapolis, Indiana, USA: ACM Press, 2013, pp. 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581. URL: http://dl.acm.org/citation.cfm?doid=2509578.2509581 (visited on 11/25/2019).

[50] Alexander Yermolovich, Christian Wimmer, and Michael Franz. "Optimization of Dynamic Languages Using Hierarchical Layering of Virtual Machines". In: *Proceedings of the 5th Symposium on Dynamic Languages.* DLS '09. Orlando, Florida, USA: ACM, 2009, pp. 79–88. ISBN: 978-1-60558-769-1. DOI: 10.1145/1640134.1640147. URL: http://doi.acm.org/10.1145/1640134.1640147.

[51] *You lose more when slow than you gain when fast.* https://blog.mozilla.org/nnethercote/2011/05/31/you-lose-more-when-slow-than-you-gain-when-fast/. Accessed: 2019-11-07.