Ongoing Thesis Writeup

Michael Buch

April 15, 2019

Abstract

Intuitively towers of interpreters are a program architecture by which sequences of interpreters interpret each other and a user program is evaluated at the end of this chain. While one can imagine such construct in everyday applications, prior research made use of towers of interpreters as a foundation to model reflection. As such, towers of interpreters in literature are synonymous with reflective towers and provide a tractable method with which to reason about reflection and design reflective languages. As a result, the assumptions and constraints that govern tower models make them unapplicable to practical or non-functional settings. Prior formalizations of reflective towers have identified partial evaluation and reflection to harmonize in the development of such towers. We lift several restrictions of reflective towers including reflectivity, meta-circularity and homogeneity of data representation and then construct non-reflective towers of interpreters to explore how partial evaluation techniques can be used to effectively remove levels of interpretation within such systems.

1 An Introduction to Towers of Interpreters

1.1 3-LISP, Meta-circularity and Reflection

In his proposal for a language extension to Lisp called 3-LISP [1], Smith introduces the notion of a reflective system, a system that is able to reason about itself. The treatment of programs as data is a core concept in the Lisp family of languages and enables convenient implementations of Lisp interpreters written in Lisp themselves. These are known as *meta-circular* interpreters. Smith argued that there is a distinction between reflection and the ability to reference programs as just ordinary values. Reflection requires a way with which an embedded language can access the structures and state of the process that the embedding lives in. Crucial is the idea of implicit as opposed to explicit information that an evaluator exposes. Smith's idea of reflection is the capability to explicitly instantiate a language construct that was implicit prior. While environment and continuations, which form the state of a tradidional Lisp process, are implicitly passed around, a 3-LISP program can access both these structures explicitly at any point in time. 3-LISP achieves this by way of a, conceptually infinite, *reflective tower*. Smith divides a process into two parts: a *structural field* that consists of a program with accompanying data structures and an evaluator that acts on the structural field. Replacing the evaluator with a meta-circular one then provides a way to construct an infinite reflective tower. In Smith's original model, meta-circular interpreters each with its own environment and structural field, included a builtin *reflective procedure*, which when called provided access to the state, i.e. environment, of it's interpreter. A meta-interpreter, also referred to as "the ultimate machine" is the upper-most interpreter in a tower and is itself not evaluated but simply a necessesity for the tower to exist in the first place. Questions of performance, potential uses and a complete model separate from implementation details was provided in subsequent work.

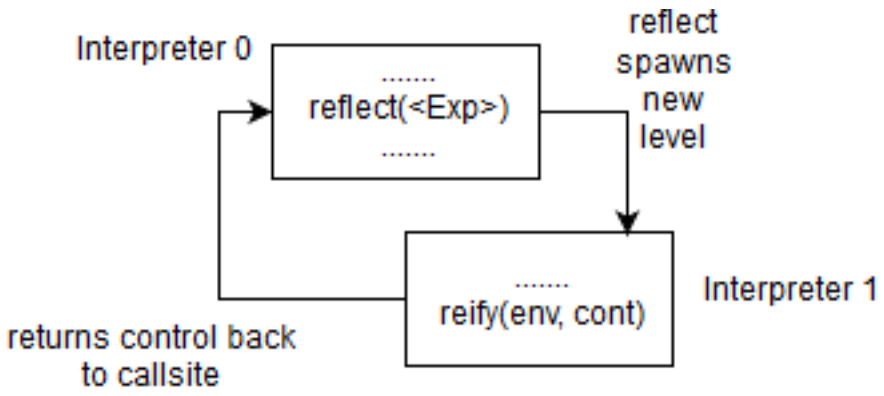

Figure 1: Demonstration of reflection and reification in a two-level tower.

1.2 A Formal Account

Friedman et al. took Smith's account of reflection and decomposed it into distinct operations: *reification* and reflection [2, 3]. The authors provide one of the early denotational descriptions of reflective towers assuming a model where an interpreter is a *valuation function* ($\mathcal{E}$) that maps an expression (Exp), environment (E) and continuation (K) to a domain of answers (A) and the input values to the interpreter bind to ϵ, ρ, κ respectively:

$$\mathcal{E} : Exp \to Env \to K \to A = \lambda\epsilon\rho\kappa.... \tag{1}$$

An open question was how Smith's meta-interpreter [4] could be described in a cohesive framework that relates different levels within a tower through formal semantics. The need for a formalization of reflection was also that earlier work described reflection in terms of reflective towers but which themselves were explained through the use of reflection. To part from such circular reasoning, Friedman et al. defined levels within a tower and their mutual interaction through operators on the state of the valuation function in equation 1 [2]. Reflection operators take values for expression, environment and continuation and re-install them into the interpreter state. Reification operators provide access to the interpreter state and pass it to the program as values. A packaged up state of ϵ, ρ, and κ that can be treated as regular values is said to be *reified*. In the context of multiple-levels of interpretation in a tower, calling a reflection operator spawns a new level in the tower with the interpreter state being the one at the time of application. Once evaluation was performed in the new level control is passed back to the interpreter that spawned it. Reification operators package up the state of the interpreter at the level of application and pass it to the expression to be evaluated. Diagramatically this process is shown in figure 1.

A subsequent study due to Danvy et al. [5] provides a systematic approach to constructing reflective towers. The authors provide a denotational semantic account of their reflection model similar to the technique described above and realize these formalizations into a language built with a reflective tower called "Blond". The authors start with a non-reflective tower and non-meta-circular tower. An assumption that the authors carry throughout their paper is that of single-threadedness. This is both to reduce the complexity of designing an implementation and prevents racey side-effects between concurrent towers. The restriction is that the effects of each level in a tower is the interpretation of the level below it. Any non-interpretative work is performed at the last level of the tower, also referred to as its *edge*. Danvy et al.'s key insight was the need for an intensional description of an interpretative tower that relates the interpreter state at different levels of a tower to the reflection and reification operations. To gain a a better intuition for their description, we summarize Danvy et al.'s model of reflection and reification below.

Adding supscripts to the valuation function from equation 1 and providing domains for components of the interpreter's state, we get:

$$\rho_n \in Environment_n = Identifier_n \to DenotableValue_n \tag{2}$$
$$\kappa_n \in Continuation_n = DenotableValue_n \to Answer_n \tag{3}$$
$$\mathcal{E}_n : Expression_n \times Environment_n \times Continuation_n \to Answer_n \tag{4}$$
$$\mathcal{E}_n[\![interpreter_n]\!]\rho_n\kappa_n \simeq \mathcal{E}_{n-1} \tag{5}$$

A $DenotableValue_n$ is any valid language construct and its representation as defined by the interpreter at level n. A consequence of this formulation is the fact that domains between levels are distinct but connected via the valuation function in equation 5 and formalizes the earlier notion of an interpreter at level n spawning a new evaluator at $n-1$ through some reflective operation. An even more relevant fact is that according to this denotational model, we are free to choose the representation of denotable values in each level. The authors assume for the rest of their study that levels are identical, however, in our work we assume the exact opposite. None of our levels are identical but can be formulated in the same framework given above. An example would be the denotation of an expression $Exp_0 = (1 + 2)$ at level 0 in our hypothetical tower of n levels. At level $n = 1$ this can be represented as $Exp_1 = (+\ 1\ 2)_1$ or at level $n = 14$ as $Exp_{14} = (01 + 10)_{14}$, i.e. in binary. In our model we not only keep the notion of non-identical levels and non-metacirularity, but also the concept of a store, which the authors purposefully ommitted to keep the description purely functional.

$$\begin{aligned}\mathcal{E}_n[\![((reify\ (e\ r\ k)\ body)\ E*)]\!]\rho_n\kappa_n \\ = \mathcal{E}_{n+1}[\![body]\!]([\![e]\!] \mapsto (map_n\ \hat{exp}_n[\![E*]\!]) \\ [\![r]\!] \mapsto (\hat{env}_n\rho_n) \\ [\![k]\!] \mapsto (\hat{cont}_n\kappa_n)]\rho_{n+1})\kappa_{n+1}\end{aligned} \tag{6}$$

Equation 6 describes the effect of a reification operation, here *reify(e r k)* where e, r and k are the variables that are bound to the expression, environment and continuation of the upper level respectively, between a level n and the level above (i.e. its interpreter) $n + 1$.

$$\begin{aligned}\mathcal{E}_n[\![(reflect(E\ R\ K))]\!]\rho_n\kappa_n \\ = \mathcal{E}_n[\![E]\!]\rho_n(\lambda\, a.\mathcal{E}_n[\![R]\!]\rho_n \\ (\lambda\, b.\mathcal{E}_n[\![K]\!]\rho_n \\ (\lambda\, c.\mathcal{E}_{n-1}(\check{exp}_n\ a)(\check{env}_n\ b)(\check{cont}_n\ c))))\end{aligned} \tag{7}$$

The reflection operation in relation between two levels, n and the interpreter it interprets $n - 1$, is shown in equation 7. The domains for individual reflection and reification operations (superscripted with $\hat{}$ and $\check{}$ respectively),

are given equation 8

$$
\begin{aligned}
exp_n^{\wedge} &: DenotableValue_n \to Expression_{n-1} \cup error \\
env_n^{\wedge} &: DenotableValue_n \to Environment_{n-1} \cup error \\
cont_n^{\wedge} &: DenotableValue_n \to Continuation_{n-1} \cup error \\
exp_n^{\check{}} &: Expression_n \to DenotableValue_{n+1} \\
env_n^{\check{}} &: Environment_n \to DenotableValue_{n+1} \\
cont_n^{\check{}} &: Continuation_n \to DenotableValue_{n+1}
\end{aligned}
\tag{8}
$$

An important observation is that in this model, reification and reflection operators are not commutative. As an example reifying a continuation at level n, $cont_n$, followed by reflecting the continuation in level $n+1$ does not yield the same domain when called in reversed order: $con\check{t}_{n+1} \circ \hat{cont}_n \neq \hat{cont}_n \circ con\check{t}_{n+1}$. The expression types given by equations 8 let us explain this trivially by substituting the domains into the inequality.

A less formal explanation of this statement is in terms of the possible values reflection versus reification can result in. Reflection spawns a new interpreter that can yield any result, including an error. Then reifying an error would not yield a valid interpreter state. If one reflects a reified expression it by definition corresponds to simple evaluation in the current interpreter. The importance of this is that this restricts us from being able to fully explain a reflective tower model and valuation functions that act on a level below, $\mathcal{E}_n^{-1}$. If we are not able to provide a definition for reflection and reification at interpreters below any level n we will not have a full description of a reflective tower. This gives a denotational account for the metacontinuations that 3-LISP originally introduced. It was to deal with exaclty this discrepancy in the compositionality of reflection and reification operations. Danvy et al. then add meta-continuations into the equations previously described and their purpose is to describe the continuation that accepts the result of the interpreter it interprets.

1.3 Heterogeneity

A central part of our study revolves around the notion of heterogeneous towers. Prior work on towers of interpreters that inspired some these concepts includes Sturdy's work on the Platypus language framework that provided a mixed-language interpreter built from a reflective tower [6], Jones et al.'s Mix partial evaluator [7] in which systems consisting of multiple levels of interpreters could be partially evaluated and Amin et al.'s study of collapsing towers of interpreters in which the authors present a technique for turning systems of meta-circular interpreters into one-pass compilers. We continue from where the latter left of, namely the question of how one might achieve the effect of compiling multiple interpreters in heterogeneous settings. Our definition of *heterogeneous* is as follows:

Definition 1.1. Towers of interpreters are systems of interpreters, $I_0, I_1, ..., I_n$ where $n \in \mathbb{R}_{\geq 0}$ and I_n determines an interpreter at level n interpreted by I_{n-1}, written in language L such that L_{I_n} is the language interpreter I_n is written in.

A level here is analogous to an instance of an interpreter within the tower and as such level n implies I_n if not mentioned explicitly otherwise.

Definition 1.2. Heterogeneous towers of interpreters are towers which exhibit following properties:

1. For any two levels $n, m \in \mathbb{R}_{\geq 0}, L_{I_n} \not\equiv L_{I_m}$
2. For any two levels $n, m \in \mathbb{R}_{\geq 0}, L_{I_n} \not\blacktriangleleft L_{I_m}$, where $\blacktriangleleft$ implies access to the left-hand side interpreter's state and $m \geq n$

3. For any language used in the tower $L_m \in \Sigma_L$, $\exists L_a \notin \Sigma_L . L_m \blacktriangleleft L_c \wedge L_c \blacktriangleleft L_c$

A common situation where one find such properties within a system of languages is the embedding of domain-specific languages (DLSs) and we describe the consequence of these properties in the subsequent sections.

1.3.1 Absence of: Meta-circularity

The first constraint imposed by definition 1.3 is that of necessarily mixed languages between levels of an interpretative tower. A practical challenge this poses for partial evaluators is the inability to reuse language facilities between levels of a tower. This also implies that one cannot define reflection and reification procedures as in 3-LISP [1], Blond [5], Black [8] or Pink [9].

1.3.2 Absence of: Reflectivity

The ability to introspect and change the state of an interpreter during execution is a tool reflective languages use for implementation of debuggers, tracers or even language features. With reflection, however, programs can begin to become difficult to reason about and the extent of control of potentially destructive operations on a running interpreter's semantics introduces overhead. Reflection in reflective towers implies the ability to modify an interpreter's interpreter. Hierarchies of language embeddings as the ones we are interested in rarely provide reflective capabilities at every part of the embedding.

1.3.3 Mixed Language Systems

An early mention of non-reflective and non-metacircular towers was provided in the first step of Danvy's systematic description of the reflective tower model [5]. However, potential consequences were not further investigated in their study. However, their denotational explanation of general interpretation and description of interpretater state served as a useful foundation for later work and our current study.

An extensive look at mixed languages in reflective towers was performed in chapter 5 of Sturdy's thesis [6] where he highlighted the importance of supporting a mixture of languages within a interpretation framework. Multi-layer systems such as YACC and C or Shell and Make are common practice. Sturdy goes on to introduce into his framework support for mixed languages that transform to a Lisp parse tree to fit the reflective tower model. Our work is similar in its commmon representation of languages, however, we remove the requirement of reflectivity and argue that this provides a convenient way of collapsing, through partial evaluation a mixed level tower of interpreters. While Sturdy's framework *Platypus* is a reflective interpretation of mixed languages, we construct a non-reflective tower consisting of mixed languages. The mix parial evluation framework [7], Jones et al. demonstrate

the PE of a simple interpreter into a language called Mixwell developed by the authors. This is similar in spirit to our framework except it is smaller in height. (section 5 of the paper [7]). also mentions removal of layers of metainterpretation in its conclusion

Recent work due to Sampson et al. [10] differentiates between value splicing and materialization. Materialization and cross-stage references are used to persist information across stages. This provides a possible solution to pass information about staging decisions across levels.

Partial Evaluation of Machine Code

Put together, the three properties imposed by definition 1.3 encourage a generalized solution irregardless of the language or structure of the tower at hand.

One of the earliest serious mentions of collapsing levels of interpretation using partial evaluation was [6]

‘

1.4 Coming Full Circle: Partial Evaluation and Reflective Towers

1.4.1 Compiling Reflective Languages

[8]: Language “Black”; has early uses of the act of collapsing modes of interpretation in a reflective setting. Its reflective model is closer to 3-LISP than to Blond or Brown [11]

The Truffle framework due to Whürthinger et al. [12] demonstrate a practical partial evluation framework for interpreters independent of language by providing a language and interpreter specifically designed to partially evaluate and thus collect as much information about a dynamic language at run time as possible.

2 Examples

Examples drawn from paper on collapsing towers [9]:

- Regular expression matcher ¡- Evaluator ¡- Virtual Machine
  - Generate low-level VM code for a matcher specialized to one regex (through arbitrary number of intermediate interpreters)
- Modified evaluator ¡- Evaluator ¡- Virtual Machine
  - Modified for tracing/counting calls/be in CPS
  - Under modified semantics ”interpreters become program transformers”. E.g. CPS interpreter becomes CPS transformer

3 Problems

To put our work and motivation into context consider following program architecture (originally described in [9]): some user script is executed by a Python interpreter running on a JavaScript emulator of a x86 CPU, all of which is run within a browser that eventually is executed on hardware. This construction resembles a practical realization of our defintion of heterogeneity in towers of interpreters, or in this case a tower of languages. Although such scenario might seem far-fetched, other forms of towers such as domain-specific languages embedded within a host are a form of tower of interpreters. Our study presents a systematic construction of a tower that resembles the structure of the tower we describe above: each level’s language is different from the one it interprets and strict interpretation, where implementation is infeasible, is replaced with compilation.

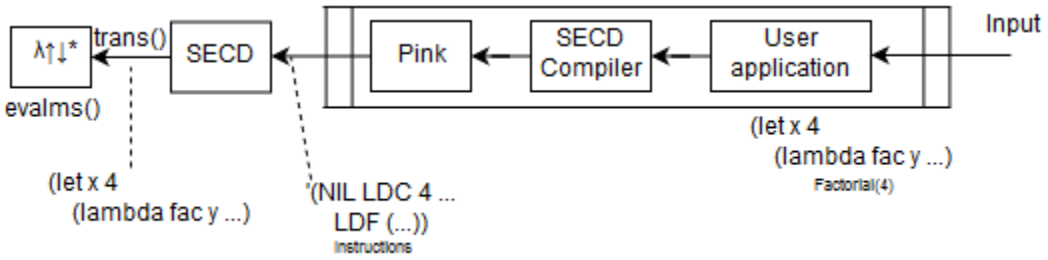

Figure 2: "Effectively functional" $\lambda^*_{\uparrow\downarrow}$with SECD tower above it

What we envision (with reference to this hypothetical setting) is handling the two following cases:

1. A one-off run of a python script on top of this stack should be collapsed by bypassing the emulator interpretation
2. A continuously running emulator evaluating a continuously running python interpreter should collapse individual runs of interpretation while respecting the dynamically changing environment
    - Here a dynamically changing environment also implies effects that are capable of changing the semantics of interpreters within the tower at runtime
    - In literature, the closest to compiling a dynamically changing tower is [13, 9] (for a *reflective* language Black) and GraalVM [14]

To tackle the first of these problems we construct a similar yet condensed form of the setting as shown in 2.

4 Contributions

In our study we investigate previous work on collapsing towers of interpreters and aim to take another step towards applying these techniques to real-world settings. We demonstrate that given a multi-level language and a lift operator we can stage individual interpreters in a sequence of non-metacircular interpreters and effectively generate code specialized for a given program eliminating interpretative overhead in the process. As part of the development of this framework our contributions include: (1) the development of extensions to the SECD machine that allow it to be staged (2) implementation of a compiler from a minimal LISP-style language to instructions of the staged SECD machine (3) demonstration of collapsing towers of interpreters built on top of the aforementioned SECD machine (4) and finally evaluation of the structure of the generated code and possible optimizations.

We deviate from traditional research in reflective towers in that we do not develop a separate language that demonstrates reflective tower capabilities and part from the constraints of metacircularity and reflection. Instead of generating levels in the tower dynamically through reflection and reification operators we construct a pre-determined tower resembling towers of interpreters in practice. We demonstrate initially how meta-circularity and reflection eases the collapsing process and then wire the tower in a way that breaks key implicit assumptions of said technique. Finally we propose a generalization of the original framework that deals with the constraints such a semantic gaps and lack of reflection and reification. We evaluate the framework on a set of abstract machines that are convenient to implement in Lisp-like fashion but are capable of modelling a broad set of functional and non-functional language properties.

5 Methodologies Background

- Stage polymorphism [15]: "abstract over staging decisions" i.e. single program generator can produce code that is specialized in many different ways (instance of the Fourth Futamura Projection? [16])
- Multi-level base evaluator written in $\lambda_{\uparrow\downarrow}$: supports staging operators (**polymorphic Lift**)
- Modify other interpreters: make them **stage polymorphic**, i.e. commands either evaluate code (like an interpreter) or generate code (like a translator)
- Stage only user-most interpreter: *wire tower* such that the **staging commands in L_n are interpreted directly in terms of staging commands in L_0** i.e. staging commands pass through all other layers handing down commands to layers below without performing any staging commands
- Non-reflective method: meta-circular evaluator **Pink** =¿ collapse arbitrary levels of "self-interpretation"
- By abstracting over staging decisions one can write the same program to both perform staging or evaluate directly [9] (maybe-lift)
- $\lambda_{\uparrow\downarrow}$features:
  - *run residual code*
  - binding-time/stage polymorphism [17]
  - preserves execution order of future-stage expressions
  - does not require type system or static analysis
    - TDPE [18] (great explanation also at [19]): **polymorphic Lift** operator turns static values into dynamic (future-stage) expressions

5.1 Towers of Interpreters Project Overview

5.1.1 Scala

- base.scala: implements definitional interpreter for $\lambda_{\uparrow\downarrow}$

6 Results

- Able to achieve compilation of stack-machine on top the Pink evaluator (including tracing evaluator etc.)
- Compilation i.e. collapsing through explicit staging annotations requires intricate knowledge of infrastructure and does not support all data structures e.g. stacks

6.1 Benchmarks

We extended the benchmarks provided as part of the original framework [9] with timings for staging the stack machine with respect to a user factorial program and timings for evaluating said program. The compilation output yielded and is essentially a loop unrolling of the (non-recursive SECD) factorial program without traces of the SECD emulator left:

```
Let(
  Gt(Lit(3),Lit(1)),
  Let(
    Equ(Var(3),Lit(0)),
    Let(
      If(Var(4),
        Lit(1),
        Let(
          Lift(Sym(.)),
          Let(
            Cons(Lit(3),Sym(.)),
            Let(
              Fst(Var(6)),
              Let(
                Minus(Lit(1),Var(7)),
                Let(
                  Times(Var(8),Lit(-1)),
                  Let(
                    Fst(Var(6)),
                    Let(
                      Times(Var(9),Var(10)),
                      Let(
                        Snd(Var(6)),
                        Let(
                          Minus(Lit(1),Var(7)),
                          Let(
                            Times(Var(13),Lit(-1)),
                            Let(
                              Minus(Lit(1),Var(14)),
                              Let(
                                Times(Var(15),Lit(-1)),
                                Let(
                                  Times(Var(16),Var(11)),
                                  Let(
                                    Minus(Lit(1),Var(14)),
                                    Let(
                                      Times(Var(18),Lit(-1)),
                                      Var(17))))))))))))))),
        Var(5))))
```

Generalization: because we sacrifice the fact input is static and mark them as dynamic (code) values PE technique is more like a translation then evaluation. The result of evluation is a new IR in terms of the base language

Varying degrees of generalization:

1. Interpreter: VM, Static input: Instructions, Dynamic inpute: Generalized to be the numbers or specially tagged value =¿ here we benchmark interpretative overhead of SECD machine for various generalization points
   - Treat all input as static =¿ equivalent to full evaluation
   - Treat all input as dynamic =¿ Generate a recursive loop in base-language terms but doesn't require case checking against non-existent instructions
   - Treat small part of input as dynamic
   - Treat part of input as dynamic =¿ Evaluates most of program
2. Interpreter: Pink, Static input: VM, further input: instructions =¿ need to decide where to stage
3. Interpreter: VM, Static input: Evaluator, further input: User program =¿ need to decide where to stage

Technical difficulties: implementation of letrec/multi-arg lambdas, implementation of mutable cells, decision on how to stage (i.e. where to annotate) but is essential to performance, leaking implementations between layers, base language getting bloated with features, non-termination when performing recursive calls from within SECD machine (no differentiation between a function being recursive or not)

Following is the generated code without the SECD machine

```
(lambda f0 x1
  (let x2 (eq? x1 0)
  (if x2 1

  (let x3 (- x1 1)
  (let x4 (f0 x3) (* x1 x4))))))
```

Following is the generated code with the SECD machine:

```
(let x0
  (lambda f0 x1
    (let x2
    (lambda f2 x3
      (let x4 (car x3)
      (let x5 (car x4)
      (let x6 (eq? x5 0)
      (let x7 (eq? x6 0)
      (if x7
      (let x8 (car x3)
      (let x9 (cdr x8)
      (let x10 (car x9)
      (let x11 (car x3)
      (let x12 (car x11)
      (let x13 (* x12 x10)
      (let x14 (cdr x3)
      (let x15 (cdr x14)
      (let x16 (car x15)
      (let x17 (cdr x16)
      (let x18 (car x17)
      (let x19 (car x3)
      (let x20 (car x19)
      (let x21 (- x20 x18)
      (let x22 (cons x13 '.)
      (let x23 (cons x21 x22)
      (let x24 (cons 1 '.)
      (let x25 (cons 10 x24)
      (let x26 (cons x25 x1)
      (let x27 (cons '. x26)
      (let x28 (cons x23 x27) (f2 x28))))))))))))))))))))))))))

      (let x8 (car x3)
      (let x9 (cdr x8) (car x9)))))))))
    (let x3 (cons 1 '.)
    (let x4 (cons 10 x3)
    (let x5 (cons 1 '.)
    (let x6 (cons 10 x5)
    (let x7 (cons x6 x1)
    (let x8 (cons '. x7)
    (let x9 (cons x4 x8) (x2 x9))))))))))) (x0 '.))
```

With a simple "smart constructor" optimization we can reduce the generated code size to:

```
(let x0
```

```
(lambda f0 x1
  (let x2
    (lambda f2 x3
      (let x4 (car x3)
      (let x5 (car x4)
      (let x6 (eq? x5 0)
      (let x7 (eq? x6 0)
      (if x7
        (let x8 (car x3)
        (let x9 (cdr x8)
        (let x10 (car x9)
        (let x11 (car x8)
        (let x12 (* x11 x10)
        (let x13 (cdr x3)
        (let x14 (cdr x13)
        (let x15 (car x14)
        (let x16 (cdr x15)
        (let x17 (car x16)
        (let x18 (- x11 x17)
        (let x19 (cons x12 '.)
        (let x20 (cons x18 x19)
        (let x21 (cons 1 '.)
        (let x22 (cons 10 x21)
        (let x23 (cons x22 x1)
        (let x24 (cons '. x23)
        (let x25 (cons x20 x24) (f2 x25))))))))))))))))))))

        (let x8 (car x3)
        (let x9 (cdr x8) (car x9)))))))))
  (let x3 (cons 1 '.)
  (let x4 (cons 10 x3)
  (let x5 (cons x4 x1)
  (let x6 (cons '. x5)
(let x7 (cons x4 x6) (x2 x7)))))))) (x0 '.))
```

Ackermann (with optimizations):

```
(let x0
(lambda f0 x1
  (let x2
  (lambda f2 x3
    (let x4 (car x3)
    (let x5 (car x4)
    (let x6 (eq? x5 0)
    (let x7 (eq? x6 0)
    (if x7
    (let x8 (car x3)
    (let x9 (cdr x8)
    (let x10 (car x9)
    (let x11 (eq? x10 0)
    (let x12 (eq? x11 0)
    (if x12
      (let x13 (car x3)
      (let x14 (cdr x13)
      (let x15 (car x14)
      (let x16 (- x15 1)
      (let x17 (car x13)
      (let x18 (cons x16 '.)
      (let x19 (cons x17 x18)
```

```
        (let x20 (cons '. x1)
        (let x21 (cons x19 x20)
        (let x22 (f2 x21)
        (let x23 (- x17 1)
        (let x24 (cons x22 '.)
        (let x25 (cons x23 x24)
        (let x26 (cons x25 x20) (f2 x26)))))))))))))))

        (let x13 (car x3)
        (let x14 (car x13)
        (let x15 (- x14 1)
        (let x16 (cons 1 '.)
        (let x17 (cons x15 x16)
        (let x18 (cons '. x1)
        (let x19 (cons x17 x18) (f2 x19)))))))))))))))

      (let x8 (car x3)
      (let x9 (cdr x8)
      (let x10 (car x9) (+ x10 1)))))))))))
    (let x3 (car x1)
    (let x4 (car x3)
    (let x5 (cdr x3)
    (let x6 (car x5)
    (let x7 (cons x4 '.)
    (let x8 (cons x6 x7)
    (let x9 (cons '. x1)
    (let x10 (cons x8 x9) (x2 x10)))))))))))) (x0 '.))
```

New method of lifting recursive applications:

```
(let x0
(lambda f0 x1
  (let x2 (cons 1 '.)
  (let x3 (cons 10 x2)
  (let x4 (cons x3 x1)
  (let x5
  (lambda f5 x6
    (let x7 (car x6)
    (let x8 (car x7)
    (let x9 (eq? x8 0)
    (if x9
    (let x10 (car x6)
    (let x11 (cdr x10) (car x11)))

    (let x10 (car x6)
    (let x11 (cdr x10)
    (let x12 (car x11)
    (let x13 (car x10)
    (let x14 (* x13 x12)
    (let x15 (cdr x6)
    (let x16 (cdr x15)
    (let x17 (car x16)
    (let x18 (cdr x17)
    (let x19 (car x18)
    (let x20 (- x13 x19)
    (let x21 (cons x14 '.)
    (let x22 (cons x20 x21)
    (let x23 (cons '. x4)
    (let x24 (cons x22 x23) (f5 x24)))))))))))))))))))))))))
```

```
    (let x6 (car x4)
    (let x7 (cdr x6)
    (let x8 (car x7)
    (let x9 (car x6)
    (let x10 (cons x8 '.)
    (let x11 (cons x9 x10)
    (let x12 (cons '. x4)
    (let x13 (cons x11 x12) (x5 x13)))))))))))) (x0 '.))
```

A partial evaluator is referred to as *strong* if, when the result of partially evaluating to an interpreter with respect to a target program yields the target proram as output. This is called Jones' optimality directly implies the removal of a layer of interpretation. Our experiments show that depending on which level of a tower we apply our partial evaluator on we can have this strong property of partial evaluators hold. When staging the user-most interpreter we are able to generate ANF code that is equivalent to the subject program the user wrote. If we stage at the level of the SECD machine we generate code that resembles some the structure of the SECD machine and are not able to get completely rid of the stack-based operations of the SECD machine.

6.1.1 Similarities to Mix

In the Mix partial evaluator [7] interpretative overhead is removed in a similar fashion from a sample interpreter when partially evaluated to terms in the Mixwell language of the same paper. However, the method by which they achieve PE differs ...

6.1.2 String Matcher

To evaluate our extended tower to the one used in the original Pink implementation we wrote a string matcher KNUTH to be executed on our staged SECD machine. Running the SECD in evalms's compilation mode we see SECD instructions as literals being included in the generated code. What's worse is that these literals are never used. Their presence is an example of a trade-off we make between implementation complexity and intrusiveness versus removal of interpretative overhead. Although we removed most of the evaluation overhead of the SECD machine itself, these instructions are present in the generated code because our implementation converts anything in the *env* and *stack* registers into dynamic values in the whenever we use lambdas as firt-class citizens for instance when using them as return values. If we have a . If we wanted to reduce generated code further we could either reuse the *fns* register for non-recursive functions as well and prevent instructions loaded as part of the *LDF* instruction to be lifted.

6.2 Parallels To Towers In the Wild

secd -¿ emulator (can PE of machine code help? [20]) meta-eval -¿ python lisp -¿ JavaScript base -¿ VM? GraalVM (could it be adapted)?

7 Normalization By Evaluation (NBE)

Useful Slides NBE paper More slides Supercompilation by Evaluation Supercompilation and Normalization by Evaluation

8 Type-Directed Partial Evluation (TDPE)

Alternative techniques include syntax directed partial evaluation and off-line partial evaluation Futamura's second project showed the direct relationship between an interpreter and a compiler -¿ hard to realize Jone's et al. developed MIX which was the first self-applicable PE that operated on a language developed by the same authors called MIXWELL. MIX innovated by making binding-time decisions offline as opposed to during partial evaluation time using source annotations for the PE However as noted in [19], traditionally partial evaluation worked on untyped languages where a single universal datatype could represent all static values PE of typed languages was one of the challenges with PE outlined by Jones [21].

8.1 Staging

There are two types of partial evaluation methodologies [22]:

- Offline partial evaluation
- Online partial evaluation [23]

Namin et al. [9], propose two languages Pink and Purple. Pink uses a form of online partial evaluation but requires manual staging facilities. Purple relies on LMS for automatic binding time analysis and staging which limits it to offline partial evaluation and thus relies on further optimization heuristics to achieve the same level of program specialization in the generated code as Pink.

Our language extends Pink with side effects and a stack machine that makes use of pointer like semantics for Lisp-like cons pairs. Thus we build on top of the NBE-style lift operator for staging. However, calling into the base-level lift requires knowledge about its use to be passed from the layers above. We can employ several strategies of doing this:

- A basic approach exposes the base layer staging operation to the level above. This is how the original Pink implementation works.
- At every layer *deduce* whether we need to call the underlying interpreter staging operator
  - This requires every level to include an implementation of such staging operations
- A mixture of passing staging operations to the layer below or implementing ones own operators
- Find a method of passing staging decisions through each layer in a generic way without intrusive changes to the evaluators of the layers
- Decide about calling staging operations at a particular point in the tower and apply previous points

We are interested in the last two point. In heterogeneous and practical towers a programmer does not have the liberty to introduce intrusive changes along each layer. The original Pink implementation assumes we are allowed to make arbitrary changes to evaluators. It effectively adds tags to the emitted representation of a layer above and lets the layer below infer from these tags what tag it itself should pass to the next layer, eventually calling the base-level Lift term.

8.2 Normalization by Evaluation

Since the languages involved in our study are untyped (except in the base-evaluator that serves as our partial evaluator) we also describe the underlying technique of partial evaluation called normalization-by-evaluation (NBE).

9 Why do we want to collapse towers?

The main reason is performance. The key realization of partial evaluation that lead to its development is that interpreters do redundant work but we can make it so they don't. Program specialization is simple and attractive on paper but poses significant engineering challenges and has not seen widespread adoption (until recent increasingly successful work on interpreter virtualization [14]).

Binding time analysis is one of the obstacles of program specialization. The program specializer needs to decide, either automatically or with assistance from the programmer, which data to treat as static and which as dynamic. Simple divisions can lead to code explosion or inefficient code generation, or worse, to non-termination of the specializer. This problem is known as *division* and is one of the key differences between offline and online PE techniques [22].

An curious use-case for staging towers of interpreters started with the challenge of compiling the reflective language proposed by Asai et al. [13, 11]. The authors are able to compile a Lisp-like reflective language, built through the infinite tower of interpreters model [4, 3, 5], with respect to the initial semantics of the tower. Amin et al. [9] then extended this work to allow compilation of such towers under modified semantics i.e. dynamically changing behaviour of individual interpreters. An interesting consequence demonstrated in their paper is the ability to derive translators in the process of collapsing.

9.0.1 Example of Deriving Translators

A trivial but useful example is logging. Given the tower in figure **??** we want to keep the added *useful* behaviour of I_3 while removing the *unuseful* other work of interpreting an intermediate representation. The interpretation of the IR of the level above is a mere accidental consequence of design instead of a necessity. We claim this work to be *interpretative overhead* and defer its quantification by benchmarks to a later section.

Collapsing the tower achieves exactly what we wanted, base-language (here the compilation target language) expressions including logging specialized to the user-level program.

A restriction with this method is its reliance on meta-circularity and reflection and other unsafe techniques:

- we are able "inject" the logging evaluator into tower because of its meta-circularity
- we expose staging operators throughout the tower through simple string manipulation
- instrumentation relies on meta-circularity since we simply redefine how constructs are evaluated before injecting the evaluator
- modification of semantics of the tower are done via reflection (ELABORATE)

Nonetheless, it is an interesting consequence of compiling towers that we can collect side-effects in individual levels (in the original framework at no extra cost), and have an interpreter above exhibit them. Analogous to a sieve, we take a coarse-grained collection of functionality, loosen it and extract only the individual grains we want. Imagine the sieve being coated with a thin film. All grains passing through will now have the film applied to them. Here sieves are the levels below and grains are the levels above. (REFINE ANALOGY)

Our study examines this property by testing the limits of how we can get the side effects to stick to interpreters in a useful way. One could imagine optimizations, parallelization or instrumentation as possible use cases. Under certain side-effects, we may, however, reach limits in terms of security (TROJANS IN HYPERVISORS) or ability to reason about a system. We are interested in the extent of these limits. (concurrency as a side-effect: instead of launching missiles we launch threads)

Partial evaluation is a conceptually powerful technique and its potential applications go far beyond what the largest case studies have shown. The idea of specialization is not limited to improving performance of language

interpreters but removing generality imposed by generic interfaces. This extends to programming languages, systems and even hardware design.

10 Towers in the Wild

To provide a real-world analogy of the language towers we are constructing describe some existing arrangements of multi-interpreter systems below:

- Here is a list of languages that are built on top of JavaScript. This is a three-level interpreter system: User-application¡-¿DSL¡-¿JavaScript Interpreter
- Here is a list of languages that compile to Python.
- v86 is a x86 CPU emulator written in JavaScript. This closely resembles our stack machine that is evaluated in both Pink or the Base language's multi-stage evaluator
- 6502asm is a microcontroller emulator in JavaScript

11 On Side-effects and Dynamic Semantics of Programming Languages

11.1 Adding Levels

Added stack machine DIAGRAMS

11.2 Result of collapsing

11.3 Cost of Emulation (or interpretation)

[24] Turing Tax Turing Tax specific slides

12 Methodologies

12.1 Abstract Machines

12.2 Collapsing Towers

In order to achieve the collapsing effect in Pink [9], the authors make use of two keys points: (1) side-effects are deferred to $\lambda_{\uparrow\downarrow}$ (2) staging commands are available throughout the tower

To achieve the same effect under constraints imposed by heterogeneity, we address both requirements first by showing heterogeneous towers are a special case of a generalized model for reflective towers and then providing a framework that uses the concept of a meta-controller from previous work on towers of interpreters to encapsulate staging information and pass it between stages.

Kogge's extensive study of machines for symbolic computation [25] and Diehl et al.'s survey of abstract machines provide several attractive opportunities for future investigations. The widely used Warren Abstract Machine (WAM) in logic programming, PCKS machine [26], MultiLisp

We implement some of the extensions to SECD described in [25] that permit non-deterministic computation and backtracking. The machine itself still remains purely functional. Abstract machines that aim to model specific

real system behaviour require the capability of the underlying machinery to provide mechanisms to manipulate cell locations and permit some form of store. For this purpose we extended the original $\lambda_{\uparrow\downarrow}$with the ability to create cells and mutate these. We then implement MultiLisp and the demand-driven evaluation extensions to the SECD machine and present our findings below.

Figure 12.2.4 shows a representation of our tower in terms of "sideways-stacked" tombstone diagrams. Although here the tower grows upwards and to the left this does not necessarily be. The compilers, or *translators* labelled LABEL and LABEL have been implemented in scala for convenience. To realize a vertical tower structure our Lisp-to-$\lambda_{\uparrow\downarrow}$translator, which converts Lisp source to Scala ASTs, could be ommitted letting the base-evaluator evaluate s-expressions directly. Similarly, the μ-to-SECD translator could be implemented in SECD instructions itself, possibly through generating such instructions using LABEL in a bootstrapping fashion.

12.2.1 SECD

TODO: matcher, cesk on metaeval The SECD machine due to Landin CITE is a mature stack-based abstract machine initially developed in order to provide a machine model capable of interpreting LISP programs. - Quick description of semantics The SECD operational semantics are shown in TABLE. All operations on the original SECD machine are performed on four registers: stack (S), environment (E), control (C), dump (D). SHOW EXAMPLE - Why is it a good machine for our tower? * Mature * Semantic gap * Extensions exist Numerous extensions exist that show promising candidates for experimentations with collapsing through the SECD machine. We describe a subsection of them in this section. * Straightforward Semantics Compared to WAM * Interprets Lisp style programs and allowed to reuse components - Full description of semantics - Problem with model as is for our implementation MENTION TYING THE KNOT One of the requirements in the implementation of a complete SECD machine is support for a "set!" instruction in the implementing language. EXAMPLE OF HOW SECD USES SET This is also a reason for the subsequent CESK machine which does not impose such a restriciton. Our work extends Amin et al.'s work on towers of interpreters CITE which exclusively supports functional languages and whose partial evaluation strategy relies on the absence of a store. In the design of our tower we had the choice between adding support for an underlying "set!" instruction in $\lambda_{\uparrow\downarrow}$or extend the SECD machine such that recursive functions applications do not require mutation in the underlying language. We chose the latter because it was simpler to add such extension than rewriting the Pink framework and would show additional use cases for $\lambda_{\uparrow\downarrow}$.

Digression: At earlier stages of our study we went down the route of extending the base evaluator with a store and add support for mutating cells. We experimented with turning the evaluator into a traditional CESK-style interpter. When it came to staging the SECD machine, however, we realized the importance of the interplay between recursive lambdas, the functional (stationary (?)) environment and flexibility of single-argument lambdas.

Difficulties arise whenever we attempt to mix dynamic and static values since its easy to want to compare and operate on the stack. Thus we need another register that is purely reserved for static values. To keep the amount of noise in the generated code lower one also needs to be careful to only mix static values into dynamic registers when necessary (show example of SECD noisy instrs in the generated code e.g. in simple non-recursive curried lambda or matcher code pre-noise reduction vs. factorial generated code).

12.2.2 Changes to $\lambda_{\uparrow\downarrow}$

Since SECD is a stack-based virtual machine all operations and data storage is performed on one of the four stack registers. By the principle of *division* in partial evaluators CITE, we want to be able to separate static and dynamic values to prevent undesired behaviour in our partial evaluator (EXAMPLE OF NON-TERMINATION). A minor extension we add to the original framework is the ability to lift nested tuples as opposed to only two-element tuples. This addition to the small-step semantics is shown in FIGURE.

SHOW OPERATIONAL SEMANTICS

Turns out single argument recursive lambdas are important to keep implementation simple but also for lifting closures - Modifications - Results

An issue that became apparent during the conversion from a regular to a staged SECD machine is the machine's stepwise operational semantics proneness to non-termination when applying our NBE-style partial evaluation. This is particularly prominent whenever an if-statement decides the arguments to the next step of the machine.

Figure FIGURE presents the augmented operational semantics for the staged SECD machine. The key to partially evaluating the SECD VM itself through "evalms" is the introduction of a function register, labelled *fns*, that holds closures to-be-called recursively using the *RAP* instruction. This alleviates us from the need to tie the knot in the environment as is done in the traditional SECD model. The benefit is two-fold:

1. We no longer require a store and ability to mutate variables in the underlying language
2. Prevent additional complexity of overloading the *LD* instruction to load values but also functions

RTN:

v.s e (RET.c) (s' e' c'd'fns'.d) fns $\longrightarrow$

$$\begin{cases} (v.s')\ e'c'd'fns' & \text{if d-register is tagged with 'ret} \\ (\text{lift-all } v) & \text{otherwise} \end{cases}$$

LDR:

s e (LDR (i j).c) d fns $\longrightarrow$ ((λ.(locate i j fns)).s) e c d fns

LDF:

s e (LDF ops.c) d fns $\longrightarrow$ ((λe'.(machine '() e' ops 'ret fns)) ops.e).s e c d fns

RAP:

(entryClo recClo.s) e (RAP.c) d fns $\longrightarrow$ '() e entryOps (s e c fns.d) (rec entryEnv '()).fns

where

entryClo := (entryFn (entryOps entryEnv))

recClo := (recFn (recOps recEnv))

rec := λenv.(machine '() env recOps 'ret (rec recEnv.'()).fns)

AP:

(fn env.v).s e (AP.c) ('() env') fns $\longrightarrow$ res.s env c d fns

where

res := (fn (lift-all (v s.env')))

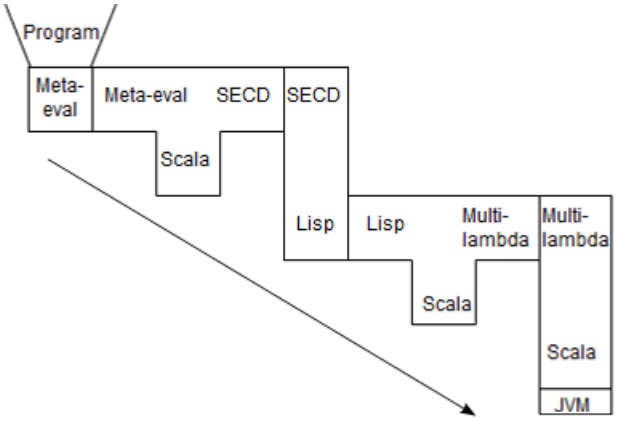

Figure 3: A tombstone digram representation of our framework

12.2.3 SECD Compiler

12.2.4 Tying the Knot

13 Problems in Partial Evaluation

[22]: page 26. Only recently has partial evaluation seen growing adoption with frameworks such as Scala's LMS [27] or Oracle's GraalVM [14]. The design space of partial evaluators is multi-faceted and considerations include:

- use of mixed languages (Futamura's mix for compilation requires same PE input language as mix is written in)
- accept different types of interpreters with different semantics (this is what we partly address) (and what GraalVM addresses?)
- performance
- BTA strategy: automatic or manual? offline or online? how to deal with non-termination?
- Heuristics for guiding PE to produce efficient/desirable code (e.g. you cannot say of a compiler that it doesn't produce more valuable program transformation. But PE's accept that some applications do not warrant it. How to detect such situations?)

Chapter 17: specifcies partial evaluation in the context of general program transformation and outlines PE research areas

TODO: staging vs pe partial evaluatability of SECD (and other linear interpreters/small step semantics vs denotational/big step/recursive descent parsers) retaining info in PE of TDPE style (our strategy is thunks) contribution: was natural to start with EBASE but uncovered problems BLANK tying the knot in the code instead of data transfer whiteboard diagram are able to stage/lift through compiler/translator structurally similar/equal to javascript tower implementations: 1. regular expression matcher (on metaeval or in place of metaeval) 2. try/catch 3. using cps adding amb (needs set! and store =¿ turn metaeval into cesk machine (possible because SECD supports recursion, lambdas, letrec, etc.)) Our collapsing strategy is based on the fact that we can choose to either evaluate

code or generate code which is an inherent property of the multi-level language base $\lambda_{\uparrow\downarrow}$. Thus in a realistic tower base needs to be a multi-level language

Partial evaluators can be thought of as lightweight optimizing compilers targeting specific optimization goals

Red thread: reflective towers-¿formalization of towers (first mention of PE by Sturdy and Blond) and first hint at heterogeneity-¿shift perspective to how such model is realizable-¿what is evaluation overhead of such models-¿compiling towers-¿polymorphic lift-¿define heterogeneity-¿find abstract machine for our purpose-¿SECD-¿requires side-effects-¿tie not in code instead of data-¿benchmarks+code comparisons+optimizations-¿extend tower-¿more benchmarks

Future work: abstract machine for partial evaluation? multi-level language abstract machine

Our study lives in the field of deriving compilers from interpreters and presents techniques of realizing such transformations using previous work on TDPE in the context of towers of interpreters. The crucial difference between modern compilers and interpreters, though the line got blurrier in the last years through JIT compilation in interpreted languages like Python and interpreter virtualization such as GraalVM, are the architecture dependent optimizations and complex pipeline of optimization passes present in compilers. Even in our small-scale study of partial evaluators we encountered questions of optimizing size and structure of generated code and proposed solutions specific to the interpreters we are partially evaluating. We echo the question posed in Jones [21] discussion of challenges in PE of whether we can generate compilers or in our case even simply generate code that rivals modern optimiing compilers. -¿ need more testing, all optimizations we introduced are very specific to the wiring of the tower, which is maybe how it is supposed to be, although compilers do not require optimizations based specifically on the program source they operate on

13.0.1 Meta-eval

In this section we extend the tower a level further by writing an interpreter that gets compiled into SECD instructions.

References

[1] B. C. Smith, "Reflection and semantics in lisp," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1984, pp. 23–35.

[2] D. P. Friedman and M. Wand, "Reification: Reflection without metaphysics," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, 1984, pp. 348–355.

[3] M. Wand and D. P. Friedman, "The mystery of the tower revealed: A nonreflective description of the reflective tower," *Lisp and Symbolic Computation*, vol. 1, no. 1, pp. 11–38, 1988.

[4] B. C. Smith, "Reflection and semantics in a procedural language (ph. d. thesis)," *Technical Report*, 1982.

[5] O. Danvy and K. Malmkjaer, "Intensions and extensions in a reflective tower," in *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM, 1988, pp. 327–341.

[6] J. C. Sturdy, "A lisp through the looking glass." 1993.

[7] N. D. Jones, P. Sestoft, and H. Søndergaard, "Mix: a self-applicable partial evaluator for experiments in compiler generation," *Lisp and Symbolic computation*, vol. 2, no. 1, pp. 9–50, 1989.

[8] K. Asai, S. Matsuoka, and A. Yonezawa, "Duplication and partial evaluation," *Lisp and Symbolic Computation*, vol. 9, no. 2-3, pp. 203–241, 1996.

[9] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 52, 2017.

[10] A. Sampson, K. S. McKinley, and T. Mytkowicz, "Static stages for heterogeneous programming," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 71, 2017.

[11] K. Asai, "Compiling a reflective language using metaocaml," *ACM SIGPLAN Notices*, vol. 50, no. 3, pp. 113–122, 2015.

[12] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 662–676.

[13] K. Asai, H. Masuhara, and A. Yonezawa, *Partial evaluation of call-by-value λ-calculus with side-effects*. ACM, 1997, vol. 32, no. 12.

[14] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013, pp. 187–204.

[15] G. Ofenbeck, T. Rompf, and M. Püschel, "Staging for generic programming in space and time," in *ACM SIGPLAN Notices*, vol. 52, no. 12. ACM, 2017, pp. 15–28.

[16] R. Glück, "Is there a fourth futamura projection?" in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 2009, pp. 51–60.

[17] F. Henglein and C. Mossin, "Polymorphic binding-time analysis," in *European Symposium on Programming*. Springer, 1994, pp. 287–301.

[18] O. Danvy, "Type-directed partial evaluation," in *Partial Evaluation*. Springer, 1999, pp. 367–411.

[19] B. Grobauer and Z. Yang, "The second futamura projection for type-directed partial evaluation," *Higher-Order and Symbolic Computation*, vol. 14, no. 2-3, pp. 173–219, 2001.

[20] V. Srinivasan and T. Reps, "Partial evaluation of machine code," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 860–879.

[21] N. D. Jones, "Challenging problems in partial evaluation and mixed computation," *New generation computing*, vol. 6, no. 2-3, pp. 291–302, 1988.

[22] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[23] W. R. Cook and R. Lämmel, "Tutorial on online partial evaluation," *arXiv preprint arXiv:1109.0781*, 2011.

[24] M. Steil, "Dynamic re-compilation of binary risc code for cisc architectures," *Technische Universitat Munchen*, 2004.

[25] P. M. Kogge, *The architecture of symbolic computers*. McGraw-Hill, Inc., 1990.

[26] L. Moreau, "The pcks-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations," in *European Symposium on Programming*. Springer, 1994, pp. 424–438.

[27] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, no. 2. ACM, 2010, pp. 127–136.