Collapsing Towers of Interpreters

NADA AMIN and TIARK ROMPF

Given a tower of interpreters, i.e. a sequence of interpreters interpreting each other, we aim to collapse this tower into a compiler that removes all interpretive overhead in a single pass. We present a multi-level lambda calculus that features *staging constructs* and *stage polymorphism*: based on runtime parameters, an interpreter can either act as a normal interpreter or generate code, which turns it into a compiler, according to the first Futamura projection. We identify stage polymorphism as the key mechanism to make such interpreters compose in a collapsible way.

We present a meta-circular Lisp interpreter on top of this calculus and demonstrate that we can collapse arbitrarily many levels of self-interpretation, including levels with semantic modifications. We discuss several examples: compiling regular expressions through a Lisp interpreter to base code, building program transformers from modified interpreters, and others. We develop these ideas further to include reflection and reification, culminating in an implementation of the reflective language Black, which implements a conceptually infinite tower, where every aspect of the semantics can change dynamically. As a novel feature, we demonstrate how user programs can be compiled and recompiled under user-modified semantics.

1 INTRODUCTION

This paper is concerned with the following challenge: given a sequence of programming languages $L_0, \ldots, L_n$ and interpreters for L_{i+1} written in L_i, derive a compiler from L_n to L_0. This compiler should be optimal in the sense that the translation removes all interpretive overhead of the intermediate languages, and it should require just a single pass. Without loss of generality, we restrict the scope to interpreters based on variations of the lambda calculus as L_0. To make matters more interesting, we also consider that a) some or all interpreters may be *reflective*, i.e. can be inspected and modified at runtime; and b) the sequence of interpreters may be *conceptually infinite*.

It is well known that staging an interpreter yields a compiler (review in Section 2). So as a first attempt, we might try to stage all intermediate interpreters individually. However, this approach falls short of solving the general challenge: first, it requires each intermediate language to have dedicated code generation facilities targeting the next language. Second, it would produce a chain of translators instead of a one-pass compiler. This means that it cannot work in the case of a reflective tower, where delineations between languages are fuzzy and execution might jump back and forth between different levels.

Is there another way? The key insight of our approach is to start with a multi-level language L_0, i.e. a language that has built-in staging operators, and to express all other interpreters in a way that makes them *stage polymorphic*, i.e. able to act either as an interpreter or as a translator. Then, we wire up the interpretive tower so that the staging commands for L_n are directly interpreted in terms of the staging commands of L_0. All intermediate interpreters $L_1, \ldots, L_{n-1}$ act in a kind of pass-through mode, handing down staging commands from L_n, but not executing any staging commands of their own. Our approach only requires staging the final interpreter, instead of all interpreters in the tower. As we will see, our approach can sustain collapsing arbitrary levels of interpretation below the staged interpreter and compiling under user-modified semantics.

As an example of collapsing a tower of interpreters, consider a base virtual machine executing an evaluator executing a regular expression matcher. We can think of this setup as a tower of three interpreters (virtual machine, evaluator, regular expression matcher). By collapsing this tower, we can generate low-level (virtual machine) code for a matcher specialized to one regular expression. In our approach, we can add an arbitrary number of intermediate evaluators, while still enabling end-to-end collapse.

As an example of compiling under user-modified semantics, consider a base virtual machine executing an evaluator executing a modified evaluator executing a user program. The modified evaluator can, for example, (1) add tracing or counting of variable accesses, or (2) it can be written in continuation-passing style (CPS). Now, collapsing the tower will translate the user program to low-level (virtual machine) code, and this code will (1) have extra calls for tracing or counting, or (2) be in CPS. Thus, under modified semantics, interpreters become program transformers. In particular, a CPS-interpreter becomes a CPS-converter. Throughout this paper, we will see several examples of collapsing towers of interpreters, in particular in a reflective setup, where each level in the tower is open to inspection and change.

The high-level contribution of this paper is to show that explicit staging with the ability to abstract over staging decisions (i.e., *stage polymorphism*) is a versatile device to collapse towers of interpreters, even in very dynamic scenarios, where users can modify semantics on the fly. The specific contributions of this paper are the following:

- We develop a two-level kernel language $\lambda_{\uparrow\downarrow}$ that supports staging through a polymorphic `Lift` operator and stage polymorphism through dynamic operator overloading (Section 3). We discuss a first use case of interpreter specialization and stage polymorphism.
- We present a meta-circular interpreter for Pink, a restricted Lisp front-end, and demonstrate that we can collapse arbitrarily many levels of self-interpretation via compilation: this achieves our challenge of collapsing (finite) towers of interpreters (Section 4). We discuss optimality and correctness of the approach.
- We extend Pink with mechanisms for reflection and compilation from within, enabling user programs to execute expressions as part of an interpreter at any level in a tower, and compiling functions under modified semantics (Section 5).
- We develop these ideas further into the language Purple, a variant of Asai's reflective language Black [3], where every aspect of the semantics can change dynamically based on a conceptually infinite tower. In contrast to Black, Purple programs can be recompiled on the fly to adapt to modified semantics—a challenge left open by Asai [2] (Section 6).
- We present a range of examples in Purple / Black that make extensive use of reflection (Section 7). We implement Purple on top of LMS (Section 9), and discuss how stage polymorphism can be achieved using type classes in this typed setting (Section 8).
- We show benchmarks that confirm compilation and collapsing (Section 10).

We discuss related work in Section 11 and offer concluding remarks in Section 12.

2 PRELIMINARIES

It is well known that interpreters and compilers are fundamentally linked through *specialization*, as formalized in the three Futamura projections [24]. First, specializing an interpreter to a given program yields a compiled version of that program. Second, a process that can specialize a given interpreter to any program is equivalent to a compiler. Third, a process that can take any interpreter and turn it into a compiler is a *compiler generator*, also called *cogen*.

For a given interpreter, the corresponding compiler is also called its *generating extension* [18]. Since compilers are often preferable to interpreters, and preferable to running a potentially costly specialization process on an interpreter for every input program, how does one compute the generating extension of a given program?

The third Futamura projection tells us that double self-application of a generic program specializer is one way to produce a compiler generator *cogen*, which can compute a generating extension for any program that resembles an interpreter, i.e. takes a static and a dynamic piece of input.

In the simplest possible setting, partial evaluation can be viewed as a form of normalization, which propagates constants and performs reductions whenever it encounters a redex, i.e. a combination of introduction and elimination form. But most interesting languages are not strongly normalizing, i.e., uncurbed eager reduction might diverge, and even for terminating languages or programs it can lead to exponential blow-up due to duplication of control-flow paths. This means that some static redexes need to be residualized—but how to pick which ones to reduce, and which ones to residualize?

In general this is a very hard problem. In a traditional offline partial evaluation setting, it is the job of a binding-time analysis (BTA). The result of binding-time analysis is an annotated program in a *multi-level* language, which defines which expressions to reduce statically and which to residualize.

A key realization is that if one starts with a binding-time annotated interpreter, expressed in a multi-level language, then deriving a *cogen* by hand is actually quite straightforward [6, 57]. What is more, when starting from a multi-level program, it is actually easy to derive the generating extension itself! Thus, multi-level languages are attractive in their own right as tools for *programmable* specialization, as evidenced for example by MetaML [55] and MetaOCaml [9].

Proposed multi-level languages differ in many details, but usually provide a syntax like this:

$$n \mid x \mid e \mathbin{@^b} e \mid \lambda^b x.e \mid \ldots$$

Function application uses an explicit infix operator `@`, and the binding-time annotations b define at which *stage* an abstraction or application is computed. Well-formedness of binding-time annotations is usually specified as a type system. In the simplest case, b ranges over S, D for static or dynamic, but in more elaborate systems (e.g. in [27, 57]) b can range over integers or include variables β for polymorphism [32].

Multi-stage languages in the line of MetaML [55] feature quasiquotation syntax instead:

$$n \mid x \mid e\ e \mid \lambda x.e \mid \langle e \rangle \mid {\sim} e \mid \texttt{run}\ e \mid \ldots$$

Brackets $\langle e \rangle$ correspond to quotes, and escapes $\sim e$ correspond to unquotes; `run` e executes a piece of quoted code.

Other systems are implemented as libraries in a general-purpose host language, e.g. LMS [47] in Scala. Multi-level languages differ also quite significantly in their semantics. MetaML and its descendants, for example, provide hygiene guarantees for bindings, but interpret quotation in a purely syntactic way. This can lead to reordering or duplication of quoted expressions, which is often undesirable, in particular when combined with side effects.

3 MULTI-LEVEL CORE LANGUAGE $\lambda_{\uparrow\downarrow}$

With an eye towards the challenge posed in the introduction, we present a new multi-level kernel language $\lambda_{\uparrow\downarrow}$ which combines a number of desirable features. Like MetaML, it contains facilities to run residual code. Like polymorphic BTA [32], it supports stage- or binding-time polymorphism. Like LMS, its evaluation preserves the execution order of future-stage expressions. But unlike most other systems, $\lambda_{\uparrow\downarrow}$ does not require a type system or any other static analysis. Its key mechanism is a polymorphic `Lift` operator that turns a static, present-stage, *value* into a future-stage expression.

We present a small-step operational semantics in Figure 1 and a big-step operational semantics as a definitional interpreter written in Scala in Figure 2. The small-step version lends itself to formal reasoning, while the big-step version is more suitable for experimentation. Note that the definitional interpreter does not use any advanced Scala features, and can easily be translated to other call-by-value languages with mutable references. As a case in point, we also implemented an equivalent semantics in Scheme. The small-step semantics is implemented in PLT Redex [19].

Syntax

$$
\begin{array}{lll}
e & ::= & x \mid \texttt{Lit}(n) \mid \texttt{Str}(s) \mid \texttt{Lam}(f,x,e) \mid \texttt{App}(e,e) \mid \texttt{Cons}(e,e) \mid \texttt{Let}(x,e,e) \mid \texttt{If}(e,e,e) \mid \\
 & & \oplus^1(e) \mid \oplus^2(e,e) \mid \texttt{Lift}(e) \mid \texttt{Run}(e,e) \mid g \\
g & ::= & \texttt{Code}(e) \mid \texttt{Reflect}(e) \mid \texttt{Lam}_c(f,x,e) \mid \texttt{Let}_c(x,e,e) \\
\oplus^1 & ::= & \texttt{Car} \mid \texttt{Cdr} \mid \texttt{isLit} \mid \texttt{isStr} \mid \texttt{isCons} \\
\oplus^2 & ::= & \texttt{Plus} \mid \texttt{Minus} \mid \texttt{Times} \mid \texttt{Eq} \\
v & ::= & \texttt{Lit}(n) \mid \texttt{Str}(s) \mid \texttt{Lam}(f,x,e) \mid \texttt{Cons}(v,v) \mid \texttt{Code}(e) \\
B(X) & ::= & \texttt{Cons}(X,e) \mid \texttt{Cons}(v,X) \mid \texttt{Let}(x,X,e) \mid \texttt{App}(X,e) \mid \texttt{App}(v,X) \mid \texttt{If}(X,e,e) \mid \\
 & & \oplus^1(X) \mid \oplus^2(X,e) \mid \oplus^2(v,X) \mid \texttt{Lift}(X) \mid \texttt{Run}(X,e) \mid \texttt{Reflect}(X) \\
R(X) & ::= & \texttt{Lift}(\texttt{Lam}_c(f,x,X)) \mid \texttt{If}(\texttt{Code}(e),X,e) \mid \texttt{If}(\texttt{Code}(e),v,X) \mid \texttt{Run}(\texttt{Code}(e),X) \mid \texttt{Let}_c(x,e,X) \\
E & ::= & [\,] \mid B(E) \\
M & ::= & [\,] \mid B(M) \mid R(M) \\
P & ::= & [\,] \mid B(Q) \mid R(P) \\
Q & ::= & B(Q) \mid R(P)
\end{array}
$$

Reduction rules $\boxed{e \longrightarrow e}$

$$
\begin{array}{lcl}
M[\texttt{Let}(x,v,e)] & \longrightarrow & M[[v/x]e] \\
M[\texttt{App}(\texttt{Lam}(f,x,e),v)] & \longrightarrow & M[[v/x][\texttt{Lam}(f,x,e)/f]e] \\
M[\texttt{If}(n \neq 0,e_1,e_2)] & \longrightarrow & M[e_1] \\
M[\texttt{If}(0,e_1,e_2)] & \longrightarrow & M[e_2] \\
M[\texttt{Car}(\texttt{Cons}(v_1,v_2))] & \longrightarrow & M[v_1] \\
M[\texttt{Cdr}(\texttt{Cons}(v_1,v_2))] & \longrightarrow & M[v_2] \\
M[\texttt{isLit}(\texttt{Lit}(n))] & \longrightarrow & M[\texttt{Lit}(1)] \\
M[\texttt{isLit}(v \neq \texttt{Code}(\_),\neq \texttt{Lit}(\_))] & \longrightarrow & M[\texttt{Lit}(0)] \\
M[\texttt{isStr}(\texttt{Str}(s))] & \longrightarrow & M[\texttt{Lit}(1)] \\
M[\texttt{isStr}(v \neq \texttt{Code}(\_),\neq \texttt{Str}(\_))] & \longrightarrow & M[\texttt{Lit}(0)] \\
M[\texttt{isCons}(\texttt{Cons}(v_1,v_2))] & \longrightarrow & M[\texttt{Lit}(1)] \\
M[\texttt{isCons}(v \neq \texttt{Code}(\_),\neq \texttt{Cons}(\_,\_))] & \longrightarrow & M[\texttt{Lit}(0)] \\
M[\texttt{Plus}(\texttt{Lit}(n_1),\texttt{Lit}(n_2))] & \longrightarrow & M[\texttt{Lit}(n_1+n_2)] \\
M[\texttt{Minus}(\texttt{Lit}(n_1),\texttt{Lit}(n_2))] & \longrightarrow & M[\texttt{Lit}(n_1-n_2)] \\
M[\texttt{Times}(\texttt{Lit}(n_1),\texttt{Lit}(n_2))] & \longrightarrow & M[\texttt{Lit}(n_1 \cdot n_2)] \\
M[\texttt{Eq}(v_1 \neq \texttt{Code}(\_),v_2 \neq \texttt{Code}(\_))] & \longrightarrow & M[\texttt{Lit}(\text{if } v_1 = v_2 \text{ then 1 else 0})] \\
M[\texttt{App}(\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{App}(e_1,e_2))] \\
M[\texttt{If}(\texttt{Code}(e_0),\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{If}(e_1,e_2,e_3))] \\
M[\,\oplus^1(\texttt{Code}(e))] & \longrightarrow & M[\texttt{Reflect}(\oplus^1(e))] \\
M[\,\oplus^2(\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\oplus^2(e_1,e_2))] \\
M[\texttt{Lift}(\texttt{Lit}(n))] & \longrightarrow & M[\texttt{Code}(\texttt{Lit(n)})] \\
M[\texttt{Lift}(\texttt{Cons}(\texttt{Code}(e_1),\texttt{Code}(e_2)))] & \longrightarrow & M[\texttt{Reflect}(\texttt{Cons}(e_1,e_2))] \\
M[\texttt{Lift}(\texttt{Lam}(f,x,e))] & \longrightarrow & M[\texttt{Lift}(\texttt{Lam}_c([\texttt{Code}(x)/x][\texttt{Code}(f)/f]e))] \\
M[\texttt{Lift}(\texttt{Lam}_c(f,x,\texttt{Code}(e)))] & \longrightarrow & M[\texttt{Reflect}(\texttt{Code}(\texttt{Lam}(f,x,e)))] \\
M[\texttt{Lift}(\texttt{Code}(e))] & \longrightarrow & M[\texttt{Reflect}(\texttt{Lift}(\texttt{Code}(e_1)))] \\
M[\texttt{Run}(\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{Run}(e_1,e_2))] \\
M[\texttt{Run}(v_1 \neq \texttt{Code}(\_),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{App}(\texttt{Lam}(\_,\_,e_2),v_1)] \\
P[E[\texttt{Reflect}(\texttt{Code}(e))]] & \longrightarrow & P[\texttt{Let}_c(x,e,E[\texttt{Code}(x)])] \text{ where } x \text{ is fresh} \\
M[\texttt{Let}_c(x_1,e_1,\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Code}(\texttt{Let}(x_1,e_1,e_2))]
\end{array}
$$

Fig. 1. $\lambda_{\uparrow\downarrow}$ Small-Step Semantics

```
// Syntax -- Exp corresponds to e from small-step (Fig. 1), without internal-only forms (Reflect, Code, ...).
//        -- Instead of explicit names, we use de Bruijn levels (Var(n)), hence Lam(e) instead of Lam(x,e).
//        -- We elide Strings and many straightforward operators for simplicity.
Exp ::= Lit(n:Int) | Var(n:Int) | Lam(e:Exp) | App(e1:Exp, e2:Exp) | Cons(a:Exp,b:Exp)
      | Let(e1:Exp,e2:Exp) | If(c:Exp,a:Exp,b:Exp) | Plus(a:Exp,b:Exp) | ... -, *, =, ...
      | Lift(e:Exp) | Run(b:Exp,e:Exp)
Val ::= Cst(n:Int) | Tup(a:Val,b:Val) | Clo(env:Env,e:Exp) | Code(e:Exp)
Env = List[Val]

// NBE-style polymorphic lift operator
def lift(v: Val): Exp = v match {
  case Cst(n)       => Lit(n)
  case Tup(a,b)     => val (Code(u),Code(v))=(a,b); reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)      => reflect(Lift(e)) }
def liftc(v: Val) = Code(lift(v))

// Multi-stage evaluation
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)       => Cst(n)
  case Var(n)       => env(n)
  case Cons(e1,e2)  => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)       => Clo(env,e)
  case Let(e1,e2)   => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case Lift(e)      => liftc(evalms(env,e))
  case App(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)    => evalms(env,c) match {
    case Cst(n)       => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)     => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case Plus(e1,e2)  => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  ...
  case Run(b,e)     => evalms(env,b) match {
    case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _            => evalmsg(env, reifyc(evalms(env, e)))) } }
def evalmsg(env: Env, e: Exp) = reifyv(evalms(env,e))

// Additional helpers
var stFresh: Int       = 0
var stBlock: List[Exp] = Nil
def fresh()            = { stFresh += 1; Var(stFresh-1) }
def run[A](f: => A): A = { val sF = stFresh; val sB = stBlock; try f finally { stFresh = sF; stBlock = sB } }
def reify(f: => Exp)   = run { stBlock = Nil; val last = f; (stBlock foldRight last)(Let) }
def reflect(s:Exp)     = { stBlock :+= s; fresh() }
def reifyc(f: => Val)  = reify { val Code(e) = f; e }
def reflectc(s: Exp)   = Code(reflect(s))
def reifyv(f: => Val)  = run { stBlock = Nil; val res = f
  if (stBlock == Nil) res else { val Code(last) = res; Code((stBlock foldRight last)(Let)) } }
```

Fig. 2. $\lambda_{\uparrow\downarrow}$ Big-Step Semantics as a Definitional Interpreter in Scala

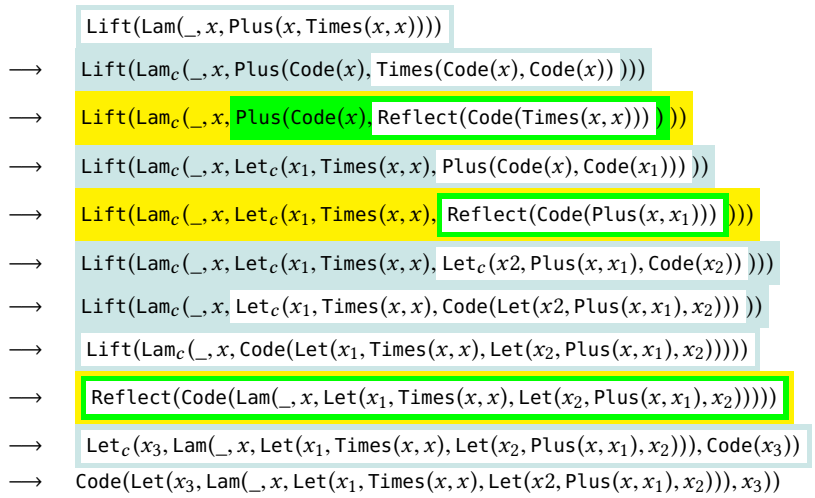

Fig. 3. Example of small-step derivation in $\lambda_{\uparrow\downarrow}$ highlighting the P, E and M contexts

The term syntax contains λ-calculus constructs, plus operators `Lift` and `Run`. In small-step, there are additional intermediary constructs for book keeping, such as `Reflect`, $\texttt{Lam}_c$ and $\texttt{Let}_c$ (noted under the syntax term g in Figure 1). The value syntax contains standard constants, tuples, closures (plain lambdas in the small-step semantics), and in addition `Code` objects that hold expressions. All functions are potentially recursive, taking a self-references as the first additional argument. This means that the term $\texttt{Lam}(f, x, e)$ is equivalent to the term $\texttt{fix}(\lambda f.\lambda x.e)$ in the usual λ-calculus with an explicit fixpoint combinator `fix`.

The polymorphic `Lift` operator is inspired by a corresponding facility in normalization by evaluation (NBE) [5, 13]. Its purpose is to convert values into future-stage expressions. Lifting a number is immediate, lifting a tuple is performed element-wise and lifting a code value creates a `Lift` expression. To lift a (potentially recursive) function, the function creates a λ-abstraction via two-level η-expansion, as in NBE. In the small-step semantics, lifting a function steps to the intermediary $\texttt{Lam}_c$ construct, which marks the body for reification. Helper terms like `Reflect`, $\texttt{Lam}_c$, $\texttt{Let}_c$ serve to create future-stage code in administrative normal form (ANF) [21] to maintain the relative evaluation order of expressions (see example in Figure 3). This is a standard practice in partial evaluators that deal with state and effects, and otherwise known as "let-insertion" [7, 31, 39, 59].

In the big-step semantics, ANF conversion is achieved with a set of helper functions. Each individual expression is *reflected*, storing it in the `stBlock` data structure, and all reflected expressions in a scope can be captured into a sequence of `Let` bindings via `reify`.[1] In the small-step semantics, the same behavior is modeled by the last two rules of the operational semantics. The first rule carefully splits an expression into a reification context and a reflection context $P[E[.]]$ and pulls out the reflected sub-expression into a $\texttt{Let}_c$, which is eventually transformed in a `Code` of `Let` by the second rule.

To illustrate how a simple function term is lifted and how the context decomposition guides insertion of `Let` bindings in the right places, we show a small-step derivation for the term e =

[1]It is important to note that the `reflect` and `reify` functions are only a semantic device to generate code in ANF. They provide a direct-style API to a conceptual let-insertion monad via *monadic reflection* [20], but they have nothing to do with reflective language capabilities in the sense of Section 5 and 6.

`Lam`(_, x, `Plus`(x, `Times`(x, x))) in Figure 3. The big-step evaluator computes the equivalent result in a single call to `evalmsg(Nil,e)`. In the big-step implementation, we use a name-less De Bruijn level representation for simplicity. The top-level entrypoint to multi-level evaluation is `evalmsg` which delegates to `evalms` and also packages up and returns all generated code, if any.

The key design behind $\lambda_{\uparrow\downarrow}$ is that introduction forms (e.g. `Lit`, `Lam`, `Cons`) always create present-stage values, which can be lifted explicitly using `Lift`, and that elimination forms (e.g. `App`, `If`, `Plus`) are *overloaded*, i.e. they match on their arguments and decide on present-stage execution or future-stage code generation based on whether their arguments are code values or not. Mixed code and non-code values lead to errors, but a variant with automatic conversion of primitive constants would be conceivable as well.

A curious case is `Run`, the elimination form for code values. Unlike other elimination forms, `Run` *always* receives `Code` values as arguments, hence matching on the expression to be evaluated would not allow us to decide whether to evaluate the argument now or generate a `Run` expression for the future stage. Hence, `Run` takes an *additional* initial argument b, which is not evaluated, but solely exists for the purpose of matching. Thus, `Run`(`Cst`(0), e) will evaluate e, whereas `Run`(`Lift`(`Cst`(0)), e) will generate a call to `Run`(`Cst`(0), e) in the next stage.

3.1 A Lisp-Like Front-End

We implement a small Lisp reader that translates S-expression to $\lambda_{\uparrow\downarrow}$ syntax. The mapping is straightforward, with proper names vs De Bruijn levels being the biggest difference between the front-end and the core definitional interpreter. We also introduce syntactic sugar for multi-argument functions, and we extend the core language slightly to add support for proper booleans, equality tests, quote and a few other constructs. We make this reader available via a function `trans`, and it will play a key role when we implement reflection in Section 5.1.1.

As a first programming example, here is a tiny generic list matcher that tests if the list `s` has the list `r` as a prefix.

```
(define matches (lambda (r) (lambda (s)
  (if (null? r) #t (if (null? s) #f
    (if (eq? (car r) (car s)) ((matches (cdr r)) (cdr s)) #f))))))
> ((matches '(a b)) '(a c))   ;; => #f
> ((matches '(a b)) '(a b))   ;; => #t
> ((matches '(a b)) '(a b c)) ;; => #t
```

To play with multi-level evaluation and Futamura projections, let us turn this string matcher, which can be viewed as a (simple) interpreter over the pattern string `r`, into a compiler that generates specific matching code for a given pattern. We treat the pattern `r` as static and the input `s` as dynamic:

```
(define matches (lambda (r) (lambda (s)
  (if (null? r) (lift #t) (if (null? s) (lift #f)
    (if (eq? (lift (car r)) (car s)) ((matches (cdr r)) (cdr s)) (lift #f)))))))
```

To make this work, the inner function has to return code values as well. Hence we lift all result values `#t` and `#f`. We also need to lift the result of `(car r)`, the current (static) pattern character to be compared with the (dynamic) input character.

With these modifications, we can generate code for matching a particular prefix, by partially applying `matches`, lifting the result, and running it:

```
(define start_ab (run 0 (lift (matches '(a b)))))
```

Recall that the `0` argument to `run` designates the desired "run now". The resulting generated code has only the low-level operations on the dynamic input `s`. The static input `r` causes three unfolding

of the matches body with the first static if disappearing from the generated code. The generated code is identical to the internal ANF representation of the following user-level program:

```
(define start_ab (lambda (s)
  (if (null? s) #f (if (eq? 'a (car s)) (let (s1 (cdr s))
  (if (null? s1) #f (if (eq? 'b (car s2)) #t #f)))))))
```

While this simple example conveys the key ideas, it deliberately leaves many questions unanswered. For example, how do we deal with loops and recursion, e.g. if we want to support patterns with wildcards or repetition patterns? We will present a more powerful matcher that supports such features in Figure 5.

3.2 Stage Polymorphism

Going back to the examples of Section 3.1, we started with a plain, unmodified, interpreter program and added lift annotations in judiciously chosen places to turn it into a code generator. But now the original program is lost! Sure, being diligent software engineers, we can still retrieve the previous version from our version control system of choice, but if we want to keep both for the future, then we will have to maintain two slightly different versions of the same piece of code.

And there are good reasons for running generic code sometimes, without specialization: imagine, for example, that we want to use our string matcher with very long patterns. Then generating a big chunk of code for each such pattern will likely be wasteful. In fact, we might want to introduce a dynamic cut off: specialize only if the pattern length is less then a certain threshold. Assume that matches-gen is the generic matches function, and matches-spec is the one including lifts from Section 3.1, we would like to write:

```
(define matches-maybe-spec (lambda (r) (if (< (length r) 20) (run 0 (lift (matches-spec r))) (matches-gen r))))
```

The notion of *stage-polymorphism* or *binding-time- polymorphism* enables us to actually achieve this. The key insight is that we can *abstract over* lift via η-expansion. We rewrite matches-spec back into a generic matches as follows, replacing lift with calls to a parameter maybe-lift:

```
(define matches (lambda (maybe-lift) (lambda (r) (lambda (s)
  (if (null? r) (maybe-lift #t) (if (null? s) (maybe-lift #f)
    (if (eq? (lift (car r)) (car s)) ((matches (cdr r)) (cdr s)) (maybe-lift #f))))))))
```

Now, we can define matches-spec and matches-gen simply as

```
(define matches-spec (matches (lambda (e) (lift e))))
(define matches-gen (matches (lambda (e) e)))
```

which completes our stage-polymorphic string matcher.

4 BUILDING AND COLLAPSING TOWERS

We have seen in the previous sections how we can turn simple interpreters into compilers, and how we can abstract over staging decisions. We now turn our attention to the challenge posed in the introduction: collapsing *towers* of interpreters, i.e. sequences of multiple interpreters interpreting one another as input programs. Stage polymorphism is the key mechanism to make interpreters compose in such a collapsible way.

In this section we will focus on finite towers and defer conceptually infinite towers to Section 6. We start by defining a meta-circular interpreter, shown in Figure 4, for a slightly more restricted Lisp front-end that is closer to $\lambda_{\uparrow\downarrow}$. We dub this language *Pink*. In comparison to Scheme, only single-argument functions are supported and in the syntax (lambda f x body), f is the recursive self reference as in $\lambda_{\uparrow\downarrow}$. For non-recursive functions we will use _ instead of an identifier. The

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
  (if (num?               exp)    (maybe-lift exp)
  (if (sym?               exp)    (env exp)
  (if (sym?          (car exp))
    (if (eq?  '+     (car exp))   (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
    (if (eq?  '-     (car exp))   (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
    (if (eq?  '*     (car exp))   (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
    (if (eq?  'eq?   (car exp))   (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
    (if (eq?  'if    (car exp))   (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env) ((eval (cadddr exp))
         env))
    (if (eq?  'lambda (car exp))   (maybe-lift (lambda f x ((eval (cadddr exp))
      (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
    (if (eq?  'let    (car exp))   (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
      (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
    (if (eq?  'lift   (car exp))   (lift ((eval (cadr exp)) env))
    (if (eq?  'run    (car exp))   (run  ((eval (cadr exp)) env) ((eval (caddr exp)) env))
    (if (eq?  'num?   (car exp))   (num? ((eval (cadr exp)) env))
    (if (eq?  'sym?   (car exp))   (sym? ((eval (cadr exp)) env))
    (if (eq?  'car    (car exp))   (car  ((eval (cadr exp)) env))
    (if (eq?  'cdr    (car exp))   (cdr  ((eval (cadr exp)) env))
    (if (eq?  'cons   (car exp))   (maybe-lift (cons ((eval (cadr exp)) env) ((eval (caddr exp)) env)))
    (if (eq?  'quote  (car exp))   (maybe-lift (cadr exp))
    ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))
  (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

Fig. 4. Meta-circular stage-parametric interpreter for Pink

interpreter in Figure 4 is binding-time parametric (parameter `maybe-lift`), so it can also act as a compiler. It also uses open recursion (parameter `eval`), so that it can be customized from the outside.

The key technique to enable binding-time agnostic staging for this interpreter is to put a call to `maybe-lift` around all *values* the interpreter creates: literal numbers (case `'num?`), closures (case `'lambda`), and cons cells (case `'cons`).

4.1 Correctness and Optimality

We can instantiate this polymorphic interpreter as a normal interpreter as follows, assuming that the above code is bound to an identifier `eval-poly`:

```
(define eval ((lambda ev e (((eval-poly (lambda _ e e)) ev) e)) #nil)))
```

The resulting interpreter `eval` can then be applied to quoted S-expressions:

```
(define fac-src (quote (lambda f n (if n (* n (f (- n 1))) 1))))
> ((eval fac-src) 4)   ;; => 24
```

For correctness, we should be able to add levels of interpretation, through the equation `eval = (eval eval-src)` without changing the result of the program (albeit slowing it down). Thus, we can verify that double and triple interpretation work, given a quoted `eval-src`:

```
> (((eval eval-src) fac-src) 4)           ;; => 24
> ((((eval eval-src) eval-src) fac-src) 4) ;; => 24
```

To obtain a compiler, all we have to do is to instantiate `eval-poly` as follows, with the proper `lift` operation:

```
(define evalc (lambda eval e ((((eval-poly (lambda _ e (lift e))) eval) e) #nil)))
```

```
(let star_loop (lambda star_loop m (lambda _ c (maybe-lift (lambda inner_loop s
  (if (eq?  (maybe-lift 'yes) (m s)) (maybe-lift 'yes)
  (if (eq?  (maybe-lift 'done) (car s)) (maybe-lift 'no)
  (if (eq?  '_ c) (inner_loop (cdr s))
  (if (eq?  (maybe-lift c) (car s)) (inner_loop (cdr s)) (maybe-lift 'no)))))))))
(let match_here (lambda match_here r (lambda _ s (if (eq?  'done (car r)) (maybe-lift 'yes)
  (let m (lambda _ s
      (if (eq?  '_ (car r)) (if (eq?  (maybe-lift 'done) (car s)) (maybe-lift 'no) ((match_here (cdr r)) (cdr s)))
      (if (eq?  (maybe-lift 'done) (car s)) (maybe-lift 'no)
      (if (eq?  (maybe-lift (car r)) (car s)) ((match_here (cdr r)) (cdr s)) (maybe-lift 'no)))))
    (if (eq?  'done (car (cdr r))) (m s)
    (if (eq?  '* (car (cdr r))) (((star_loop (match_here (cdr (cdr r)))) (car r)) s) (m s)))))))
(lambda _ r (if (eq?  'done (car r)) (maybe-lift (lambda _ s (maybe-lift 'yes))) (maybe-lift (match_here r))))))
```

Fig. 5. Binding-time polymorphic string matcher in Pink

And we can use `evalc` in place of `eval` to compile:

```
> (evalc fac-src)                    ;; => < code of fac in λ↑↓ >
> ((run 0 (evalc fac-src)) 4)        ;; => 24
```

Obtaining the same result as interpretation for a range of different programs and inputs, we can convince ourselves of correctness of the compilation step.

For optimality, we should be able to collapse levels of self-interpretation, even across towers. More formally, we want to verify Jones-optimality [25, 36] for the interpreter `eval`: for each program `p-src`, running the compiled program `evalc p-src` should be at least as efficient as evaluating the program `eval p-src`. Thus, we verify collapse:

```
> ((eval evalc-src) fac-src)         ;; => < code of fac >
> ((eval evalc-src) eval-src)        ;; => < code of eval >
> ((eval evalc-src) evalc-src)       ;; => < code of evalc >
```

And even further, for as many levels as we like:

```
> (((eval eval-src) evalc-src) fac-src) ;; => < code of fac >
```

The key pattern here is that all the base interpreters, i.e. `(eval eval-src)` are instantiated in actual interpretation mode, but the final interpreter operates as a compiler. Thus, the base interpreters are merely *interpreting* the staging commands of the target compiler.

Compiling User-Level Languages. We can also add evaluators to the tower at the user level, for example through DSLs. Let us exercise this pattern with our string matcher acting as the top compiler in a chain: we obtain a string matching compiler that operates through arbitrarily many levels of self-interpretation of the base evaluator. Figure 5 shows a more complete string matcher written directly in Pink and handling wildcard and repeat patterns compared to Section 3.1.

4.2 Deriving Translators from Heterogeneous Towers

4.2.1 Instrumenting Execution. Let us consider an evaluator that logs accesses to any variable named `n`. We simply change the variable line of our evaluator:

```
;; ... old ...                     ;; ... new ...
(if (sym? exp) (env exp) ...       (if (sym? exp) (if (eq? 'n exp) (log (maybe-lift 0) (env exp)) (env exp)) ...
```

The side-effecting function `log` is an extension of $\lambda_{\uparrow\downarrow}$. It prints its second argument value and returns it, and is lifted into code only if the first argument is code (like `run`). We use `maybe-lift` so that we introduce the logging in the generated code if the string matcher is compiled.

```
> (define fac-src '(lambda f n (if n (* n (f (- n 1))) 1)))
> (evalc fac-src) ;; =>
(lambda f0 x1
  (let x2 (if x1
    (let x3 (- x1 1)
      (let x4 (f0 x3)
        (let x5 (* x2 x4)
          x5)))
    1) x2))
```

```
> (trace-n-evalc fac-src) ;; =>
(lambda f0 x1
  (let x2 (log 0 x1)
    (let x3 (if x2
      (let x3 (log 0 x1)
        (let x4 (log 0 x1)
          (let x5 (- x4 1)
            (let x6 (f0 x5)
              (let x7 (* x3 x6)
                x7)))))
      1) x3)))
```

```
> (cps-evalc fac-src) ;; =>
(lambda f0 x1 (lambda f2 x3 (if x1
  (let x4 (- x1 1)
    (let x5 (f0 x4)
      (let x6 (lambda f6 x7
        (let x8 (* x1 x7)
          (x3 x8)))
       (x5 x6))))
  (x3 1))))
```

Fig. 6. Code for factorial in $\lambda_{\uparrow\downarrow}$: source (top), after plain compilation (left), tracing variable accesses (middle), cps conversion (right). Code is shown in Pink syntax for readability.

Now, we can construct a tracing compiler `trace-n-evalc` using the same pattern as before. If we call it on the definition of `fac`, we get the code for `fac` transformed so that for each occurrence of the variable named `n`, we get an additional call to the `log` function.

```
> (trace-n-evalc fac-src) ;; => < code of fac with extra log calls >
```

The $\lambda_{\uparrow\downarrow}$ code for `fac`, with extra log calls as transformed by tracing variables named `n` is shown in Figure 6. As we see highlighted in gray, there are three additional log calls, one initially (for the variable `n` in the conditional), and two more in the recursive branch. Due to the additional let-bindings, some de Bruijn variable names are shifted accordingly.

The same approach applies to any user program, for example our string matcher from Figure 5. If we use a tracing interpreter in the middle of a chain for the string matcher, we can generate code for a particular regular expression that is instrumented. This instrumented code could print a trace of the `match_here` calls and arguments during a run of the matcher, which explains the backtracking structure and which part of the pattern is currently being matched.

Going a step further, the use of `maybe-lift` turns the derived interpreter into a general-purpose transformer, so that when we pass the string matcher program as input, we get a modified string matcher back (as code), which will, when we pass it a regular expression, generate instrumented code. So when we run the code, we get more or less the same trace as before.

Thus, we show source to source translation of staged code, where what happens in the future stage is changed.

4.2.2 CPS Transform. An interpreter in continuation-passing style (CPS) leads to a CPS transformer via staging or partial evaluation [35]. We turn our stage-polymorphic evaluator into an evaluator in CPS by explicitly passing the continuation as an additional argument `k`:

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env (lambda _ k ...
```

Most cases are straightforward, and do not interfere with staging; here are the first three:

```
  ...
  (if (num?         exp)     (k (maybe-lift exp))
  (if (sym?         exp)     (k (env exp))
  (if (sym?    (car exp))
  (if (eq?  '+ (car exp))    (((eval (cadr exp)) env) (lambda _ v1 (((eval (caddr exp)) env) (lambda _ v2
                             (k (+ v1 v2))))))
  ...
```

The lambda and application cases are interesting from a staging point of view, because they are the only ones that `maybe-lift` the continuation:

```
...
(if (eq?  'lambda (car exp)) (k (maybe-lift (lambda f x (maybe-lift ((eval (cadddr exp))
    (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))))
... ;; application
(((eval (cadr exp)) env) (lambda _ v2 (((env (car exp)) v2) (maybe-lift (lambda _ x (k x))))))
...
```

Armed with such an evaluator, we can create `cps-evalc`, our compiler / CPS transformer. The resulting $\lambda_{\uparrow\downarrow}$ code for factorial in CPS (on top of the usual ANF through reflect/reify) is shown in Figure 6. Each lambda takes an additional curried argument for the continuation. All function calls are in tail position, with inner lambdas passed as continuation arguments.

4.2.3 Correctness and Optimality of Transformation. We have considered correctness and optimality of unmodified metacircular towers in Section 4.1. What can we say about interpreters that implement program transformations, for example CPS conversion or tracing? Then specialization should perform the transformation, but not introduce extraneous overhead and a tower of multiple such interpreters should apply a series of transformations. A possible way to think about this is as Jones-optimality modulo projection: there exists a self-interpreter that, when specialized, realizes a certain projection on the space of programs (i.e., implements a certain program transformation). Regular Jones-optimality is the special case for the identity transform.

5 TOWARDS REFLECTIVE TOWERS

Up until now, we have considered towers consisting of cleanly separated levels. We now turn Pink into a proper, albeit simple, reflective language, which means that programs will be able to observe the behavior of a running interpreter anywhere in a tower of Pink interpreters.

5.1 Execute-at-Metalevel

To do so, we add a construct `EM`, short for `execute-at-metalevel`, inspired by the reflective language `Black` [3]. Invoking `(EM e)` will execute the expression `e` *as if it were part of the interpreter code*. Here is an example:

```
> (eval (quote ((lambda f x (EM (* 6 (env 'x)))) 4))) ;; => 24
```

The `EM` call executes the expression at the meta level. Thus, the user program's environment is in scope under the name `env` and the syntactic lookup for name `'x` yields `4` – the argument value.

It is instructive to run the same example in compiled mode:

```
> (evalc (quote ((lambda f x (EM (* (lift 6) (env 'x)))) 4))) ;; => (lambda f x (* 6 x))
```

The body of `EM` is again executed at the meta level, which means that it now runs *at compile time*. Hence, we need to lift any values that are supposed to become part of the compiled code. In the example we used `lift` explicitly, but we could have used `maybe-lift` just as well. With `EM` and `maybe-lift`, we have a meta-programming facility that can serve both as runtime reflection in interpreted mode, and essentially function as a macro system in compiled mode.

5.1.1 Implementing EM. How can we implement `EM` in a Pink tower? For towers of arbitrary height, we need to add the following meta-circular case to the Pink interpreter:

```
(lambda _ maybe-lift (lambda tie eval (lambda _ exp (lambda _ env
  ...
  (if (sym?       (car exp))
    (if (eq?  'EM  (car exp)) (let e (car (cdr exp)) (EM ((eval (env 'e)) env)))
  ...
```

As we can see, the implementation of `EM` takes its unevaluated argument, and executes it *right there*, in the interpreter code, by delegating to evaluation one level up the tower. However, `EM` is not supported natively by $\lambda_{\uparrow\downarrow}$. Thus, bootstrapping the tower necessitates a *different* implementation, in terms of $\lambda_{\uparrow\downarrow}$, at the edge of the tower:

```
(lambda _ maybe-lift (lambda tie eval (lambda _ exp (lambda _ env
  ...
  (if (sym?       (car exp))
    (if (eq?  'EM  (car exp)) (run (maybe-lift 0) (trans (car (cdr exp))))
  ...
```

To recall, `trans` is the function that translates a quoted S-expression into $\lambda_{\uparrow\downarrow}$ code. The use of `maybe-lift` as argument for `run` ensures that we remain polymorphic over compiling vs interpreting code. Once the tower is bootstrapped in this way, all further levels can use the meta-circular implementation above.

5.1.2 Modifying the Tower Structure. It is always possible to launch new tower levels by calling `eval` (or a different interpreter) on a given quoted expression, increasing the height of the tower. With `EM`, an argument to `EM` can choose to evaluate a subexpression at the *current* user level by invoking `eval`, which is in scope in the interpreter code, just like `env` in the example above. But `EM` can also choose to modify the currently executing tower by launching a *different* interpreter recursively, with added cases for new functionality, configured to trace all operations, or modified in some other way. In contrast to launching new levels of evaluation, `EM` permits us to *replace* the currently executing interpreters within a given scope. Finally, when using the CPS Pink interpreter, `EM` can also be implemented to discard the current continuation `k` of the interpreter. This will effectively terminate the current user level and reduce the height of the tower.

As an example, we use a scoped modification via `EM` instead of a whole new evaluator to achieve tracing like in Section 4.2.1.

```
> (eval (quote ((EM (((lambda ev exp (lambda _ env
    (if (if (sym? exp) (eq? 'n exp) 0) (log ((eval exp) env)) (((tie ev) exp) env))))
    '(lambda f n (if n (* n (f (- n 1))) 1))) env)) 4))) ;; => 24
;; prints 4, 4, 4, 3, 3, 3, 2, 2, 2, 1, 1, 1, 0
```

5.1.3 Language Extensions in User Code. Finally, `EM` can expose inner workings of an interpreter through expressive user-facing APIs. In contrast to non-reflective languages, such APIs do not have to be baked into the core language. As a concrete example, when we combine the CPS Pink interpreter (see Section 4.2.2) with execution at the meta level (`EM`), we can implement a range of control operators such as `call/cc` or Danvy and Filinski's `shift` and `reset` [14] at the user level. The implementation of `call/cc` is as follows:

```
(define call/cc (lambda _ f (EM (
 ((env 'f) (maybe-lift (lambda _ v (maybe-lift (lambda _ k1 (k1 (k v)))))))
 (maybe-lift (lambda _ x x))))))

> (cps-eval (quote (+ 3 (call/cc (lambda _ k (k (k (k 1)))))))) ;; => 10
```

Note that the free variable `k` inside of the `EM` expression refers to the meta-level variable which holds the user-level continuation (similar to `env` above). The function of the `call/cc` expression is passed in that continuation, suitably packaged. Control operators like `call/cc` or `shift` and `reset` can serve as a basis for further high-level abstractions such as nondeterministic or probabilistic execution, entirely implemented in user code.

5.2 Compiling under Persistent Semantic Modifications

Our starting point for Pink was a tower where the choice of interpretation vs compilation was not observable by user code (see Section 4.1). With `EM` already, user code can execute at compile time, which may lead to observably different side effects. A key question now is what is the visibility of changes to the tower semantics in interpreted vs compiled mode.

In a fully reflective setting, we want to go as far as allowing user code to change the currently running tower persistently, and in completely unforeseen ways. If we swap out the currently running `eval` function for another one (assuming that $\lambda_{\uparrow\downarrow}$ and Pink are suitably extended with mutable state), then all expressions that are evaluated in the future should obey the new semantics. In interpreted mode this is the default behavior. But how should such semantic changes, which may depend on the flow of execution in a user program, interact with its compilation? The short answer is that they can't—as already observed by Asai [2], compilation in a reflective tower is necessarily with respect to a semantics that is known at the expression's definition site.[2] Thus, when compiled, semantic modifications can only have static as opposed to dynamic extent.

For this reason, it is of interest to make compilation decisions on a finer granularity, at the level of individual functions. Following Asai [2], we introduce two separate function abstractions: the normal `lambda` (interpreted, call-site semantics), and `clambda` (compiled, definition-site semantics). In contrast to Asai's Black implementation [2], where `clambda`s are compiled with respect to unmodified initial tower semantics, our `clambda`s follow the semantics at the function definition site.

To implement the `lambda` vs `clambda` split, we change Pink's `eval` so that `maybe-lift` becomes another argument for each recursive call. In addition, we package two things within this argument `l` now: `maybe-lift` as before, and also whether we are already in compilation mode or not. Now, we can provide `clambda` as a special form that compiles its body, removing all interpretative overhead, while retaining the normal interpreted `lambda` as well:

```
(lambda tie eval (lambda _ l (lambda _ exp (lambda _ env
  ...
  (if (sym? (car exp))
    (let maybe-lift (car l) (let compile-mode (cdr l)
      (if (or (eq? 'lambda (car exp)) (and (eq? 'clambda (car exp)) compile-mode))
        (maybe-lift (lambda f x (((eval l) (cadddr exp))
          (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
      (if (eq? 'clambda (car exp))
        (exec 0 (((eval (cons (lambda _ e (lift e)) 1)) (cons 'lambda (cdr exp)))
          (lambda _ y (lift-ref (env y)))))
  ...
```

To compile a function, we evaluate it as a regular `lambda` under a staged evaluator, achieved by passing in a lifting function. The function `exec` is a variant of `run` from $\lambda_{\uparrow\downarrow}$, which does not capture the additional surrounding environment when running. The function `lift-ref` enables cross-stage persistence by injecting pointer to the existing environment values into the generated code. We assume that a suitable implementation is added to $\lambda_{\uparrow\downarrow}$.

6 PURPLE: REFLECTION À LA BLACK

We now consider a more powerful reflective tower, where we can not only launch new interpreters with modified semantics, but also modify the semantics of the *currently executing* interpreter (tower). Unlike before, this reflective tower has infinitely many levels of meta-interpretation. The tower still

[2] A possible way out would be to look at deoptimization and re-compilation techniques from JIT compilers, but our hope is that future work will show how those mechanisms can be expressed in user code based on the facilities presented here.

executes in finite time, because when executing an unmodified meta level, the tower can fall back to built-in semantics. Another way to think about this is that meta-level are spawned lazily, as they are accessed and modified by constructs like `EM`. In the previous examples in Pink, we were able to raise towers above the user level; here, we will be able to modify the semantics of an executing tower, changing the foundations at arbitrary depths.

Our language, dubbed Purple, is heavily inspired by Asai's reflective language Black [3]. In the spirit of Black, there are a couple of important differences in comparison to Pink: Purple is conceptually infinite as a tower; it allows persistent mutable modifications; it has a richer API (i.e., user-observable interpreter); it is designed to closely match Black, which was proposed previously, including the open challenge of compilation; and it is based on an existing practical multi-stage programming framework (LMS [47]). While we could have grown Purple from Pink, we found it more interesting to investigate if we can apply the idea of stage polymorphism end-to-end in the setting of an existing language (Black) and an existing staging toolkit (LMS).

In a fully reflective programming language such as Black [3], programs are interpreted by an infinite tower of meta-circular interpreters. Each level of the tower can be accessed and modified, so the semantics of the language changes dynamically during execution. Previous work [2] used MetaOCaml [9] to stage the built-in interpreter, and thus enable compilation of functions with respect to the built-in semantics of the tower. But a key question was left open [2]: can we specialize a function with respect to the *current* semantics of the tower, which is (1) possibly user-modified, and (2) possibly also compiled via staging?

Here, we answer in the affirmative, showing that, using polymorphic staging, it is possible to compile a user program under modified, possibly also compiled, semantics.

An Example. At the user-level, we define the usual function for the Fibonacci sequence.

```
(define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
> (fib 7) ;; => 13
```

At the meta-level, we change the evaluation of variables so that it increments a meta-level counter when a variable name is `n`. Like in Pink (Section 5.1), the special form `EM` shifts the tower up, so that its argument executes at the meta-level. By changing the definition of the function `eval-var`, we modify the meaning of evaluating a variable one level down. Each interpreter function such as `eval-var` takes three arguments: the expression, environment and continuation from the level below.

```
(EM (begin (define counter 0) (define old-eval-var eval-var)
(set! eval-var (clambda (e r k) (if (eq? e 'n) (set! counter (+ counter 1))) (old-eval-var e r k)))))
> (fib 7) ;; => 13
> (EM counter) ;; => 102
```

Like in Pink (Section 5.2), we can compile a function by defining it with `clambda` instead of `lambda` (as we do above for `eval-var`). Our goal is that the behavior of a `clambda` matches that of a `lambda` when applied, assuming the *current* semantics remain fixed.

```
(set! fib (clambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
> (EM (set! counter 0))
> (fib 7) ;; => 13
> (EM counter) ;; => 102
```

On the other hand, if we undo the meta-level changes, the compiled function still updates the counter. If we re-compile the function under the current semantics, it stops updating the counter.

```
(EM (set! eval-var old-eval-var))
> (EM (set! counter 0))
```

```
> (fib 7) ;; => 13
> (EM counter) ;; => 102
(set! fib (clambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
> (EM (set! counter 0))
> (fib 7) ;; => 13
> (EM counter) ;; => 0
```

As for the generated code, the `fib` function compiled under the modified semantics has extra code (summarized in black) for incrementing the counter for each occurrence of the variable `n` in its body, but is otherwise similar to the code (summarized in gray) compiled under the original semantics. The syntax for the generated code is presented early in Section 9 (see `trait Ops`).

```
{(k, xs) => _app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_1 => _cell_set(<cell counter>, c_1)
_app(‘<, _cons(_car(xs), ‘(2)), _cont{v_1 => _if(_true(v_1),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_2 => _cell_set(<cell counter>, c_2)
_app(k, _cons(_car(xs), ‘()), _cont{v_2 => v_2})}),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_3 => _cell_set(<cell counter>, c_3)
_app(‘-, _cons(_car(xs), ‘(1)), _cont{v_3 => _app(_cell_read(<cell fib>), _cons(v_3, ‘()), _cont{v_4 =>
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_4 => _cell_set(<cell counter>, c_4)
_app(‘-, _cons(_car(xs), ‘(2)), _cont{v_5 => _app(_cell_read(<cell fib>), _cons(v_5, ‘()), _cont{v_6 =>
_app(‘+, _cons(v_4, _cons(v_6, ‘())), _cont{v_7 => _app(k, _cons(v_7, ‘()), _cont{v_8 => v_8})})})})})})})})})}
```

In the generated code, mutable cells are directly referenced, outside the concepts (such as environments) explicitly represented in the user-accessible interpreters. The modified interpreter function from the meta-level is inlined into code compiled for the user-level. Compilation collapses the levels of the tower.

Compilation of User Code. For our use case, we want to use interpreter functions, *both* to interpret expressions and to compile the body of `clambda` expressions. In case of compilation, the body expression is known (static); however, the actual arguments and meta-continuation are not known (dynamic), since they are only supplied when the function is called. Furthermore, user-defined compiled functions should be usable as or in interpreter functions, in both interpretation and compilation mode. Hence, compilation should produce code that can also produce code. Similarly, first-class objects with code, such as functions and continuations, should be usable in both modes (interpretation and compilation), regardless of which mode they were created in. In summary, we want our functions to be polymorphic over whether they interpret or compile, and also generate code that is polymorphic in the same way.

7 PURPLE / BLACK EXAMPLES

In this section, we show several examples implemented in our system. For all of these, unless otherwise specified, using `lambda` or `clambda` does not alter the semantics.

7.1 Instrumentation

We can extend the initial example, so that we have an `instr` special form, that count calls to several meta-level functions, and prints a nice summary.

Here is a general hook to add a special form:

```
(EM (define add-app-hook! (lambda (n ev)
  (let ((original-eval-application eval-application))
      (set! eval-application (lambda (exp env cont)
        (if (eq? (car exp) n) (ev exp env cont)
          (original-eval-application exp env cont))))))))
```

7.2 Stack Inspection

There is a cute trick [15] to construct `(cnv xs ys) = (zip xs (reverse ys))` in `n` recursive calls and no auxiliary data list, where `xs` and `ys` are lists of size `n`.

```
(define walk (lambda (xs ys) (if (null? xs) (cons '() ys)
  (let ((rys (walk (cdr xs) ys))) (let ((r (car rys)) (ys (cdr rys)))
    (cons (cons (cons (car xs) (car ys)) r) (cdr ys)))))))
(define cnv (lambda (xs ys) (car (walk xs ys))))
```

To understand the trick, it is helpful to visualize the stack. So we create a special form `taba`, which takes a list of function names to monitor and an expression to evaluate under temporarily modified semantics that instrument the stack for monitored function calls.

```
(EM (begin
(define eval-taba-call (lambda (add! original-eval-application) (lambda (exp env cont)
  (eval-list (cdr exp) env (lambda (ans-args) (original-eval-application exp env
    (lambda (ans) (add! ans-args ans) (cont ans))))))))
(define eval-taba (lambda (fns) (lambda (exp env cont)
  (let ((original-eval-application eval-application) (stack '()))
  (map (lambda (fn) (add-app-hook! fn (eval-taba-call (lambda (ans-args ans)
    (set! stack (cons (list fn ans-args ans) stack))) eval-application))) fns)
  (base-eval exp env (lambda (ans) (set! eval-application original-eval-application)
    (cont (list ans stack))))))))
(add-app-hook! 'taba (lambda (exp env cont) ((eval-taba (car (cdr exp))) (car (cdr (cdr exp))) env cont)))))
```

Using our special form on a particular instance of the problem, we see what happens when we go There And Back Again (TABA). As we walk down the first list, we push elements on the stack, that we pair with the elements of the second list explored on the way up.

```
> (taba (cnv walk) (cnv '(1 2 3) '(a b c))) ;; =>
;; (((1 . c) (2 . b) (3 . a))
;; ((cnv ((1 2 3) (a b c)) ((1 . c) (2 . b) (3 . a)))
;; (walk ((1 2 3) (a b c)) (((1 . c) (2 . b) (3 . a))))
;; (walk ((2 3) (a b c)) (((2 . b) (3 . a)) c))
;; (walk ((3) (a b c)) (((3 . a)) b c))
;; (walk (() (a b c)) ((() a b c))))
```

Note that since the `taba` special form modifies the semantics temporarily, it won't be able to monitor any already compiled functions. Still, as expected, functions that are compiled inside the `taba` expression will behave according the monitoring semantics.

7.3 Reifiers

In towers of interpreters, a reifier is a way to go up the tower and get a reified structure for the current computation from below. From level n, the expression `((delta (e r k) body...) args...)` evaluates the expression `body...` with the environment from level $n + 1$, with `e` bound to the unevaluated expression `args...`, `r` bound to the environment from level n, and `k` to the continuation from level n. Within the body, `(meaning e r k)` can be used to reflect back.

We can use `delta` to reify the continuation, like in Scheme's `call/cc`:

```
(define call/cc (lambda (f) ((delta (e r k) (k ((meaning 'f r (lambda (v) v)) k))))))
> (+ 1 (call/cc (lambda (k) 0)))                    ;; 1
> (+ 1 (call/cc (lambda (k) (k 0))))                ;; 1
> (+ 1 (call/cc (lambda (k) (begin (k 1) (k 3)))))  ;; 2
> (+ 1 (call/cc (lambda (k) (begin (k (k 1)) (k 3))))) ;; 2
```

First, at the meta-level, we need a way to reify the current environment, the one from the same meta-level. So we add this facility by changing the meta-meta-level:

```
(EM (EM (begin (define old-eval-var eval-var)
(set! eval-var (clambda (e r k) (if (eq? '_env e) (k r) (old-eval-var e r k)))))))
```

Now, at the meta-level, we provide the definition of `delta` by recognizing its pattern of application. This is a simplification: we do not turn `delta` into a first-class object.

```
(EM (begin (define delta? (lambda (e) (if (pair? e) (if (pair? (car e)) (eq? 'delta (car (car e))) #f) #f)))
(define apply-delta (lambda (e r k) (let ((operator (car e)) (operand (cdr e)))
  (let ((delta-params (car (cdr operator))) (delta-body (cdr (cdr operator))))
    (eval-begin delta-body (extend _env delta-params (list operand r k)) id-cont)))))
(define old-eval-application eval-application)
(set! eval-application (lambda (e r k) (if (delta? e) (apply-delta e r k) (old-eval-application e r k))))
(define meaning base-eval)))
```

7.4 Meta-Level Undo

We can implement undo at the meta-level, so that it is easy to experiment with changes and roll back to a previous state.

At the meta-meta-level, we change `eval_var` to provide the `_env` reifier like for `delta`. In addition, we also monitor `eval_set!` to keep track of changes at the meta-level:

```
(EM (EM (begin
(define undo-list '())
(define old-eval-set! eval-set!)
(set! eval-set! (clambda (e r k) (let ((name (car (cdr e))))
  (eval-var name r (lambda (v) (set! undo-list (cons (cons name v) undo-list)) (old-eval-set! e r k))))))
(define reflect-undo! (clambda (r) (if (null? undo-list) '() (begin
  (old-eval-set! (list 'set! (car (car undo-list)) (list 'quote (cdr (car undo-list)))) r (lambda (v) v))
  (set! undo-list (cdr undo-list)))))))))
```

At the meta-level, we just provide a nice `undo!` function:

```
(EM (define undo! (clambda () ((EM reflect-undo!) _env))))
```

Here is a use example of `undo!`:

```
(EM (define old-eval-var eval-var))
(EM (set! eval-var (clambda (e r k) (if (eq? e 'n) (k 0) (old-eval-var e r k)))))
(define n 1)
> n                                ;; 0
> (EM (eq? old-eval-var eval-var)) ;; #f
(EM (undo!))
> n                                ;; 1
> (EM (eq? old-eval-var eval-var)) ;; #t
```

7.5 Collapsing User-Level Interpreters

Last but not least, we show how our running string matcher example can be expressed and collapsed as a user-level interpreter:

```
(define matches (clambda (r) (clambda (s) (if (null? r) #t (if (null? s) #f
  (if (eq? (car r) (car s)) ((matches (cdr r)) (cdr s)) #f))))))
> ((matches '(a b)) '(a c))   ;; => #f
> ((matches '(a b)) '(a b))   ;; => #t
> ((matches '(a b)) '(a b c)) ;; => #t
```

We can generate code for matching a particular prefix as follows:

```
(define start_ab ((clambda () (matches '(a b)))))
```

The resulting code is the same as show in earlier sections.

We can combine this user-level interpreter with a user-modified meta-level too. For example, we can modify the meta-level `eval-var` to count evaluations of variables named `r` or `s`, similar to the initial example at the beginning of Section 6.

```
> ((matches '(a b)) '(a c))   ;; => #f (r: 5, s: 5)
> ((matches '(a b)) '(a b))   ;; => #t (r: 7, s: 6)
> ((matches '(a b)) '(a b c)) ;; => #t (r: 7, s: 6)
```

8 FROM $\lambda_{\uparrow\downarrow}$ TO LMS

Purple is internally implemented in Scala, using the Lightweight Modular Staging (LMS) framework [47]. Before we dive further into the implementation of Purple, we first discuss how to achieve the essential ingredient of stage polymorphism in LMS [47], We move beyond $\lambda_{\uparrow\downarrow}$ to LMS for flexibility and variety, demonstrating that the ideas prototyped in the context of $\lambda_{\uparrow\downarrow}$ scale to LMS.

Having seen in Section 3.1 how $\lambda_{\uparrow\downarrow}$ can serve as a model for untyped front-ends, we now turn our attention to typed models of multi-stage evaluation, showing how $\lambda_{\uparrow\downarrow}$ relates to LMS. LMS uses a type constructor `Rep[T]` to designate future-stage expressions, i.e. those that evaluate to `Code` values in $\lambda_{\uparrow\downarrow}$, and provides overloaded method on such `Rep[T]` values, e.g. `+` for `Rep[Int]`. Thus, Scala's local type inference and overloading resolution performs a kind of local binding-time analysis.

Essentially [45] we take the `evalms` function from Section 3 and turn it inside out, from an interpreter over an initial term language `Exp` to an evaluator in tagless-final [10] style. Doing so, we can assign the following overloaded type signatures to the $\lambda_{\uparrow\downarrow}$operations:

```
def lift(v: Int): Rep[Int]
def lift[A,B](v: (Rep[A], Rep[B])): Rep[(A,B)]
def lift[A,B](v: Rep[A]=>Rep[B]): Rep[A=>B]
def lift[A](v: Rep[A]): Rep[Rep[A]]
def run[A](b: Any, e: Rep[A]):A
def run[A](b: Rep[Any], e: Rep[Rep[A]]): Rep[A]
def app[A,B](f: A=>B,x:A): B
def app[A,B](f: Rep[A=>B], x: Rep[A]): Rep[B]
def if_[A](c: Boolean, a: =>A, b: => A): A
def if_[A](c: Rep[Boolean], a: =>Rep[A], b: => Rep[A]): Rep[A]
def plus(x: Int, y: Int): Int
def plus(x: Rep[Int], y: Rep[Int]): Rep[Int]
```

Note that there are no introduction forms anymore, as these correspond to normal Scala values.

Looking closely at the structure of the interpreter of Figure 2, it turns out that fortunately, we never need to get our hands on *unevaluated* Scala expressions. This is not a given for a multi-level language: the fact that tools like LMS operate as libraries inside a general-purpose host language is a key difference from offline partial evaluation systems that operate on the program text. We identify `Rep[T] = Exp` and note how the implementations carry over immediately from `evalms` and `lift` of Figure 2, and how the normal Scala evaluation takes over the role of present-stage evaluation from `evalms`, e.g.:

```
def app[A,B](f: A=>B, x: A): B = f(x)
def app[A,B](f: Rep[A=>B], x: Rep[A]): Rep[B] = reflect(App(s1,s2))
```

Note also the use of `reflect` instead of `reflectc` since we no longer need a specific `Code` data type, and use plain `Exp` instead.

This API corresponds almost directly to the actual implementation in LMS. Behind the `reflect` and `reify` API, the internal representation of LMS is different though. Rather than as simple expression trees, LMS has a graph-based intermediate representation (IR) that directly supports code motion, common subexpression elimination, dead code elimination and a range of other

optimizations [11, 47, 48]. LMS also makes pervasive use of smart constructors to perform rewriting while the IR is constructed.

For comparison with Section 3.1, here is the staged string matcher in LMS, using regular and staged lists:

```
def matches(r: List[Char])(s: Rep[List[Char]) = {
  if (r.isEmpty) lift(true)  else if (s.isEmpty) lift(false)  else
  if (lift(r.head) == s.head) matches(r.tail, s.tail) else lift(false) }
```

Here is the specialization to pattern `a b`:

```
val start_ab = run(0,lift(matches(List('a','b'))))
```

The `run` construct built on top of LMS will generate Scala source code, compile it at runtime, and load the generated class files into the running JVM. Note that the explicit calls to `lift` for primitives could be dropped by declaring the corresponding `lift` function as `implicit`. We have left them in here for clarity and for consistency with $\lambda_{\uparrow\downarrow}$.

8.1 Stage Polymorphism with Type Classes

The question now is, how to achieve the same flavor of stage polymorphism as in Section 3.2 ? We again consider LMS, which relies on the normal Scala type system without specific support for polymorphic binding time abstraction and application operators as in [32]. So far, we have a fixed type distinction between normal types `T` and staged types `Rep[T]`, and we have overloaded methods based on those static types. The key idea here is to introduce another higher-kinded type `R[T]` which can be instantiated with either `T` or `Rep[T]` in a given context. But how do we get the correct operations on `R[T]` values?

It turns out that we can leverage type classes [60] to good effect. We define a type class interface:

```
trait Ops[R[_]] {
  def lift(v: Int): R[Int]
  def app[A,B](f: R[A=>B], x: R[A]): R[B]
  def plus(x: R[Int], y: R[Int]): R[Int]
  /* ... */
}
```

And corresponding instances of `Ops[NoRep]` and `Ops[Rep]` (where `NoRep[T] = T`) that delegate to the appropriate base methods on plain types or `Rep` types respectively.

```
implicit object PlainOps extends Ops[NoRep] { ... }
implicit object RepOps   extends Ops[Rep]   { ... }
```

Now we can go ahead and define the staging-time polymorphic matcher in Scala:

```
def matches[R[_]:Ops](r: List[Char])(s: R[List[Char]) = {
  val o = implicitly[Ops[R]]; import o._
  if (r.isEmpty) lift(true) else if (s.isEmpty) lift(false) else
  if (lift(r.head) == s.head) matches(r.tail, s.tail) else lift(false) }
```

Note that the line `val o = ...; import o._` could be eliminated with an additional level of indirection, defining `lift` etc. as methods outside of `Ops` that are parameterized over `R[_]:Ops`.

In the driver code, we just have to pick the correct type parameter when calling `matches`:

```
def matches-maybe-spec(r: List[Char]) = if (r.length < 20) run(0,lift(matches[Rep](r))) else matches[NoRep](r)
```

Scala's implicit resolution will pass the correct type class instance (either `PlainOps` or `RepOps`) to `matches` automatically.

9 A SKETCH OF PURPLE'S IMPLEMENTATION

Now we show some aspects of Purple's implementation, focusing on compilation and reflection. The Purple system comprises three languages:

(1) The host language, Scala, in which the built-in interpreter functions are written.
(2) The user language, which exposes the user-level and the tower structure, including all the meta-level interpreter functions.
(3) The compilation language, which is defined by the lifted operations `Ops[R]`.

Values are represented in the host language as follows:

```
Value ::= I(n: Int) | B(b: Boolean) | S(sym: String) | Str(s: String) | P(car: Value, cdr: Value)
        | Clo(params: Value, body: Value, env: Value, menv: MEnv)
        | Evalfun(key: Int) | Cell(key: String)
        | /* ... nil, primitives, continuations ... */
```

Integers, booleans, symbols, strings and pairs are completely standard. Closures hold a meta-environment `MEnv` in addition to the parameters, body, and value environment. Cells encapsulate a store location. `Evalfuns` represent a reference to a built-in interpreter function, or to a compiled user function (`clambda`).

The compilation language necessary for turning the interpreter into a compiler is small:

```
trait Ops[R[_]] {
  implicit def _lift(v: Value): R[Value]
  def _liftb(b: Boolean): R[Boolean]
  def _app(fun: R[Value], args: R[Value], cont: Value): R[Value]
  def _true(v: R[Value]): R[Boolean]
  def _if(c: R[Boolean], a: => R[Value], b: => R[Value]): R[Value]
  def _fun(f: Fun[R]): R[Value]
  def _cont(f: FunC[R]): Value
  def _cons(car: R[Value], cdr: R[Value]): R[Value]
  def _car(p: R[Value]): R[Value]
  def _cdr(p: R[Value]): R[Value]
  def _cell_new(v: R[Value], memo: String): R[Value]
  def _cell_read(c: R[Value]): R[Value]
  def _cell_set(c: R[Value], v: R[Value]): R[Value]
  def inRep: Boolean
}
```

This trait can be instantiated either for normal interpretation with the identity type constructor (`R = NoRep`) or for compilation using LMS (`R = Rep`), as discussed in Section 8.

9.1 Fixed Tower Structure

In a "classic" tower of interpreters, as in languages 3-Lisp [52], Brown [61], Blond [16], or Black [2, 3], each level has its own environment and its own continuation. Conceptually, the semantics of one level is given by the interpreter at the level above, which takes an expression, an environment and a continuation, together with a stream of all the environments and continuations of the levels above.

For compilation, we would rather avoid reasoning about level shifting, up and down, explicitly. Therefore, we drop meta-continuations in favor of one global continuation, which fits better with a model of one global "pc" counter.

Thus, our tower has a fixed level structure, where level 0 is interpreted by level 1, which is interpreted by level 2, and so on. Each level has its own global environment, containing bindings for primitives (such as `null?`, `+`, `display`) and, except level 0, for interpreter functions (such as `base-eval`, `eval-var`, `base-apply`). Each interpreter function takes as arguments an expression, an

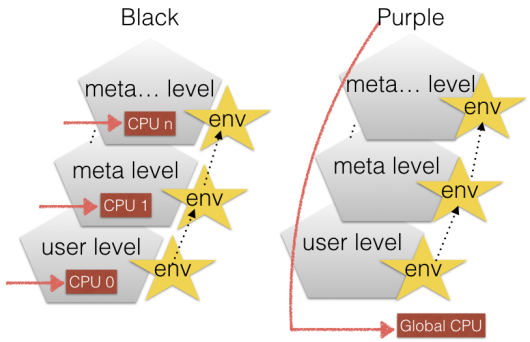

Fig. 7. Different Structures for Reflective Towers

environment and a continuation. These arguments usually come from the level below, and are manipulated as reified structures. In addition, an interpreter function statically knows about its fixed meta-environment (the stream of all environments in levels above and including its own) and is dynamically given a meta-continuation, which represents the rest of the computation or context around the interpreter function call. Similarly, closures, created from `lambda`, save not only their lexical environment, but also their lexical meta-environment.

In a classic reflective tower, shifting levels up and down needs to be made explicit (and would need to be part of the compilation language). In addition, there is this tension between the meta-state provided during compilation and the one provided during application. What if the function is applied at a different level than the one in which it was created? The tower at run-time might not even match any preconceived model since by pushing onto the meta-state, one can change the tower structure arbitrarily.

```
(begin (define where_am_i 'user) (EM (define where_am_i 'meta))
(EM (let ((old-eval-var eval-var) (__k (lambda (x) x)))
  (set! eval-var (lambda (e r k) (if (eq? e '__k) (k __k) (begin (if (eq? e '_) (set! __k k) '())
    (old-eval-var e r k)))))))
(define _ 0))
> (+ _ 1)          ;; => 1
> where_am_i       ;; => user
> (EM where_am_i) ;; => meta
> (__k 2)          ;; => 3
> where_am_i       ;; => user
> (EM where_am_i) ;; => meta (in Black: user!)
```

That calling the continuation `_k` pushes another user-level in Black may seem a bit strange and counterintuitive.

Even though the tower structure in Purple is fixed, the semantics is not, since interpreter functions can be redefined using `set!`. Hence, in the host language, we do not hard-code references to mutually-recursive interpreter functions, but look up such references in the meta-environment (via `meta_apply`).

Figure 7 summarizes the differences in structure between Black and Purple: (1) each level has a CPU in Black, while there is a single global CPU in Purple (2) in Black, the environments (represented by stars in Figure 7) are detachable from the tower structure, while they are fixed in Purple.

9.2 Initializing the Tower

Below is the code for building the tower level structure. Instead of meta-continuations, Purple uses meta-environments. The representation of interpreter functions is wrapped to look just like user-defined compiled functions. The meta-environment being created is passed lazily on to the interpreted functions during frame initialization. The meta-environment constructor takes an environment eagerly, and the next meta-environment lazily; thus, the stream of environments fixes infinitely many upper levels, potentially.

```
def binding(s: String, v: Value): Value = P(S(s), cell_new(v, s))
def init_frame_list = List(
  P(S("null?"), Prim("null?")),
  P(S("+"), Prim("+")),
  /* ... */
  P(S("display"), Prim("display")))
def init_mframe(m: => MEnv) = list_to_value(List(
  binding("eval-begin", evalfun(eval_begin_fun(m))),
  binding("eval-EM", evalfun(eval_EM_fun(m))),
  binding("eval-clambda", evalfun(eval_clambda_fun(m))),
  binding("eval-lambda", evalfun(eval_lambda_fun(m))),
  binding("eval-application", evalfun(eval_application_fun(m))),
  binding("eval-var", evalfun(eval_var_fun(m))),
  /* ... */
  binding("base-eval", evalfun(base_eval_fun(m))),
  binding("base-apply", evalfun(base_apply_fun(m)))) ++
  init_frame_list)
def init_env = cons(cell_new(init_frame, "global"), N)
def init_meta_env(m: => MEnv) = cons(cell_new(init_mframe(m), "global"), N)
def init_menv[R[_]:Ops]: MEnv = { lazy val m: MEnv = MEnv(init_meta_env(m), init_menv[R]); m }
```

Mutable cells are represented explicitly. From the code above, each initial environment is a list of one global frame. The head of the list is mutable so that more definitions can be added to the frame. The value binding of interpreter functions is also mutable, so that an interpreter can be modified.

9.3 Base Evaluation

Below is the code for the entry-point interpreter function. The function dispatches on the form of the expression, delegating to other interpreter functions accordingly. Even though the expression, environment and continuation of the level below are known (static: `Value`), the result of the function is not known (dynamic: `R[Value]`). The function calls starting with `_` (such as `_lifted`) are part of the operations for `R[_]` types, behaving differently for each instantiation type (`NoRep` vs `Rep`).

```
def base_eval[R[_]:Ops](m: MEnv, exp: Value, env: Value, cont: Value): R[Value] = {
  val o = implicitly[Ops[R]]; import o._; exp match {
    case I(_) | B(_) | Str(_) => apply_cont[R](cont, _lift(exp))
    case S(sym) => meta_apply[R](m, S("eval-var"), exp, env, cont)
    case P(S("lambda"), _) => meta_apply[R](m, S("eval-lambda"), exp, env, cont)
    case P(S("clambda"), _) => meta_apply[R](m, S("eval-clambda"), exp, env, cont)
    /* ... case let, if, begin, set!, define, quote, ... */
    case P(S("EM"), _) => meta_apply[R](m, S("eval-EM"), exp, env, cont)
    case P(fun, args) => meta_apply[R](m, S("eval-application"), exp, env, cont) }}
```

The result is potentially dynamic (`R[Value]`), so that we indeed can turn the interpreter into a compiler. Above, we use `_lift` to turn a static (known) `Value` into dynamic (though constant) `R[Value]`.

Application. The `meta_apply` function delegates to `static_apply` to actually apply the interpreter function looked up in the meta-environment. If we have a closure (from an uncompiled `lambda`), then we recursively call `meta_apply` again possibly going further up in the levels. If we have a compiled or built-in function, then we just apply it as a black-box, thus breaking out of an infinite-tower regress.

```
def meta_apply[R[_]:Ops](m: MEnv, s: Value, exp: Value, env: Value, cont: Value): R[Value] = {
  val MEnv(meta_env, meta_menv) = m; val o = implicitly[Ops[R]]; import o._
  val c@Cell(_) = env_get(meta_env, s); val fun = cell_read(c); val args = P(exp, P(env, P(cont, N)))
  static_apply[R](fun, args, id_cont[R]) }
def static_apply[R[_]:Ops](fun: Value, args: Value, cont: Value) = {
  val o = implicitly[Ops[R]]; import o._; fun match {
    case Clo(params, body, cenv, m) => meta_apply[R](m, S("eval-begin"), body,
      env_extend[R](cenv, params, args), cont)
    case Evalfun(key) => val f = funs(key).fun[R]; f(cons(cont, args))
    case Prim(p) => apply_cont[R](cont, apply_primitive(p, args))
    case _ if isCont(fun) => apply_cont[R](fun, car(args)) /* lost cont */ }}
```

In the case for `Evalfun`, the target function is looked up from a global table `funs` of compiled functions, which holds objects of type `Fun`:

```
abstract class Fun { def fun[R[_]:Ops]: R[Value] => R[Value] }
```

These functions, which represent internal interpreter functions like `base_eval` or `static_apply` itself, or user-defined `clambda`s, are stage polymorphic, and can be invoked for any given `[_]:Ops`.

Eval at Metalevel. Here is the definition of `eval_EM`, which shifts the computation up a level:

```
def eval_EM[R[_]:Ops](m: MEnv, exp: Value, env: Value, cont: Value): R[Value] = {
  val o = implicitly[Ops[R]]; import o._; val P(_, P(e, N)) = exp; val MEnv(meta_env, meta_menv) = m
  meta_apply[R](meta_menv, S("base-eval"), e, meta_env, cont) }
```

The continuation `cont` is kept when shifting up with `eval_EM`. However, `meta_apply` gives a new outer-continuation `id_cont` because the continuation from the level below becomes part of the arguments.

Compiled Lambdas. To evaluate a `clambda` expression, we might need to switch from interpretation to compilation. In any case, we evaluate the static body of the `clambda` in an extended environment, in which parameters of the `clambda` are mapped to symbolic values.

```
case class Code[R[_]](c: R[Value]) extends Value
```

The continuation with which the body returns is also dynamic, as it is passed when the function is applied. We "unwrap" the continuation argument `k`, because `meta_apply` like other interpreter functions expects a known continuation.

```
def eval_clambda[R[_]:Ops](m: MEnv, exp: Value, env: Value, cont: Value): R[Value] = {
  val o = implicitly[Ops[R]]; import o._; val P(_, P(params, body)) = exp
  def eval_body[RF[_]:Ops](kv: RF[Value]): RF[Value] = {
    val or = implicitly[Ops[RF]]; val args = or._cdr(kv); lazy val k = or._car(kv)
    meta_apply[RF](m, S("eval-begin"), body, env_extend[RF](env, params, Code(args)), unwrap_cont[RF](k)) }
  val f = if (!inRep) { /* switch to compilation mode */
    trait Program extends EvalDsl { def snippet(kv: Rep[Value]): Rep[Value] = eval_body[Rep](kv)(OpsRep) }
    val r = new EvalDslDriver with Program; r.precompile
    _lift(evalfun(r.f)) // insert into funs table
  } else { /* already in compilation mode, create a poly-stage function */
    _fun(new Fun[R] { def fun[RF[_]:Ops] ={ kv => eval_body[RF](kv) }}) }
  apply_cont[R](cont, f) }
```

10 BENCHMARKS

We show some micro benchmarks in Figure 8, computing factorial from 0 to 9, through evaluation and compilation, as well as with and without tracing. In Pink, compilation is achieved by using a lifted evaluator (e.g. `evalc`). In Purple, compilation is achieved by using `clambda`. In both Pink and Purple, tracing is achieved by incrementing a counter cell.

As expected, these benchmarks confirm that collapsing towers does of course speed up computation, exercising evaluation and collapse across standard and non-standard semantics. For overheads, we expect positive ratios, due to growing input, interpretation overhead, and tracing overhead. While the Pink system is much leaner than the Purple system, they both exhibit these trends.

	Pink								Purple							
n	e	c	e_t	c_t	$\frac{e(n)}{e(n-1)}$	$\frac{e}{c}$	$\frac{e_t}{e}$	$\frac{e_t}{c_t}$	e	c	e_t	c_t	$\frac{e(n)}{e(n-1)}$	$\frac{e}{c}$	$\frac{e_t}{e}$	$\frac{e_t}{c_t}$
0	569	52	786	146	1.00	10.98	1.38	5.40	52921	1972	34175	1999	1.00	26.83	0.65	17.10
1	1590	99	2269	517	2.79	16.05	1.43	4.39	76554	2133	75555	3145	1.45	35.89	0.99	24.03
2	2634	170	3787	803	1.66	15.46	1.44	4.71	149781	2753	175706	4708	1.96	54.41	1.17	37.32
3	3696	234	5343	1172	1.40	15.81	1.45	4.56	321412	3855	381856	6344	2.15	83.39	1.19	60.19
4	4852	313	7015	1542	1.31	15.50	1.45	4.55	444264	4449	485257	6673	1.38	99.85	1.09	72.72
5	5870	363	8431	1906	1.21	16.18	1.44	4.42	658563	8935	702719	7683	1.48	73.71	1.07	91.46
6	7095	443	10174	2319	1.21	16.01	1.43	4.39	953057	5376	1012975	15205	1.45	177.27	1.06	66.62
7	7815	501	11642	2639	1.10	15.61	1.49	4.41	1248392	6939	1689002	12636	1.31	179.91	1.35	133.67
8	9072	583	13364	3043	1.16	15.56	1.47	4.39	1847398	10232	1952642	16625	1.48	180.56	1.06	117.45
9	11470	739	14940	3399	1.26	15.52	1.30	4.40	2423174	10357	2486263	19943	1.31	233.96	1.03	124.67

Fig. 8. Benchmark contrasting `fac(n)` computations that are evaluated (e) vs collapsed (c) with standard vs tracing ($\square_t$) interpreters. The raw numbers are in ms per 100'000 iterations.

11 RELATED WORK

Partial Evaluation. Partial evaluation [36] is an automatic program specialization technique. Despite their automatic nature, most partial evaluators also provide annotations to guide specialization decisions. Some notable systems include DyC [29], an annotation-directed specializer for C, JSpec/Tempo [49], the JSC Java Supercompiler [38], and Civet [50].

Partial evaluation has addressed higher-order languages with state using similar let-insertion techniques as discussed here [7, 31, 39, 59]. Further work has studied partially static structures [41] and partially static operations [58], and compilation based on combinations of partial evaluation, staging and abstract interpretation [12, 37, 54]. Two-level languages are frequently used as a basis for describing binding-time annotated programs [36, 42].

Multi-level binding-time analysis extends binding-time analysis (BTA) from two stages to more levels [27]. Polymorphic binding-time analysis extends binding-time analysis so that some expressions can be assigned multiple stages [32]. Our kernel language $\lambda_{\uparrow\downarrow}$ complements such binding-time analyses: the stages are explicit, but can be abstracted over, just like with a polymorphic multi-level BTA. However in $\lambda_{\uparrow\downarrow}$, we want fine-grained control over multi-level computation, and hence provide a lightweight but explicit API instead of purely automatic behavior.

The "writing cogen by hand approach" is due to Birkedal et al. [6], and was developed further by Thiemann [57]. Jones made a case for program transformation via interpreter specialization [35]. Glück has shown how optimizing specializers can be derived by layering and specializing interpreters [26, 28].

Multi-stage programming. Multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [55] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [9] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [46, 47] uses types instead of syntax to identify binding times, and generates

an intermediate representation instead of target code [44]. LMS draws inspiration from earlier work such as TaskGraph [4], a C++ framework for program generation and optimization.

Reflective Towers. Smith introduced reflective towers in seminal papers on 3-Lisp [51, 52]. The motivation stems from enabling processes to inspect on their computation arbitrarily. Friedman and Wand distill the essence of reflection in Brown [22, 61], explaining reflection and reification in a self-contained semantics, which does not re-allude to reflection. Later, Jefferson and Friedman also give a simplified account for a finite tower, $\mathcal{I}_R$ [34], and at the same time, Sobel and Friedman also give an account of reflection without towers [53]. Danvy and Malmkjær present a denotational semantics of Blond [16]. Their account justifies the use of meta-continuations for a compositional semantics. As discussed earlier, our Purple reflective tower is inspired chiefly by Asai's Black [2, 3].

Aspect-Oriented Programming. Some of the non-standard semantics we cover in this paper (e.g. tracing) are reminiscent of aspect-oriented programming, which also shares some of the compilation challenges that arise in towers of interpreters [40, 56].

Program Generators. A number of high-performance program generators have been built, for example ATLAS [62] (linear algebra), FFTW [23] (discrete Fourier transform), and Spiral [43] (general linear transformations). Other systems include PetaBricks [1], and CVXgen [30].

12 CONCLUSIONS

We have shown how to collapse towers of interpreters using a stage-polymorphic multi-level lambda calculus. We have also shown that we can re-create a similar effect using LMS and polytypic programming via type classes. We have discussed several examples including novel reflective programs in Purple / Black. Looking beyond this paper, we believe that collapsing towers, in particular heterogeneous towers, has practical value. Here are some examples:

(1) It is often desirable to run other languages on closed platforms, e.g. in a web browser. For this purpose, Emscripten [63] translates LLVM code to JavaScript. Similarly, Java VMs and even entire x86 processor emulators that are able to boot Linux have been written in JavaScript. It would be great if we could run all such artifacts at full speed, e.g. a Python application executed by an x86 runtime, emulated in a JavaScript VM. Naturally, this requires not only collapsing of static calls, but also adapting to a dynamically changing environment.

(2) It can be desirable to execute code under modified semantics. Key usecases here are: (a) instrumentation/tracing for debugging, potentially with time-travel and replay facilities, (b) sandboxing for security, (c) virtualization of lower-level resources as in environments like Docker, and (d) transactional execution with atomicity, isolation, and potential rollback.

(3) Non-standard interpretations, e.g. program analysis, verification, synthesis. We would like to reuse those artifacts if they are implemented for the base language. For example, a Racket interpreter in miniKanren [8] has been shown to enable logic programming for a large class of Racket programs without translating them to a relational representation. Other examples are the Abstracting Abstract Machines (AAM) framework [33], which has recently been extended to abstract definitional interpreters [17]. For these indirect approaches to be effective, it is important to remove intermediate interpretive abstractions which would otherwise confuse the analysis.

For these use cases, our approach hints at a solution where we only need to manually lift the meta interpreter of the user level while the rest of the tower acts in a kind of pass-through mode, handing down staging commands to level 0, which needs to support stage polymorphism. Thus, we hope our work spurs further activity in implementing stage polymorphic virtual machines and collapsing towers of interpreters in the wild.

REFERENCES

[1] S. P. Amarasinghe. Petabricks: a language and compiler based on autotuning. In *High Performance Embedded Architectures and Compilers (HiPEAC)*, 2011.

[2] K. Asai. Compiling a reflective language using MetaOCaml. In *GPCE*, 2014.

[3] K. Asai, S. Matsuoka, and A. Yonezawa. Duplication and partial evaluation: For a better understanding of reflective languages. *Lisp and Symbolic Computation - Special issue on computational reflection*, 1996.

[4] O. Beckmann, A. Houghton, M. R. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.

[5] U. Berger, M. Eberl, and H. Schwichtenberg. *Normalization by Evaluation.* 1998.

[6] L. Birkedal and M. Welinder. Hand-writing program generator generators. In *International Symposium on Programming Language Implementation and Logic Programming*, 1994.

[7] A. Bondorf. *Self-applicable partial evaluation.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1990.

[8] W. E. Byrd, E. Holk, and D. P. Friedman. miniKanren, live and untagged: quine generation via relational interpreters (programming pearl). In *Scheme*, 2012.

[9] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. GPCE, 2003.

[10] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[11] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

[12] C. Consel and S.-C. Khoo. Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.*, 15(3):463–493, 1993.

[13] O. Danvy. Type-directed partial evaluation. In *POPL*, 1996.

[14] O. Danvy and A. Filinski. Abstracting control. In *Proc. LFP'90*, 1990.

[15] O. Danvy and M. Goldberg. There and back again. In *ICFP*, 2002.

[16] O. Danvy and K. Malmkjær. Intensions and extensions in a reflective tower. In *Lisp and Functional Programming*, 1988.

[17] D. Darais, N. Labich, P. C. Nguyen, and D. Van Horn. Abstracting definitional interpreters. 2016.

[18] A. P. Ershov. On the essence of compilation. *Formal Description of Programming Concepts*, pages 391–420, 1978.

[19] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[20] A. Filinski. Representing monads. In H. Boehm, B. Lang, and D. M. Yellin, editors, *POPL*, 1994.

[21] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[22] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *LISP and Functional Programming*, pages 348–355, 1984.

[23] M. Frigo. A fast fourier transform compiler. In *PLDI*, 1999.

[24] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan*, 54-C(8):721–728, 1971.

[25] R. Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *PEPM*, 2002.

[26] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *PLILP*, 1995.

[27] R. Glück and J. Jørgensen. Fast binding-time analysis for multi-level specialization. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, 1996.

[28] R. Glück and J. Jørgensen. Multi-level specialization (extended abstract). In *Partial Evaluation*, 1998.

[29] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000.

[30] M. Hanger, T. A. Johansen, G. K. Mykland, and A. Skullestad. Dynamic model predictive control allocation using CVXGEN. In *9th IEEE International Conference on Control and Automation, ICCA 2011, Santiago, Chile, December 19-21, 2011*, pages 417–422. IEEE, 2011.

[31] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

[32] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In *ESOP*, 1994.

[33] D. V. Horn and M. Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *CACM*, 54(9):101–109, 2011.

[34] S. Jefferson and D. P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2-3):181–202, 1996.

[35] N. D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52:307–339, 2004.

[36] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[37] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT*, 2004.

[38] A. V. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2009.
[39] J. L. Lawall and P. Thiemann. Sound specialization in the presence of computational effects. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190. Springer, 1997.
[40] H. Masuhara, G. Kiczales, and C. Dutchyn. *A Compilation and Optimization Model for Aspect-Oriented Programs.* 2003.
[41] T. A. Mogensen. Partially static structures in a self-applicable partial evaluator. 1988.
[42] F. Nielson and H. R. Nielson. Multi-level lambda-calculi: An algebraic description. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, volume 1110 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 1996.
[43] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.
[44] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming.* PhD thesis, EPFL, 2012.
[45] T. Rompf. The essence of multi-stage evaluation in LMS. In *WadlerFest*, 2016.
[46] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE, pages 127–136, New York, NY, USA, 2010. ACM.
[47] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *CACM*, 55(6):121–130, 2012.
[48] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. In *POPL*, 2013.
[49] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.
[50] A. Shali and W. R. Cook. Hybrid partial evaluation. OOPSLA, 2011.
[51] B. C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, 1982.
[52] B. C. Smith. Reflection and semantics in lisp. In *POPL*, 1984.
[53] J. M. Sobel and D. P. Friedman. An introduction to reflection-oriented programming.
[54] M. Sperber and P. Thiemann. Realistic compilation by partial evaluation. In *PLDI*, 1996.
[55] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
[56] E. Tanter. Execution levels for aspect-oriented programming. In *AOSD*, 2010.
[57] P. Thiemann. Cogen in six lines. In *ICFP*, 1996.
[58] P. Thiemann. Partially static operations. In *PEPM*, 2013.
[59] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state. Technical report, 1999.
[60] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989.
[61] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Lisp and Functional Programming*, 1986.
[62] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
[63] A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA Companion*, pages 301–312. ACM, 2011.