Collapsing Towers of Interpreters

NADA AMIN, University of Cambridge, UK
TIARK ROMPF, Purdue University, USA

Given a tower of interpreters, i.e., a sequence of multiple interpreters interpreting one another as input programs, we aim to collapse this tower into a compiler that removes all interpretive overhead and runs in a single pass. In the real world, a use case might be Python code executed by an x86 runtime, on a CPU emulated in a JavaScript VM, running on an ARM CPU. Collapsing such a tower can not only exponentially improve runtime performance, but also enable the use of base-language tools for interpreted programs, e.g., for analysis and verification. In this paper, we lay the foundations in an idealized but realistic setting.

We present a multi-level lambda calculus that features *staging constructs* and *stage polymorphism*: based on runtime parameters, an evaluator either executes source code (thereby acting as an interpreter) or generates code (thereby acting as a compiler). We identify stage polymorphism, a programming model from the domain of high-performance program generators, as the key mechanism to make such interpreters compose in a collapsible way.

We present Pink, a meta-circular Lisp-like evaluator 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 an interpreter to base code, building program transformers from modified interpreters, and others. We develop these ideas further to include reflection and reification, culminating in Purple, a reflective language inspired by Brown, Blond, and Black, which realizes a conceptually infinite tower, where every aspect of the semantics can change dynamically. Addressing an open challenge, we show how user programs can be compiled and recompiled under user-modified semantics.

CCS Concepts: • **Software and its engineering → Compilers**; *Interpreters*; *General programming languages*;

Additional Key Words and Phrases: interpreter, compiler, staging, reflection, Scala, Lisp

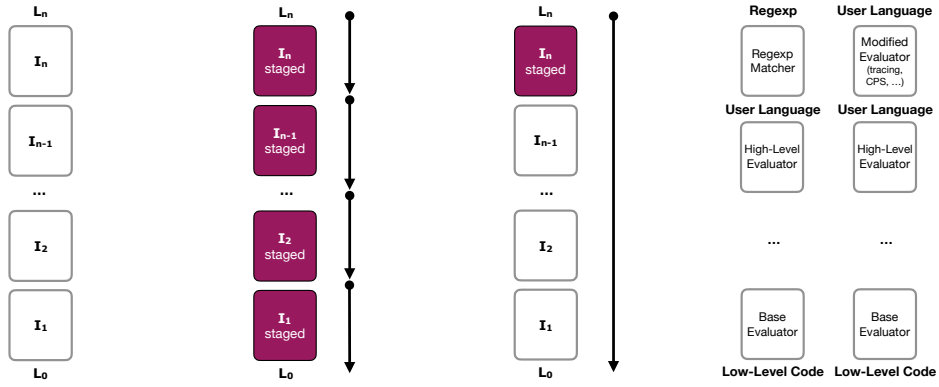
ACM Reference Format:
Nada Amin and Tiark Rompf. 2018. Collapsing Towers of Interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 33 (January 2018), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper is concerned with the challenge of collapsing towers of interpreters, i.e., sequences of multiple interpreters interpreting one another as input programs. As illustrated in Figure 1a, given a sequence of programming languages $L_0, \ldots, L_n$ and interpreters I_{i+1} for L_{i+1} written in L_i, the challenge is to derive a compiler from L_n to L_0. This compiler should be optimal in the sense that the translation removes all interpretive overhead, and the compiler should run in 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 tower of interpreters

Authors' addresses: Nada Amin, Computer Laboratory, University of Cambridge, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, first.last@cl.cam.ac.uk; Tiark Rompf, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN, 47907, USA, first@purdue.edu.

2475-1421/2018/1-ART33 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Tower L_n to L_0 (b) n-Pass Compiler (c) One-Pass Compiler (d) Examples

Fig. 1. Collapsing Towers of Interpreters

may be *conceptually infinite*, i.e., each interpreter can itself be interpreted, so that the number of meta levels can be arbitrarily large and dynamically adjusted.

Examples. As an example of collapsing a tower of interpreters, consider a base virtual machine executing an evaluator executing a regular expression matcher (illustrated in Figure 1d). 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 (illustrated in Figure 1d). 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. For instance, 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.

Proposed Solution. It is well known that *staging* an interpreter – making it generate code whenever it would normally interpret an expression – yields a compiler (review in Section 2). So as a first attempt illustrated in Figure 1b, 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 multi-pass compiler 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? We draw on a key insight from the domain of high-performance program generators for numeric libraries, namely the idea that by abstracting over staging decisions through an explicit notion of *stage polymorphism*, a single program generator can produce code that is specialized in many different ways [Ofenbeck et al. 2017]. Armed with this insight, the key idea of our approach is to abstract over compilation vs interpretation. We start with a multi-level language L_0, i.e., a language that has built-in staging operators, and express all other interpreters in a way

that makes them *stage polymorphic*, which means that they are able to act either as an interpreter or as a translator. Then, as illustrated in Figure 1c, we wire up the 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. As a result, only the staging commands that represent the top-level user program will lead to actual code generation commands. In essence, this approach only stages the final interpreter, but not the rest of the tower. As we will see, this approach can sustain collapsing arbitrary meta-levels of interpretation as well as compiling under user-modified semantics.

Contributions. 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. To the best of our knowledge, no previous work achieves compilation in a reflective tower with respect to user-modified semantics, and no previous work that we are aware of achieves collapsing even fixed towers of interpreters, reliably, into single-pass compilers.

The specific contributions of this paper are the following:

- We develop a multi-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 via stage polymorphism.
- We present a meta-circular evaluator 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 [Asai et al. 1996], 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 [2014] (Section 6).
- We present a range of examples in Purple / Black that make extensive use of reflection (Section 7). We implement Purple (Section 9) on top of Lightweight Modular Staging (LMS) and discuss how stage-polymorphic interpreters can be implemented 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. All our code is available from popl18.namin.net.

2 PRELIMINARIES

It is well known that interpreters and compilers are fundamentally linked through *specialization*, as formalized in the three Futamura projections [Futamura 1971, 1999]. First, specializing an interpreter to a given program yields a compiled version of that program, in the implementation language of the interpreter. 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* [Ershov 1978]. 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?

Futamura has not only clarified the relationship between formal descriptions of a programming language (i.e., an interpreter) and an actual compiler, but also proposed to realize this process through automatic program specializers or *partial evaluators*. The third Futamura projection in particular 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) [Jones et al. 1989]. 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 [Birkedal and Welinder 1994; Thiemann 1996]. 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 [Taha and Sheard 2000] and MetaOCaml [Calcagno et al. 2003; Kiselyov 2014], and of course by much earlier work in Lisp and Scheme [Bawden 1999].

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 b can range over integers [Glück and Jørgensen 1996; Thiemann 1996] or include variables β for polymorphism [Henglein and Mossin 1994].

Multi-stage languages in the line of MetaML [Taha and Sheard 2000] feature quasiquotation syntax, following similar facilities in Lisp-like languages:

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

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., Lightweight Modular Staging (LMS) [Rompf and Odersky 2012] in Scala. Multi-level languages differ also quite significantly in their semantics. MetaML and its descendants, for example, provide hygiene guarantees for bindings, but follow the Lisp tradition of interpreting 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.

Syntax

$$
\begin{array}{lcl}
e & ::= & x \mid \texttt{Lit}(n) \mid \texttt{Str}(s) \mid \texttt{Lam}(f,x,e) \mid \texttt{App}(e,e) \mid \texttt{Cons}(e,e) \mid \texttt{Let}(x,e,e) \mid \texttt{If}(e,e,e) \mid \\
 & & \oplus^1(e) \mid \oplus^2(e,e) \mid \texttt{Lift}(e) \mid \texttt{Run}(e,e) \mid g \\
g & ::= & \texttt{Code}(e) \mid \texttt{Reflect}(e) \mid \texttt{Lam}_c(f,x,e) \mid \texttt{Let}_c(x,e,e) \\
\oplus^1 & ::= & \texttt{IsNum} \mid \texttt{IsStr} \mid \texttt{IsCons} \mid \texttt{Car} \mid \texttt{Cdr} \\
\oplus^2 & ::= & \texttt{Plus} \mid \texttt{Minus} \mid \texttt{Times} \mid \texttt{Eq} \\
v & ::= & \texttt{Lit}(n) \mid \texttt{Str}(s) \mid \texttt{Lam}(f,x,e) \mid \texttt{Cons}(v,v) \mid \texttt{Code}(e)
\end{array}
$$

Contexts

$$
\begin{array}{lcl}
M & ::= & [] \mid B(M) \mid R(M) \qquad E ::= [] \mid B(E) \qquad P ::= [] \mid B(Q) \mid R(P) \qquad Q ::= B(Q) \mid R(P) \\
B(X) & ::= & \texttt{Cons}(X,e) \mid \texttt{Cons}(v,X) \mid \texttt{Let}(x,X,e) \mid \texttt{App}(X,e) \mid \texttt{App}(v,X) \mid \texttt{If}(X,e,e) \mid \\
 & & \oplus^1(X) \mid \oplus^2(X,e) \mid \oplus^2(v,X) \mid \texttt{Lift}(X) \mid \texttt{Run}(X,e) \mid \texttt{Reflect}(X) \\
R(X) & ::= & \texttt{Lift}(\texttt{Lam}_c(f,x,X)) \mid \texttt{If}(\texttt{Code}(e),X,e) \mid \texttt{If}(\texttt{Code}(e),v,X) \mid \texttt{Run}(v,X) \mid \texttt{Let}_c(x,e,X)
\end{array}
$$

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{App}(\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{App}(e_1,e_2))] \\
M[\texttt{If}(n \mid 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{If}(\texttt{Code}(e_0),\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{If}(e_0,e_1,e_2))] \\
M[\texttt{IsNum}(\texttt{Lit}(n))] & \longrightarrow & M[\texttt{Lit}(1)] \\
M[\texttt{IsNum}(v \mid v \neq \texttt{Code}(\_) \;\&\; v \neq \texttt{Lit}(\_))] & \longrightarrow & M[\texttt{Lit}(0)] \\
M[\texttt{IsNum}(\texttt{Code}(e))] & \longrightarrow & M[\texttt{Reflect}(\texttt{IsNum}(e))] \\
M[\texttt{Plus}(\texttt{Lit}(n_1),\texttt{Lit}(n_2))] & \longrightarrow & M[\texttt{Lit}(n_1+n_2)] \\
M[\texttt{Plus}(\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{Plus}(e_1,e_2))] \\
\textit{... other unary and binary operators elided ...} & & \\
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{Code}(\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{Code}(\texttt{Lift}(e)))] \\
M[\texttt{Run}(\texttt{Code}(e_1),\texttt{Code}(e_2))] & \longrightarrow & M[\texttt{Reflect}(\texttt{Code}(\texttt{Run}(e_1,e_2)))] \\
M[\texttt{Run}(v_1 \mid v_1 \neq \texttt{Code}(\_),\texttt{Code}(e_2))] & \longrightarrow & M[e_2] \\
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. 2. $\lambda_{\uparrow\downarrow}$ Small-Step Semantics

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 [Henglein and Mossin 1994], it supports binding-time or stage 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. Inspired by type-directed partial evaluation (TDPE) [Danvy 1996b], its key mechanism is a polymorphic `Lift` operator that turns a static, present-stage, *value* into a future-stage expression.

```
// Syntax -- Exp corresponds to e from small-step (Fig. 2), without internal-only forms (Reflect, Code, ...).
//        -- Instead of explicit names, we use de Bruijn levels Var(n), hence Lam(e) instead of Lam(f,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) | IsNum(a: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 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 IsNum(e1)    => evalms(env,e1) match {
    case Code(s1)     => reflectc(IsNum(s1))
    case Cst(n)       => Cst(1)
    case v            => Cst(0) }
  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 Lift(e)      => liftc(evalms(env,e))
  case Run(b,e)     => evalms(env,b) match {
    case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _            => evalmsg(env, reifyc({ stFresh = env.length; 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. 3. $\lambda_{\uparrow\downarrow}$ Big-Step Semantics as a Definitional Interpreter in Scala

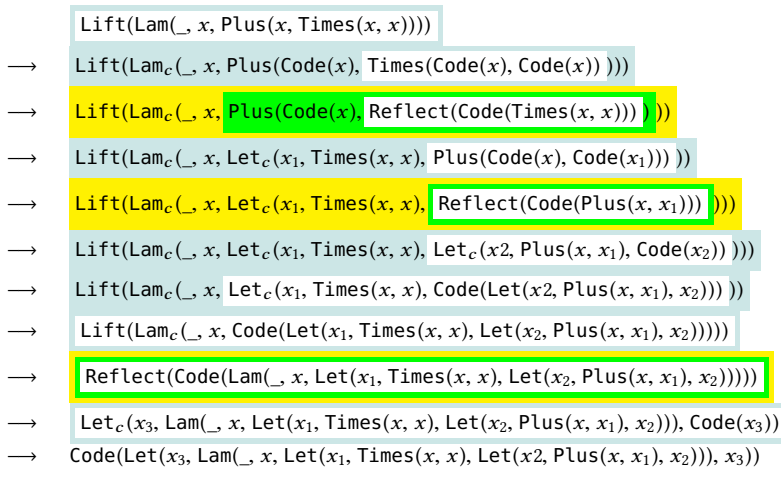


Fig. 4. Example of small-step derivation in $\lambda_{\uparrow\downarrow}$ highlighting the P, E and M contexts.

We present a small-step operational semantics in Figure 2 and a big-step operational semantics as a definitional interpreter written in Scala in Figure 3. We first designed the interpreter in Figure 3, and then devised the small-step rules to make all intermediate steps of the big-step evaluation explicit, introducing internal-only syntactic forms to represent the various pieces of the interpreter's state. The top-level entry point to the big-step evaluator is function `evalmsg`. We also write $e \Downarrow v$ for top-level evaluation in an empty environment, i.e., `evalmsg(Nil,e) = v`. We present the following claim without formal proof, but backed by experimental evidence:

Proposition 3.1 (Semantic Equivalence). *For any $\lambda_{\uparrow\downarrow}$ expression e, small-step value v, and equivalent big-step value v' we have $e \longrightarrow^* v$ if and only if $e \Downarrow v'$.*

The small-step version lends itself to formal reasoning, while the big-step version is more suitable for experimentation. We posit that a formal connection can be established through Danvy et al.'s semantic inter-derivation method [Ager et al. 2003; Danvy and Johannsen 2010; Danvy et al. 2012], via CPS conversion, defunctionalization [Reynolds 1972], and refocusing [Danvy and Nielsen 2004]. 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 [Felleisen et al. 2009].

The term syntax contains λ-calculus constructs, plus operators `Lift` and `Run`. In small-step, there are additional intermediary constructs for bookkeeping, such as `Reflect`, $\texttt{Lam}_c$ and $\texttt{Let}_c$ (noted under the syntax term g in Figure 2). 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-reference 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`. For non-recursive functions we use `_` in the place of identifier `f`.

The polymorphic `Lift` operator is inspired by a corresponding facility in normalization by evaluation (NBE) [Berger et al. 1998] or type-direction partial evaluation (TDPE) [Danvy 1996b]. 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/TDPE. 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 perform "let-insertion" [Bondorf 1990; Bondorf and Danvy 1991; Danvy 1996a; Danvy and Filinski 1990; Hatcliff and Danvy 1997; Lawall and Thiemann 1997; Thiemann and Dussart 1999] to maintain the relative evaluation order of expressions (see example in Figure 4). This is a standard practice in partial evaluators that deal with state and effects, and the result is also described as monadic normal form [Moggi 1991] or administrative normal form (ANF) [Flanagan et al. 1993].

In the big-step semantics, let-insertion (i.e., 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, i.e., made explicit, 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.

In the big-step implementation, we use a name-less de Bruijn level representation for simplicity. The variable `stFresh` holds the next available de Bruijn level. In the `Run` case, the statement `stFresh = env.length` aligns the de Bruijn levels of the present stage and the code being generated. The main entry point `evalmsg` delegates to `evalms` and also packages up and returns all generated code, if any.

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 $e = \texttt{Lam}(\_, x, \texttt{Plus}(x, \texttt{Times}(x, x)))$ in Figure 4. The big-step evaluator computes the equivalent result in a single call to `evalmsg(Nil,e)`.

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 a `Code` value as argument, hence matching on the argument would not allow us to decide whether to evaluate the expression in the `Code` value now or generate a `Run` expression for the future stage. Hence, `Run` takes an *additional* initial argument b, which solely exists for the purpose of matching. Thus, assuming e evaluates to $\texttt{Code}(e')$, $\texttt{Run}(\texttt{Lit}(0), e)$ will evaluate e' now, whereas $\texttt{Run}(\texttt{Lift}(\texttt{Lit}(0)), e)$ will generate a call to $\texttt{Run}(\texttt{Lit}(0), e')$ in the next stage.

3.1 A Lisp-Like Front-End

We implement a small Lisp reader that translates S-expressions 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.

[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* [Filinski 1994], but they have nothing to do with reflective language capabilities in the sense of Section 5 and 6.

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 unfoldings 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 6.

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* was introduced as a programming model for high-performance code generators with similar use cases in mind [Ofenbeck et al. 2017], and it enables us to achieve exactly this. The key insight here is that we can *abstract over*

```
(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. 5. Meta-circular stage-parametric evaluator for Pink.

`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 evaluator, shown in Figure 5, 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 $\lambda_{\uparrow\downarrow}$. The evaluator in Figure 5 is binding-time parametric (parameter `maybe-lift`), so it can act as an interpreter or 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 evaluator is to put a call to `maybe-lift` around all *user-level values* the evaluator creates: literal numbers (case `num?`), closures (case `'lambda`), and cons cells (case `'cons`).

4.1 Correctness and Optimality

In this section, we discuss correctness and optimality of specialization via Pink and $\lambda_{\uparrow\downarrow}$. We state these properties without formal proofs but backed by experimental evidence. We start by instantiating the stage-polymorphic Pink evaluator from Figure 5 as a normal interpreter, 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 '(lambda f n (if n (* n (f (- n 1))) 1)))
> ((eval fac-src) 4)   ;; => 24
```

Definition 4.1 (Source and Translation). For any Pink program `p`, let `p-src` be its quoted Pink source in S-expression form and let ⟦ `p` ⟧ be its translation to $\lambda_{\uparrow\downarrow}$ in ANF.

For correctness, we should be able to add levels of interpretation without changing the result of the program (albeit slowing it down), leading to the intuitive equivalence `eval` ≡ `(eval eval-src)`. We can verify that double and triple interpretation work in this way, given a quoted version of the interpreter source `eval-src`:

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

Proposition 4.2 (Correctness of Interpretation). *For any Pink program `p`, evaluating its source is observationally equivalent to the program itself:* ⟦ `(eval p-src)` ⟧ ≡ ⟦ `p` ⟧.

Correctness of (repeated) self-interpretation follows by considering `p = eval`.

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)))
```

Now 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.

Proposition 4.3 (Correctness of Compilation). *For any Pink program `p`, compiling and running its source is observationally equivalent to the program itself:* ⟦ `(run 0 (evalc p-src))` ⟧ ≡ ⟦ `p` ⟧.

However, our goal is to generate not only correct, but also efficient code. And in fact, we can show a much stronger property:

Proposition 4.4 (Optimality of Compilation). *For any Pink program `p`, compiling its source yields* exactly *the program itself (in ANF):* ⟦ `(evalc p-src)` ⟧ ⇓ ⟦ `p` ⟧.

In other words, `evalc` leaves no trace of any of the interpretive overhead that is present in the definition of `eval-poly`.

Taking this result a step further, we want to collapse levels of self-interpretation, even across towers. More formally, we want to show a notion of Jones-optimality [Glück 2002; Jones et al. 1993] for the interpreter `eval`: for each program `p`, running the compiled program `(evalc p-src)` should

be at least as efficient as evaluating the program p directly. The following definition is adapted from Section 6.4 of the book by Jones et al. [1993]:

Definition 4.5 (Jones Optimality of Specialization). A partial evaluator M is Jones-optimal for a self-interpreter E with respect to a time measure t provided $t(p'\ d) \leq t(p\ d)$ for any two-argument programs p, p' such that $p' = M\ E\ p$ and any program input d.

In our setting, there is no explicit partial evaluator, but we can think of M as the programmer-controlled specialization process and identify `evalc = M eval`. Thus, Proposition 4.4 directly implies Jones-optimality for any time measure (since p' is identical to p).

We can further verify collapsing with an additional level of interpretation:

```
> ((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 theoretical justification is given by Proposition 4.4 and Proposition 4.2, which gives rise to the equivalences `(eval evalc-src)` ≡ `evalc` and `(eval eval-src)` ≡ `eval`.

Proposition 4.6 (Multi-Level Jones Optimality). *For any Pink program* `p`*, arbitrarily many levels of self-interpretation collapse: for any natural number* `n`*,* ⟦ `((eval`^{n} `evalc-src) p-src)` ⟧ ⇓ ⟦ `p` ⟧*, where* `eval`^{n} *is defined recursively as* `eval`^{1} `= eval` *and* `eval`$^{n+1}$ `= (eval`^{n} `eval-src)`*.*

The key pattern here is that all the base evaluators, i.e., `(eval eval-src)` are instantiated in actual interpretation mode, but the final evaluator operates as a compiler. Thus, the base evaluators 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 for domain-specific languages (DSLs). Let us exercise this pattern with a 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 6 shows a string matcher written directly in Pink. This version is adapted from Kernighan and Pike [2007], and covers more functionality than the one shown in Section 3.1. In particular, the pattern syntax supports wildcard `_` and repeat `*` patterns, but it does not support nesting of wildcards. Thus, in the pattern `a**`, the second `*` is treated as a literal character. The pattern is matched against the beginning of the string, so the pattern `a**b` matches `*b`, `a*b`, `aa*b` and `a*bc`, but neither `b` nor `a**b`.

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.

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 >
```

```
(lambda _ maybe-lift
(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 (cadr r)) (m s)
    (if (eq?  '* (cadr r)) (((star_loop (match_here (cddr 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. 6. Binding-time polymorphic string matcher in Pink.

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

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

```
> (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. 7. 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.

The $\lambda_{\uparrow\downarrow}$ code for `fac`, with extra log calls as transformed by tracing variables named `n` is shown in Figure 7. 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 6. 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 evaluator 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 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 [Danvy and Filinski 1990; Jones 2004]. 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 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 7. Each lambda takes an additional curried argument for the continuation. All function calls are in tail position, with inner lambdas passed as continuation arguments.

We also have a choice to residualize or duplicate the continuation for conditionals. Duplicating the continuation is sometimes desirable, but may lead to code explosion for nested conditional expressions. Residualizing the continuation as common join point, by contrast, will guarantee linear space behavior. Since continuations take a `Code` argument and return a `Code` value when in compilation mode, we can lift a continuation into a residual function at any time [Danvy 2003].

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.

Proposition 4.7 (Jones Optimality Modulo Projection). *Given a modified self-interpreter* `t-eval` *that implements the observable effects of program transformation* T*, the specialized interpreter* `t-evalc` *materializes the transformation* T*: for any Pink program* `p`*, if* $[\![$ `(t-eval p-src)` $]\!] \equiv T\ [\![$ `p` $]\!]$ *then* $[\![$ `(t-evalc p-src)` $]\!] \Downarrow T\ [\![$ `p` $]\!]$.

This property holds for both examples shown above.

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 [Asai et al. 1996]. Invoking `(EM e)` will execute the expression `e` *as if it were part of the interpreter code*. Here is an example:

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

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 evaluator:

```
(lambda _ maybe-lift (lambda tie eval (lambda _ exp (lambda _ env ...
  (if (sym? (car exp)) ... (if (eq? 'EM (car exp)) (let e (cadr 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. Inside `EM`, `(env 'e)` retrieves the value of variable `e`. 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 (cadr 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. Here, `tie` is the recursive self-reference for the function that implements open recursion, from the interpreter signature as in Section 5.1.1. To launch a modified interpreter, we invoke `tie` with a function `ev` that overrides the desired cases and otherwise delegates to `eval`.

```
> (eval '((EM (((lambda ev exp (lambda _ env (if (if (sym? exp) (eq? 'n exp) 0) (log 0 ((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` [1990] 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))))))
> (cps-eval '(+ 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. Note that defining `call/cc` through reification is already discussed in classic works on reflective towers, notably Brown [Friedman and Wand 1984; Wand and Friedman 1986] and Blond [Danvy and Malmkjær 1988].

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 [Asai 2014], 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 [Asai 2014], 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 [Asai 2014], 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)) (run 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 `lift-ref` enables cross-stage persistence by injecting pointers to the *existing* environment values into the generated code. We assume that a suitable implementation is added to $\lambda_{\uparrow\downarrow}$.

[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.

6 PURPLE: REFLECTION À LA BLACK

We now consider a more powerful reflective tower, which has potentially infinitely many levels of meta-interpretation. The tower still 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 levels are spawned lazily, as they are accessed and modified by constructs like `EM`.

Our language, dubbed *Purple*, is heavily inspired by Asai's reflective language Black [Asai et al. 1996]. 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, Lightweight Modular Staging (LMS) [Rompf and Odersky 2012]. While we could have grown Purple from Pink, we found it more interesting to investigate if we can apply the underlying ideas 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 [Asai et al. 1996], 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 [Asai 2014] used MetaOCaml [Calcagno et al. 2003; Kiselyov 2014] 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 [Asai 2014]: 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 it is possible to compile a user program under modified, possibly also compiled, semantics, building on the techniques presented earlier in the context of Pink.

6.1 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 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 object 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 in Figure 9.

```
{(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, bypassing any explicit concepts (such as environments) of 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. Section 7 presents additional examples in Purple / Black.

6.2 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 stage-polymorphic, i.e., polymorphic over whether they interpret or compile, and also generate code that is stage-polymorphic.

While $\lambda_{\uparrow\downarrow}$ has dynamic stage polymorphism more akin to online partial evaluation, LMS has type-driven stages, and so here we explore static stage polymorphism (Section 8) more akin to offline partial evaluation. We further rely on LMS optimizations to heuristically achieve as much collapse as in Pink. Compared to Pink, Purple does not expose any manual staging or lifting constructs: only `clambda`, which marks a `lambda` for compilation.

6.3 Tower Structure

In a “classic” tower of interpreters, as in languages 3-Lisp [Smith 1984], Brown [Wand and Friedman 1986], Blond [Danvy and Malmkjær 1988], or Black [Asai 2014; Asai et al. 1996], each level has its own environment and its own continuation. Conceptually, the semantics of level n is given by the

interpreter at level $n + 1$, which takes an expression, an environment and a continuation, together with a stream of all the environments and continuations of the levels above. So level 0 (the user level) is interpreted by level 1 (the meta level), which is interpreted by level 2 (the meta meta level), and so on. Note that such a tower is depicted with the user level at the bottom, and then increasing meta levels, while confusingly fixed towers are the other way around, with the most meta level at the bottom and the user level at the top.

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. This departure from "classic" towers of interpreters is a meta-difference: a difference in the meta-execution of the tower itself rather than the observed meta-interpreter. Intuitively, the entire tower has only a single thread of execution and focus, while "classic" towers have independent threads of executions for each level.

Thus, our tower has a rigid level structure, better suited for collapsing. Each level has its own global environment, containing bindings for primitives (such as `null?`, `+`, `display`) and, except at the user level, for interpreter functions (such as `base-eval`, `eval-var`, `base-apply`). Each interpreter function takes as arguments an expression, an environment and a continuation. These arguments usually come from the object 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.

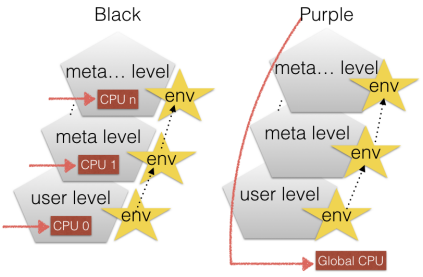


Fig. 8. "Classic" vs. Rigid Reflective Towers

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 when a function is defined (and possibly compiled) and the meta-state when it is applied. 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. In our case, we avoid all these thorny issues by using a rigid structure and a lexical discipline for both environments and meta-environments. For a strange example, calling the continuation `_k` below pushes another user level in Black:

```
(begin (define where_am_i 'user) (EM (define where_am_i 'meta)) (define _ 0)
(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))))))))
> (+ _ 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!)
```

Figure 8 summarizes the differences in structure between Black and Purple: (1) each level has a conceptual CPU in Black, while there is a single global CPU in Purple; (2) in Black, the environments (represented by stars in Figure 8) are detachable from the tower structure, while they are rigidly attached in Purple. Even though the tower structure in Purple is rigid, the semantics is not, since interpreter functions can be redefined using `set!`.

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 Introspection for TABA (There and Back Again)

There is a neat recursion pattern [Danvy and Goldberg 2005] 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 recursion pattern, it is helpful to visualize the pending operations as the recursive calls unfold. 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 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) (pending '()))
  (map (lambda (fn) (add-app-hook! fn (eval-taba-call (lambda (ans-args ans)
    (set! pending (cons (list fn ans-args ans) pending))) eval-application))) fns)
  (base-eval exp env (lambda (ans) (set! eval-application original-eval-application)
    (cont (list ans pending))))))))
(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 remember via pending operations the successive elements 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 to the monitoring semantics.

This technique for introspecting TABA calls works well for all direct-style examples (with no exceptional control flow), as the arguments to calls are directly observable. For higher-order solutions, including CPS, the introspection is more opaque due to closures as arguments.

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 the object level 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 simple string matcher example from Section 3.1 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 shown 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 of Section 6.1.

```
> ((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 [Rompf and Odersky 2012]. Before we dive further into the implementation of Purple, we first discuss how the ideas prototyped in the context of $\lambda_{\uparrow\downarrow}$ scale to LMS, especially the key ingredient of stage polymorphism.

Having seen in Section 3.1 how $\lambda_{\uparrow\downarrow}$ can serve as a model for untyped front-ends, it is instructive to look at typed models of multi-stage evaluation. 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 it provides overloaded methods on such `Rep[T]` values, e.g., `+` for `Rep[Int]`. Thus, Scala's local type inference and overloading resolution performs a form of local binding-time analysis.

Previous work has clarified how to understand multi-stage evaluation in LMS in terms of an explicit multi-level interpreter [Rompf 2016]. We summarize the main ideas below. Essentially, 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 [Carette et al. 2009] 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 3, we observe 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 3, 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(f,x))
```

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 [Click and Cooper 1995; Rompf and Odersky 2012; Rompf et al. 2013]. 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 `ab`:

```
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? LMS relies on the normal Scala type system without specific support for polymorphic binding time abstraction and application operators as in [Henglein and Mossin 1994]. 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. Based on previous work [Ofenbeck et al. 2013], 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? We follow previous work [Ofenbeck et al. 2017] in using type classes [Wadler and Blott 1989] to good effect. We define a type class interface and corresponding instances of `Ops[NoRep]` and `Ops[Rep]` that delegate to the appropriate base methods on plain types (`NoRep[T] = T`) or `Rep` types respectively. Figure 9 shows the actual type class interface used in Purple.

Now we can go ahead and define the stage-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 preamble `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)
```

```
trait Ops[R[_]] {
  implicit def _lift(v: Value): R[Value]          def _liftb(b: Boolean): R[Boolean]
  def inRep: 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 _car(p: R[Value]): R[Value]                 def _cell_new(v: R[Value], memo: String): R[Value]
  def _cdr(p: R[Value]): R[Value]                 def _cell_set(c: R[Value], v: R[Value]): R[Value]
  def _cons(car:R[Value],cdr:R[Value]): R[Value]  def _cell_read(c: R[Value]): R[Value] }
type NoRep[T] = T                                        // identity type constructor
implicit object PlainOps extends Ops[NoRep] { /* ... */ } // for normal interpretation
implicit object RepOps extends Ops[Rep] { /* ... */ }     // for compilation via LMS
```

Fig. 9. Type class interface defining target language, and instances for interpretation and compilation.

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]` of Figure 9.

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. `Evalfun`s represent a reference to a built-in interpreter function, or to a compiled user function (`clambda`).

9.1 Initializing the Tower

Below is the code for building the tower level structure. Instead of meta-continuations, Purple uses only rigid meta-environments as discussed in Section 6.3. 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 meta 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.

9.2 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. 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`. Even though the expression, environment and continuation of the object 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 (defined in Figure 9), 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 meta 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`.

Execute-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 object 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 stage-polymorphic function */
    _fun(new Fun[R] { def fun[RF[_]:Ops] ={ kv => eval_body[RF](kv) }}) }
  apply_cont[R](cont, f) }
```

By sketching the implementation of Purple, we have given a constructive answer to the challenge of collapsing reflective towers. Following earlier work on Black, we wire the tower structure so that the meta levels are potentially infinite, and the base evaluation defined in the host language leaves all interpreter functions up for modification. We treat host-defined functions and user-defined compiled functions uniformly, and to support collapsing, these functions are stage-polymorphic, requiring generating code that is stage-polymorphic. Stage polymorphism is thus a key ingredient for collapsing reflective towers.

10 BENCHMARKS

We show some micro benchmarks in Figure 10, computing factorial from 0 to 9, through evaluation and compilation, as well as with and without tracing, for the original Black (on Scheme and on MetaOCaml), and our own Pink and Purple. In all cases, tracing is achieved by incrementing a counter cell. In Pink, compilation is achieved by using a lifted evaluator (e.g. `evalc`). In Purple and Black (on MetaOCaml), compilation is achieved by using `clambda`. In Black (on MetaOCaml), compilation is only possible with respect to the default semantics (c) but not with respect to modified semantics (c_t); in Black (on Scheme), no compilation is possible.

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. The benchmarks for Black (on Scheme) and for Black (on MetaOCaml) are broadly comparable with the benchmarks for Pink and Purple, respectively, and suggest that collapsing would be beneficial.

For example, note that e_t for Black (on MetaOCaml) lies between e_t and c_t for Purple, and that the ratio $\frac{e_t}{c_t}$ for Purple shows the benefits of collapsing with speed ups in orders of magnitude.

	Black (on Scheme) [Asai et al. 1996]				Black (on MetaOCaml) [Asai 2014]					
n	e	e_t	$\frac{e(n)}{e(n-1)}$	$\frac{e_t}{e}$	e	c	e_t	$\frac{e(n)}{e(n-1)}$	$\frac{e}{c}$	$\frac{e_t}{e}$
0	2380	15070	1.00	6.33	20647	17796	34580	1.00	1.16	1.67
1	3490	22810	1.47	6.54	31162	19376	54266	1.51	1.61	1.74
2	4740	31710	1.36	6.69	42782	20553	73496	1.37	2.08	1.72
3	5840	39240	1.23	6.72	53878	21932	92338	1.26	2.46	1.71
4	7130	47870	1.22	6.71	63113	23175	110884	1.17	2.72	1.76
5	8470	55300	1.19	6.53	75404	24336	130029	1.19	3.10	1.72
6	9660	65740	1.14	6.81	85664	25535	152028	1.14	3.35	1.77
7	11330	75690	1.17	6.68	97299	26653	171987	1.14	3.65	1.77
8	12060	80820	1.06	6.70	107850	28235	191590	1.11	3.82	1.78
9	13690	89690	1.14	6.55	120602	29740	214206	1.12	4.06	1.78

	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. 10. 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 [Jones et al. 1993] 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 [Grant et al. 2000], an annotation-directed specializer for C, JSpec/Tempo [Schultz et al. 2003], the JSC Java Supercompiler [Klimov 2009], and Civet [Shali and Cook 2011].

Partial evaluation has addressed higher-order languages with state using similar let-insertion techniques as discussed here [Bondorf 1990; Hatcliff and Danvy 1997; Lawall and Thiemann 1997; Thiemann and Dussart 1999]. Further work has studied partially static structures [Mogensen 1988] and partially static operations [Thiemann 2013], and compilation based on combinations of partial evaluation, staging and abstract interpretation [Consel and Khoo 1993; Kiselyov et al. 2004; Sperber and Thiemann 1996]. Two-level languages are frequently used as a basis for describing binding-time annotated programs [Jones et al. 1993; Nielson and Nielson 1996].

Multi-level binding-time analysis extends binding-time analysis (BTA) from two stages to more levels [Glück and Jørgensen 1996]. Polymorphic binding-time analysis extends binding-time analysis so that some expressions can be assigned multiple stages [Henglein and Mossin 1994]. 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.

Type-directed partial evaluation [Danvy 1996a,b, 1998a,b; Filinski 1999] is a partial evaluation technique that leverages meta-language execution to perform static reductions. The initially proposed version corresponds to normalization by evaluation (NBE) [Berger et al. 1998] and yields residual code in $\beta\eta$-normal form, but later works have also considered variants that can residualize selected redexes, since full normalization is often too aggressive.

Our work builds heavily on ideas from TDPE. In particular, the $\lambda_{\uparrow\downarrow}$ `lift` operator corresponds exactly to the two-level η-expansion in TDPE. Unlike the original formulation of TDPE [Danvy 1996b] but somewhat similar to later work [Filinski 1999], `lift` is used explicitly by the programmer. But $\lambda_{\uparrow\downarrow}$ as well as Pink do not use (an explicit representation of) types to guide the transformation, and the residual expressions are not necessarily $\beta\eta$-normalized. The $\lambda_{\uparrow\downarrow}$ base interpreter implements eager let-insertion without requiring a notion of effect types as in corresponding TDPE approaches [Danvy 1996a], and the difficulties of dealing with polymorphic types noted in the original TDPE paper [Danvy 1996b] also do not seem to apply.

The "writing cogen by hand approach" was adopted by Birkedal and Welinder [1994] to solve the typing problem in typed self-applicable partial evaluators, and was developed further by Thiemann [1996]. Glück and Jørgensen [1995, 1998] showed how optimizing specializers can be derived by layering and specializing interpreters. Jones [2004] gives an overview of program transformation via interpreter specialization.

The book by Jones et al. [1993] briefly discusses hierarchies of languages and their repeated specialization. Glück and Klimov [1999] studied reduction of language hierarchies by program composition and specialization. Foundational works on the CPS hierarchy [Danvy and Filinski 1989, 1990] suggest an early example of residualizing layered interpreters, demonstrating that a program can be CPS transformed either in multiple passes, or all at once. Danvy's doctoral dissertation [2006] discusses how one-pass CPS transformation inspired the development of TDPE, in particular the use of two-level η-expansion for binding-time separation, and shows how the transformed type of the program guides the residualization in this case.

Multi-stage programming. Multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [2000] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [Calcagno et al. 2003; Kiselyov 2014] implements a classic staging system based on quasi-quotation. The semantics of multi-stage programming are still a subject of ongoing study [Berger et al. 2017; Ge and Garcia 2017].

Lightweight Modular Staging (LMS) [Rompf and Odersky 2010, 2012] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code [Rompf 2012]. LMS draws inspiration from earlier work such as TaskGraph [Beckmann et al. 2003], a C++ framework for program generation and optimization. LMS has been used in a variety of applications, ranging from web programming [Kossakowski et al. 2012] over domain-specific languages for machine learning [Rompf et al. 2011; Sujeeth et al. 2011] to database engines [Rompf and Amin 2015] and distributed systems [Ackermann et al. 2012].

Reflective Towers. Smith [1982, 1984] introduced reflective towers in seminal papers on 3-Lisp. The motivation stems from enabling processes to inspect on their computation arbitrarily. Friedman and Wand [1984]; Wand and Friedman [1986] distill the essence of reflection in Brown, explaining reflection and reification in a self-contained semantics, which does not re-allude to reflection. Later, Jefferson and Friedman [1996] also give a simplified account for a finite tower, $\mathcal{I}_R$, and at the same time, Sobel and Friedman [1996] also give an account of reflection without towers. Danvy and Malmkjær [1988] present a denotational semantics of Blond. Their account justifies the use of meta-continuations for a compositional semantics. As discussed earlier, our Purple reflective tower is inspired chiefly by Black [Asai 2014; Asai et al. 1996].

A line of recent work considers self-representation and self-interpretation of typed languages such as F_ω [Brown and Palsberg 2016, 2017; Rendel et al. 2009].

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 [Masuhara et al. 2003; Tanter 2010].

Program Generators. A number of high-performance program generators have been built, for example ATLAS [Whaley et al. 2001] (linear algebra), FFTW [Frigo 1999] (discrete Fourier transform), and Spiral [Püschel et al. 2004] (general linear transformations). Other systems include PetaBricks [Ansel et al. 2009], and CVXgen [Hanger et al. 2011]. Generating a variety of different code shapes and abstracting over choices such as fixed-size vs. variable-size inputs is a recurring problem in building high-performance code generators. Stage polymorphism or "generic programming in time" was recently discovered as a programming model that covers many of the important situations [Ofenbeck et al. 2017, 2013].

12 CONCLUSIONS

We have shown how to collapse towers of interpreters using a stage-polymorphic multi-level λ-calculus $\lambda_{\uparrow\downarrow}$. 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 [Zakai 2011] translates LLVM code to JavaScript. Similarly, Java VMs [Vilk and Berger 2014] and even entire x86 processor emulators [Hemmer 2017] that are able to boot Linux [Bellard 2017] 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 use cases 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 [Byrd et al. 2017] 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 [Horn and Might 2011], which has recently been extended to abstract definitional interpreters [Darais et al. 2017]. 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 the lowest level, which needs to support stage polymorphism. Last but not least, it is important to note that the present work is based on interpreters derived from variations of the λ-calculus, and thus leaves a gap towards collapsing heterogeneous towers of truly independent languages. This gap is especially prominent in a setting where a language level does not follow the usual functional or imperative paradigm, e.g., if a logic programming language or a probabilistic programming language is part of the tower. Thus, we hope that our work spurs further activity in implementing stage polymorphic virtual machines and collapsing towers of interpreters in the wild.

ACKNOWLEDGMENTS

We thank Kenichi Asai, Oliver Bračevac, Matt Brown, William E. Byrd, Olivier Danvy, Robert Glück, Sylvia Grewe, Grzegorz Kossakowski, Stefan Marr, Ulrik P. Schultz, Éric Tanter, as well as the anonymous reviewers for feedback on this work. Parts of this research were supported by ERC grant 321217, NSF awards 1553471 and 1564207, and DOE award DE-SC0018050.

REFERENCES

Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An Embedded DSL for High Performance Big Data Processing *(BigData)*.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *PPDP*.

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *PLDI*.

Kenichi Asai. 2014. Compiling a Reflective Language Using MetaOCaml. In *GPCE*. code from personal correspondence.

Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. 1996. Duplication and Partial Evaluation: For a Better Understanding of Reflective Languages. *Lisp and Symbolic Computation - Special issue on computational reflection*, 203–241. code at github.com/readevalprintlove/black.

Alan Bawden. 1999. Quasiquotation in Lisp. In *PEPM*.

Olav Beckmann, Alastair Houghton, Michael R. Mellor, and Paul H. J. Kelly. 2003. Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation. In *Domain-Specific Program Generation*.

Fabrice Bellard. 2011–2017. JSLinux. bellard.org/jslinux.

Martin Berger, Laurence Tratt, and Christian Urban. 2017. Modelling Homogeneous Generative Meta-Programming. In *ECOOP*.

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalization by Evaluation. In *Prospects for Hardware Foundations: ESPRIT Working Group 8533 NADA — New Hardware Design Methods Survey Chapters*, Bernhard Möller and John V. Tucker (Eds.). 117–137.

Lars Birkedal and Morten Welinder. 1994. Hand-writing program generator generators. In *PLILP*.

Anders Bondorf. 1990. *Self-applicable partial evaluation.* Ph.D. Dissertation. DIKU, Department of Computer Science, University of Copenhagen.

Anders Bondorf and Olivier Danvy. 1991. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming* 16, 2, 151–195.

Matt Brown and Jens Palsberg. 2016. Breaking through the normalization barrier: a self-interpreter for f-omega. In *POPL*.

Matt Brown and Jens Palsberg. 2017. Typed self-evaluation via intensional type functions. In *POPL*.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). In *ICFP*.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *GPCE*.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *JFP* 19, 5, 509–543.

Cliff Click and Keith D. Cooper. 1995. Combining analyses, combining optimizations. *TOPLAS* 17, 181–196. Issue 2.

Charles Consel and Siau-Cheng Khoo. 1993. Parameterized Partial Evaluation. *TOPLAS* 15, 3, 463–493.

Olivier Danvy. 1996a. Pragmatics of type-directed partial evaluation. In *Partial Evaluation: Dagstuhl, Selected Papers*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.).

Olivier Danvy. 1996b. Type-directed Partial Evaluation. In *POPL*.

Olivier Danvy. 1998a. Online Type-Directed Partial Evaluation. In *Fuji International Symposium on Functional and Logic Programming*.

Olivier Danvy. 1998b. Type-Directed Partial Evaluation. In *DIKU 1998 International Summer School*, John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann (Eds.).

Olivier Danvy. 2003. A New One-pass Transformation into Monadic Normal Form. In *Compiler Construction*.

Olivier Danvy. 2006. *An Analytical Approach to Programs as Data Objects.* DSc thesis. Department of Computer Science, Aarhus University, Aarhus, Denmark. ebooks.au.dk/index.php/aul/catalog/book/214.

Olivier Danvy and Andrzej Filinski. 1989. *A functional abstraction of typed contexts.* Technical Report. DIKU, University of Copenhagen.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Lisp and Functional Programming*.

Olivier Danvy and Mayer Goldberg. 2005. There and Back Again. *Fundam. Inform.* 66, 4, 397–413.

Olivier Danvy and Jacob Johannsen. 2010. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.* 76, 5, 302–323.

Olivier Danvy and Karoline Malmkjær. 1988. Intensions and Extensions in a Reflective Tower. In *Lisp and Functional Programming*.

Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. 2012. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theor. Comput. Sci.* 435, 21–42.

Olivier Danvy and Lasse R. Nielsen. 2004. *Refocusing in Reduction Semantics.* Research Report BRICS RS-04-26. Department of Computer Science, Aarhus University, Aarhus, Denmark. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). In *ICFP*.

Andrei P. Ershov. 1978. On the essence of compilation. *Formal Description of Programming Concepts*, 391–420.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press.

Andrzej Filinski. 1994. Representing Monads. In *POPL*.

Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *PPDP (Lecture Notes in Computer Science)*, Vol. 1702. Springer.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*.

Daniel P. Friedman and Mitchell Wand. 1984. Reification: Reflection without Metaphysics. In *Lisp and Functional Programming*.

Matteo Frigo. 1999. A Fast Fourier Transform Compiler. In *PLDI*.

Yoshihiko Futamura. 1971. 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.

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation*, 377–380.

Rui Ge and Ronald Garcia. 2017. Refining Semantics for Multi-stage Programming. In *GPCE*.

Robert Glück. 2002. Jones Optimality, Binding-time Improvements, and the Strength of Program Specializers. In *PEPM*.

Robert Glück and Jesper Jørgensen. 1995. Efficient Multi-level Generating Extensions for Program Specialization. In *PLILP*.

Robert Glück and Jesper Jørgensen. 1996. Fast binding-time analysis for multi-level specialization. In *Ershov Memorial Conference, PSI*.

Robert Glück and Jesper Jørgensen. 1998. Multi-Level Specialization (Extended Abstract). In *Partial Evaluation*.

Robert Glück and Andrei V. Klimov. 1999. Reduction of language hierarchies by metacomputation. In *The Evolution of Complexity*.

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. 2000. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.* 248, 1-2, 147–199.

Martin Hanger, Tor Arne Johansen, Geir Kare Mykland, and Aage Skullestad. 2011. Dynamic model predictive control allocation using CVXGEN. In *ICCA*.

John Hatcliff and Olivier Danvy. 1997. A Computational Formalization for Partial Evaluation. *Mathematical Structures in Computer Science* 7, 5, 507–541.

Fabian Hemmer. 2014–2017. x86 virtualization in JavaScript, running in your browser and NodeJS. copy.sh/v86.

Fritz Henglein and Christian Mossin. 1994. Polymorphic Binding-Time Analysis. In *ESOP*.

David Van Horn and Matthew Might. 2011. Abstracting abstract machines: a systematic approach to higher-order program analysis. *CACM* 54, 9, 101–109.

Stanley Jefferson and Daniel P. Friedman. 1996. A Simple Reflective Interpreter. *Lisp and Symbolic Computation* 9, 2-3, 181–202.

Neil D. Jones. 2004. Transformation by interpreter specialisation. *Sci. Comput. Program.* 52, 307–339.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA. www.itu.dk/people/sestoft/pebook.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1989. MIX: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation* 2, 1, 9–50.

Brian Kernighan and Rob Pike. 2007. A Regular Expression Matcher. In *Beautiful Code*, Greg Wilson and Andy Oram (Eds.). O'Reilly, Chapter 1. www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *FLOPS*.

Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. 2004. A methodology for generating verified combinatorial circuits. In *EMSOFT*.

Andrei V. Klimov. 2009. A Java Supercompiler and Its Application to Verification of Cache-Coherence Protocols. In *Ershov Memorial Conference, PSI*.

Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. 2012. JavaScript as an Embedded DSL. In *ECOOP*.

Julia L. Lawall and Peter Thiemann. 1997. Sound Specialization in the Presence of Computational Effects. In *TACS*.

Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. 2003. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Compiler Construction*.

Torben Æ. Mogensen. 1988. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation: IFIP TC2 Workshop*, Dines Bjørner, Andrei P. Ershov, and Neil D. Jones (Eds.).

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1, 55–92.

Flemming Nielson and Hanne Riis Nielson. 1996. Multi-Level Lambda-Calculi: An Algebraic Description. In *Partial Evaluation: Dagstuhl, Selected Papers*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.).

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for Generic Programming in Space and Time. In *GPCE*.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *GPCE*.

Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. 2004. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms. *IJHPCA* 18, 1, 21–45.

Tillmann Rendel, Klaus Ostermann, and Christian Hofer. 2009. Typed self-representation. In *PLDI*.

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference*.

Tiark Rompf. 2012. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. Ph.D. Dissertation. EPFL IC, Ecole Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences.

Tiark Rompf. 2016. The Essence of Multi-Stage Evaluation in LMS. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (WadlerFest)*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.).

Tiark Rompf and Nada Amin. 2015. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In *ICFP*.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*.

Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *CACM* 55, 6, 121–130.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-Level Programs. In *POPL*.

Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-Blocks for Performance Oriented DSLs, In DSL. *Electronic Proceedings in Theoretical Computer Science*.

Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *TOPLAS* 25, 4, 452–499.

Amin Shali and William R. Cook. 2011. Hybrid partial evaluation. In *OOPSLA*.

Brian C. Smith. 1982. *Reflection and Semantics in a Procedural Language*. Ph.D. Dissertation. MIT EECS, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

Brian C. Smith. 1984. Reflection and Semantics in Lisp. In *POPL*.

Jonathan M Sobel and Daniel P Friedman. 1996. An introduction to reflection-oriented programming. In *Proceedings of reflection*.

Michael Sperber and Peter Thiemann. 1996. Realistic Compilation by Partial Evaluation. In *PLDI*.

Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2, 211–242.

Éric Tanter. 2010. Execution Levels for Aspect-oriented Programming. In *AOSD*.

Peter Thiemann. 1996. Cogen in Six Lines. In *ICFP*.

Peter Thiemann. 2013. Partially static operations. In *PEPM*.

Peter Thiemann and Dirk Dussart. 1999. *Partial evaluation for higher-order languages with state*. Technical Report. www.informatik.uni-freiburg.de/ thiemann/papers/mlpe.ps.gz.

John Vilk and Emery D. Berger. 2014. Doppio: Breaking the Browser Language Barrier. In *PLDI*.

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL*.

Mitchell Wand and Daniel P. Friedman. 1986. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Lisp and Functional Programming*.

R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1-2, 3–35.
Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA*.