Collapsing Towers of Interpreters

Nada Amin*  Tiark Rompf†
*EPFL: (first.last)@epfl.ch
†Purdue University: (first)@purdue.edu

Abstract
Here is a challenge: given a tower of interpreters, i.e. a sequence of interpreters interpreting each other, collapse this tower into a one-pass compiler that removes all interpretive overhead. 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. In this paper, we lay the foundations based on a multi-level lambda calculus that features staging constructs and stage polymorphism: based on runtime parameters, the interpreter can either act as a normal interpreter or generate code, which turns it into a compiler, according to the Futamura projections. We identify stage polymorphism as the key mechanism to make such interpreters compose in a collapsible way.

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

1. Introduction
This paper is concerned with the following challenge: given a sequence of programming languages $L_0, \ldots, L_n$, and interpreters for $L_{i+1}$ written in $L_i$, derive a compiler from $L_n$ to $L_0$. This compiler should be one-pass, and it should be optimal in the sense that the translation removes all interpretive overhead of the intermediate languages. To make matters even more interesting, we also consider the case where some or all interpreters may be reflective, i.e. can be inspected and modified at runtime.

This challenge might seem artificial at first glance, but we believe that collapsing such towers of interpreters is of great 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 translates LLVM code to JavaScript. Similarly, Java VMs and even entire x86 processor emulators that are able to boot Linux have been written in JavaScript. It would be great if we could run all such artifacts at full speed, e.g. a Python application executed by an x86 runtime, emulated in a JavaScript VM. Naturally, this requires not only collapsing of static calls, but also adapting to a dynamically changing environment. (2) It can be desirable to execute code under modified semantics. Key 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 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, which has recently been extended to abstract definitional interpreters. For these indirect approaches to be effective, it is important to remove intermediate interpretive abstractions which would otherwise cloud the view of the analysis.

While we do not claim to solve the posed challenge for all possible combinations of languages, this paper makes important steps towards this goal: we collapse towers of interpreters based on variations of the lambda calculus as $L_0$.

Our approach is as follows. It is well known that staging an interpreter yields a compiler. So as a first attempt, we might stage all intermediate interpreters. However, this will produce a chain of translators, and not a one-pass compiler. The key insight is to start with a multi-level language $L_0$, i.e. a language that has built-in staging operators, and to express all other interpreters in a way that makes them stage polymorphic, i.e. able to act either as an interpreter or as a translator. Then, we wire up the interpretive tower so that the staging commands for $L_n$ are directly interpreted in terms of the staging commands of $L_0$. All intermediate interpreters $L_1,\ldots,L_{n-1}$ act in a kind of pass-through mode, handing down staging commands from $L_n$, but not executing any staging commands of their own.

Specifically, this paper makes the following contributions:

• We develop a two-level kernel language $\lambda_{12}$ that supports staging through a polymorphic lift operator and stage polymorphism (Section 3).

• Based on $\lambda_{12}$ we add an untyped front-end in Scheme, and discuss a first use case of interpreter specialization (Section 4).

• As an alternative to the untyped model, we discuss how $\lambda_{12}$ can serve as a faithful model for a typed embedding of LMS (Lightweight Modular Staging) in Scala (Section 5).

• We discuss stage polymorphism, and how it can be achieved through functional abstraction in the untyped setting and with type classes in a typed setting (Section 6).

• We present a meta-circular interpreter for Pink, a restricted Lisp front-end, and demonstrate that we can collapse arbitrarily many levels of self-interpretation via compilation: this achieves our challenge of collapsing towers of interpreters (Section 7).

• We extend Pink with a simple mechanism for reflection that enables user programs to execute expressions as part of an interpreter at any level in a tower (Section 8).

• We develop these ideas further into the language Purple, a variant of Asai’s reflective language Black, where every aspect of the semantics can change dynamically, yet programs can be recompiled on the fly to adapt to modified semantics (Section 9).

• We present a range of examples in Purple / Black that make extensive use of reflection (Section 10).

We discuss related work in Section 11.
2. Preliminaries
It is well known that interpreters and compilers are fundamentally linked through specialization, as formalized in the three Futamura projections [20]. First, specializing an interpreter to a given program yields a compiled version of that program. Second, a process that can specialize a given interpreter to any program is equivalent to a compiler. Third, a process that can take any interpreter and turn it into a compiler is a compiler generator, also called cogen.

For a given interpreter, the corresponding compiler is also called its generating extension [16]. Since compilers are often preferable to interpreters, and preferable to running a potentially costly specialization process on an interpreter for every input program, how does one compute the generating extension of a given program?

The third Futamura projection tells us that double self-application of a generic program specializer is one way to produce a 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.

Historically, program specialization or partial evaluation [29] was conceived as a fully automatic technique. However, hopes of full automation have been largely abandoned since the early 2000s. In the simplest possible setting, partial evaluation can be viewed as a form of normalization, which propagates constants and performs reductions wherever it encounters a redex, i.e. a combination of introduction and elimination form. But most interesting languages are not strongly normalizing, i.e. uncurred eager reduction might diverge, and even for terminating languages or programs it can lead to exponential blow-up due to duplicating of control-flow paths. This means that some static redexes need to be residualized – but how to pick which ones to reduce, and which ones to residualize?

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

In the late 1990s, it has been realized that if one starts with a binding-time annotated interpreter, expressed in a multi-level language, then deriving a cogen by hand is actually quite straightforward [6, 50]. What is more, when starting from a multi-level program, it is actually easy to derive the generating extension itself! Thus, multi-level languages have attracted interest in their own right as tools for programmable specialization, as evidenced for example by MetaML [49] and MetaOCaml [9].

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

\[ n \mid x \mid e \beta e \mid \lambda x. e \mid \ldots \]

The binding-time annotations \( \beta \) 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 \( \beta \) ranges over \( S, D \) for static or dynamic, but in more elaborate systems (e.g. in [22, 50]) \( b \) can range over integers or include variables \( \beta \) for polymorphism [27].

Multi-stage languages in the line of MetaML [49] feature quasiquotation syntax instead:

\[ n \mid x \mid e \Theta e \mid \lambda x. e \mid \langle e \rangle \sim e \mid \run e \mid \ldots \]

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

Other systems are implemented as libraries in a general-purpose host language, e.g. LMS [40] in Scala. Multi-level languages differ also quite significantly in their semantics. MetaML and its descendents, for example, provide hygiene guarantees for bindings, but interpret quotation in a purely syntactic way. This can lead to reordering or duplicating of quoted expressions, which is often undesirable, in particular when combined with side effects.

3. Kernel Language
With an eye towards the challenge posed in the introduction, we present a new multi-level kernel language \( \lambda_{2} \) which combines a number of desirable features. Like MetaML, it contains facilities to run residual code. Like polymorphic BTA [27], it supports stage- or binding-time polymorphism. Like LMS, its evaluation preserves the execution order of future-stage expressions. But unlike most other systems, \( \lambda_{2} \) does not require a type system or any other static analysis. Its key mechanism is a polymorphic lift operator that turns a static, present-stage, value into a future-stage expression.

We show the syntax in Figure 1. For simplicity we use a nameless DeBruijn level representation. The term syntax contains the usual \( \lambda \)-calculus constructs, plus operators Lift and Run. The value syntax contains standard constants, tuples, closures, and in addition Code objects that hold expressions. The polymorphic lift operator in Figure 2 is inspired by a corresponding facility in normalization by evaluation (NBE) [13, 5]. Its purpose is to convert values into future-stage expressions. Numbers are immediate, tuples element-wise, functions are memoized and create \( \lambda \)-abstractions via two-level \( \eta \)-expansion, as in NBE. Helper functions like reflect and reify are shown in Figure 4. They serve to create future-stage code in administrative normal form (ANF) [17] to maintain the relative evaluation order of expressions. This is a standard practice in partial evaluators that deal with state and effects, and otherwise known as "let-insertion" [7, 26, 34, 52]. Finally, lifting a code value wraps it in a Lift expression.

The main multi-stage evaluation semantics is given by a definitional interpreter in Figure 3, written in Scala. The key observation is that introduction forms (e.g. Lift, Lam, Pair) 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
elements, but a variant with automatic conversion of primitive constants
would be conceivable as well. A curious case is
expression for the future stage. Hence,
argument, which is not evaluated, but solely exists
will generate a call to
expressions in a scope can be captured into a sequence of Let
ings via reify. The top-level entrypoint to multi-level evaluation is
evalmsg in Figure 3, which delegates to evalms and also packages
up and returns all generated code, if any.

4. Untyped Front-End
We show first how \( \lambda_{\mu} \) can serve as a model for multi-level evalua-
tion in untyped languages. To do so, we implement a small Lisp
reader, that translates S-expression to \( \lambda_{\mu} \) syntax. The mapping is
straightforward, with proper names vs DeBruijin levels being the
biggest difference. We also introduce syntactic sugar for multi-
argument functions, and we extend the core language slightly to
add support for proper booleans, equality tests, and a few other

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.

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? s) #t
      (if (null? s) #f
          (if (eq? (car r) (car s))
              ((matches (cdr r)) (cdr s)) #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 \( \text{run} \), the current (static) pattern character to be
compared with the (dynamic) input character.

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

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

Recall that the 0 argument to run designates the desired “run now”.
The resulting generated code has only the low-level operations on
the dynamic input s. The static input r causes three unfolding
of the matches body with the first static if disappearing from the
generated code. The generated code is identical to the internal ANF
representation of the following user-level program:

```
(define start_ab (lambda (s)
  (if (null? s) #f
      (if (eq? 'a (car s))
          (let (s (cdr s))
            (if (null? s) #f
                (if (eq? 'b (car s2))
                    #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 Section 7.

5. Typed Front-Ends
Having seen how \( \lambda_{\mu} \) can serve as a model for untyped front-ends,
we now turn our attention to typed models of multi-stage evalua-
tion. Since we are already working in Scala, we show how \( \lambda_{\mu} \) re-
lates to LMS [40], a library-based multi-stage programming frame-
work in Scala. LMS uses a type constructor Rep[T] to designate
future-stage expressions, i.e. those that evaluate to Code values in
\(\lambda_t\), and provides overloaded method on such \(\text{Rep}[T]\) values, e.g. + for \(\text{Rep}[\text{Int}]\). Thus, Scala’s local type inference and overloading resolution performs a kind of local binding-time analysis.

Essentially we take the \emph{evalms} function from Section 3 and turn it inside out, from an interpreter over an initial term language-\(\exp\) to an evaluator in tagless-final [10] style. Doing so, we can assign the following overloaded type signatures to the \(\lambda_t\)-operations:

\[
\begin{align*}
def \text{app}(f, x) &= f(x) \\
def \text{lift}(v) &= \text{reflect}(\text{App}(s1, s2)) \\
def \text{if}_a(c, a, b) &= a \Rightarrow b \\
def \text{run}(b, e) &= \text{run}(0, \text{lift}(\text{matches}([\text{'a'}, \text{'b'}]))) \\
end{align*}
\]

The run construct built on top of \emph{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 \text{lift} for primitives could be dropped by declaring the corresponding lift function as implicit. We have left them in here for clarity.

6. Stage Polymorphism

In the examples so far, we started with a plain, unmodified, interpreter program and added lift annotations in judicious 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 \text{matches-gen} is the generic matches function, and \text{matches-spec} is the one including lifts from Section 4, we would like to write:

\[
\begin{align*}(\text{define} \text{matches-maybe-spec} (\text{lambda} (r) \text{(if} (< (\text{length} r) 20) (\text{run} 0 \text{(lift \text{matches-spec} r})) (\text{matches-gen} r))))
\end{align*}
\]

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

\[
\begin{align*}(\text{define} \text{matches} (\text{lambda} (\text{maybe-lift} (\text{lambda} (r) (\text{lambda} (s) \text{(if} (null? r) (\text{maybe-lift} #t) (\text{(if} (null? s) (\text{maybe-lift} #f) (\text{(if} (eq? (\text{lift} (\text{car} r)) (\text{car} s)) (\text{matches} (\text{cdr} r)) (\text{cdr} s))) (\text{maybe-lift} #f))))))))) \text{maybe-lift})
\end{align*}
\]

Now, we can define \text{matches-spec} and \text{matches-gen} simply as:

\[
\begin{align*}(\text{define} \text{matches-spec} (\text{matches} (\text{lambda} (e) (\text{lift} e)))) (\text{define} \text{matches-gen} (\text{matches} (\text{lambda} (e) e)))
\end{align*}
\]

Which completes our stage-polymorphic string matcher.

The question now is, how to achieve the same in a typed setting? We again consider \emph{LMS}, which relies on the normal Scala type system without specific support for polymorphic binding time abstraction and application operators as in [27]. In Section 5 we had a fixed type distinction between normal types \text{T} and staged types \text{Rep}[T], and we had overloaded methods based on those static types. The key idea here is to introduce another higher-kindded type \text{R}[T] which can be instantiated with either \text{T} or \text{Rep}[T] in a given context. But how do we get the correct operations on \text{R}[T] values?

It turns out that we can leverage type classes [53] to good effect. We define a type class interface:

\[
\begin{align*}(\text{trait} \text{Ops[R[_]} \{ \\
\text{def lift(v:R)} \}
\end{align*}
\]

Here is the specialization to pattern a b:

\[
\begin{align*}(\text{val start_ab} = \text{run}(0, \text{lift}(\text{matches}([\text{'a'}, \text{'b'}]))))
\end{align*}
\]

For comparison with Section 4, here is the staged string matcher in LMS, using regular and staged lists:

\[
\begin{align*}(\text{def matches(r: List[Char]): s: Rep[List[Char]]} &= \{ \\
&\text{if} (\text{r.isEmpty}) \text{lift}(\text{true}) \text{else} \\
&\text{if} (\text{r.isEmpty}) \text{lift}(\text{false}) \text{else} \\
&\text{if} (\text{lift(r.head) == s.head}) \text{matches(r.tail, s.tail)}
\end{align*}
\]
And corresponding instances of Ops[NoRep] and Ops[Rep] (where NoRep[T] = T) that delegate to the appropriate base methods on plain types or Rep types respectively.

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

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

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

Note that the val o = ...; import o._ on the first line 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:

```scala
def matches-maybe-spec(r: List[Char]) = matches[R[_]:Ops](r: List[Char])(s: R[List[Char]) = {
  if (r.isEmpty) lift(false)
  else
    if (lift(r.head) == s.head) matches(r.tail, s.tail)
    else lift(false)
}
```

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

## 7. 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 described in the introduction: collapsing *towers* of interpreters, i.e. sequences of multiple interpreters interpreting each other as input programs. Stage polymorphism is the key mechanism to make interpreters compose in such a collapsible way.

We start by defining a meta-circular interpreter shown in Figure 5, for a slightly more restricted Lisp front-end that is closer to \( \lambda_1 \). 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 closure reference, as in \( \lambda_1 \) directly. For non-recursive functions we will use \( \_ \) instead of an identifier. The interpreter in Figure 5 is binding-time parametric (parameter maybe-lift), so it can also act as a compiler. It also uses open recursion (parameter eval), so that it can be customized from the outside.

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

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

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

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

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

We can also verify that double and triple interpretation work, given a quoted eval-src:

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

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

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

And we can use evalc in place of eval to verify:

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

Similarly, we get:

```scala
> ((eval evalc-src fac-src) 4) ;; 24
> ((eval evalc-src eval-src) 4) ;; 24
> ((eval evalc-src evalc-src) 4) ;; 24
```

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

```scala
> ((eval evalc-src evalc-src fac-src) 4) ;; 24
```

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

We can also exercise this pattern with our string matcher acting as the top compiler in a chain: we obtain a string matching compiler that operates through arbitrarily many levels of self-interpretation. Figure 6 shows a more complete string matcher written directly in Pink. Compared to the previous sections, this version also handles * patterns, and will search for a match anywhere in a string.

## 8. Towards Reflective Towers

We now turn Pink into a proper, albeit simple, reflective language by enabling programs to observe the behavior of a running inter-
In a fully reflective programming language such as Black [3], programs are interpreted by an infinite tower of meta-circular interpreters. Each level of the tower can be accessed and modified, so the semantics of the language changes dynamically during execution. Previous work [2] used staging to specialize a function with respect to the built-in semantics of the tower. But a key question was left open [2]: can we specialize a function with respect to the current semantics of the tower, which is (1) possibly user-modified, and (2) possibly also compiled via staging?

Here, we answer in the affirmative, showing that, using staging, it is possible to compile a user program under modified, possibly also compiled, semantics. For flexibility and variety, we switch our base implementation to one entirely built on LMS.

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

\begin{verbatim}
(define fib (lambda (n) 
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
\end{verbatim}

At the meta-level, we change the evaluation of variables so that it increments a meta-level counter when a variable name is \texttt{n}. The special form \texttt{EM} shifts the tower up, so that its argument executes at the meta-level. By changing the definition of the function eval-var, we modify the meaning of evaluating a variable one level down. The interpreter function takes three arguments: the expression, environment and continuation from the level below.

\begin{verbatim}
(define counter 0)
(define old-eval-var eval-var)
(set! eval-var (lambda (r k) 
  (if (eq? r 'n) 
    (set! counter (+ counter 1)) 
    (old-eval-var r k))))
\end{verbatim}

We can compile a function by defining it with \texttt{clambda} instead of \texttt{lambda} (as we do for eval-var). Our goal is that the behavior of a \texttt{clambda} matches that of a \texttt{lambda} when applied, assuming the current semantics remain fixed.

\begin{verbatim}
(set! fib (clambda (n) 
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
\end{verbatim}

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.

As for the generated code, the \texttt{fib} function compiled under the modified semantics has extra code (summarized in black) for incrementing the counter for each occurrence of the variable \texttt{n} in its body, but is otherwise similar to the code (summarized in gray) compiled under the original semantics.

\begin{verbatim}
(\{k, xs\} \Rightarrow 
  _app\('<\texttt{\_cons\_cell\_read\_cell\_counter}', '\texttt{1}'), \_cont\(c_1\) \Rightarrow 
  _cell\set\(\texttt{\_cont\_cell\_counter}, c_1\) 
  _app\('<\texttt{\_cons\_car(xs)}', '\texttt{2}'), \_cont\(v_1\) \Rightarrow 
  _if\texttt{\_true}(v_1),
\end{verbatim}

9. Purple: Reflection à la Black

We now consider a more powerful reflective tower, where we can not only launch new interpreters with modified semantics, but also modify the semantics of the \texttt{currently executing} interpreter (tower). Our language, dubbed Purple, is heavily inspired by Asai’s reflective language Black [3]. In the spirit of Black, a couple of changes are necessary. In particular, we need mutable state, and we also need continuations.
In the generated code, mutable cells are directly referenced, outside the concepts (such as environments) explicitly represented in the user-accessible interpreters. The modified interpreter function from the meta-level is inline into code compiled for the user-level. Compilation collapses the levels of the tower.

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

### 9.1 Fixed Tower Structure

In a “classic” tower of interpreters, as in languages 3-Lisp [46], Brown [54], Blond [15], or Black [3, 2], each level has its own environment and its own continuation. Conceptually, the semantics of one level is given by the interpreter at the level above, which takes an expression, an environment and a continuation, together with a stream of all the environments and continuations of the levels above.

For compilation, we’d 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 evaluation with 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 fixed level structure, where level 0 is interpreted by level 1, which is interpreted by level 2, and so on. Each level has its own global environment, containing bindings for primitives (such as \texttt{null?}, \texttt{+}, \texttt{display}) and, except level 0, for interpreter functions (such as \texttt{base-eval}, \texttt{eval-var}, \texttt{base-eval}). Each interpreter function takes as arguments an expression, an environment and a continuation. These arguments usually come from the level below, and are manipulated as reified structures. In addition, an interpreter function statically knows about its fixed meta-environment (the stream of all environments in levels above and including its own) and is dynamically given a meta-continuation, which represents the rest of the computation or context around the interpreter function call. Similarly, closures, created from \texttt{lambda}, save not only their lexical environment, but also their lexical meta-environment.

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

In our case, we avoid all these thorny issues by using a fixed structure and a lexical discipline for both environments and meta-environments.

```scala
(begin (define where_am_i 'user)
  (EM (define where_am_i 'meta))
  (EM (let (old-eval-var eval-var
    (_k (lambda (k x))
      (set! eval-var (lambda (e r k)
        (if (eq? e '__k) (k __k)
          (begin
            (if (eq? e _) (set! _k k) '(')
            (old-eval-var e r k))))))))
    (define _0)
    (> _ _ 1) ;; => 1
    > where_am_i ;; => user
    > (EM where_am_i) ;; => meta
    > (_k _ 2) ;; => 3
    > where_am_i ;; => user
    > (EM where_am_i) ;; => meta (in Black: user!)
    
    That calling the continuation _k pushes another user-level in Black may seem a bit strange and counterintuitive.
    
    Even though the tower structure in Purple is fixed, the semantics is not, since interpreter functions can be redefined using \texttt{set!}. Hence, in the host language, we do not hard-code references to mutually-recursive interpreter functions, but look up such references in the meta-environment (via \texttt{meta_apply}).
    
    ### 9.2 Three Languages of Purple
    
The 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 \texttt{Ops[R]}.

    Values are represented in the host language as follows:

    ```scala
    Value ::= I(n: Int)
    | R(b: Boolean)
    | S(sym: String)
    | Str(s: String)
    | P(Car: Value, cdr: Value)
    | Clos(parameters: Value, body: Value, env: Value, envv: MEnv)
    | EvalFun(key: Int)
    | Cell(key: String)
    | */ ... null, primitives, continuations ... */
    ```

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

    The compilation language necessary for turning the interpreter into a compiler is small:
    ```scala
    trait Ops[R] {?
      implicit def _lift(v: Value): R[Value]
      def _liftb(b: Boolean): R[Boolean]?
      ```

    7
    2016/8/30
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 eval_apply again possibly going further up in the levels. If we have a compiled or built-in function, then we just apply it as a black-box, thus breaking out of an infinite-tower regress.

9.3 Initializing the Tower

Below is the code for building the tower level structure. Instead of meta-continuations, Purple uses meta-environments. The representation of interpreter functions is wrapped to look just like user-defined compiled functions. The meta-environment being created is passed lazily on to the interpreted functions during frame initialization. The meta-environment constructor takes an environment as the id type constructor or with eager evaluation.

This trait can be instantiated either for normal interpretation with R as the id type constructor or with R = Rep for compilation using LMS, as discussed in Section 5.

9.4 Base Evaluation

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

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.
case class eval_clambda[R[_]:Ops](m: MEnv, 

in which parameters of the clambda need to switch from interpretation to compilation. In any case, we arguments.

However, id_cont k = or._car(kv) meta_apply[R](m, S("eval-begin"), body, env_extend[R](env, params, Code(args)), unwrap_cont[R](f(k)))

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, 

val o = implicitly[Ops[R]]; import o._
val P(_, P(params, body)) = exp
def eval_body[R[(_:Ops)](kv: R[Value]): R[Value] = {
val or = implicitly[Ops[R]]
val args = or._cdr(kv)
lazy val k = or._car(kv)
meta_apply[R](m, S("eval-begin"), body, env_extend[R](env, params, Code(args)), unwrap_cont[R](f(k)))
}
val f = if (linRep) {
/* switch to compilation mode */
trait Program extends EvalDsl {
  // insert into funs table
} else {
/* already in compilation mode */
/* need to create a poly-stage function */
  def fun(new Fun[R]) {
    def fun[R[_]:Ops](kv: R[Value]) =
      eval_body[R](kv)
  }
apply_cont[R](cont, f)
}

9.5 Summary

In Purple, we showed that it is possible to compile a function in a tower of interpreters, under current, possibly user-modified, and possibly compiled, semantics. We took several design decisions, and used various techniques.

• In our design, the level structure of the tower is fixed. There is one global continuation representing the rest of the computation, instead of one continuation per level. The fixed level structure and one global “pc” counter simplifies reasoning about compilation, optimization, and application (as both environment and meta-environment are lexically scoped).

• Our system is comprised of three languages: the host language (Scala), the user language (the tower), and the compilation language (the lifted operations).

• Side-effects are explicit in the compilation language (so the system knows to generate code instead of performing the side-effects at code-generation time), not explicitly exposed in the user language, and freely manipulated in the host language.

• Staging is driven by types, abstracting over staging decisions using type classes. Because the body of functions to specialize is static, we can get away with a small set of lifted operations. We can also use host language features, such as pattern-matching, without lifting.

• There is no difference between built-in functions and compiled functions. Both can be used for interpretation or compilation. Hence, the generated code is also stage-polymorphic. This is pushing the technique to its extreme: interpreter vs compiler via staging, both interpreter and compiler via stage polymorphism, stage-polymorphic generated code.

• Stage polymorphism is complicated by first-class code (such as nested functions and continuations). These nested code fragments have to be polymorphic, and independent of their outer context. This requires converting from the outer context to the nested one. Because generated code is also stage-polymorphic, the stage is actually a dynamic property of a call.

In terms of alternative or further designs, here are some suggestions and questions for future work:

• It would be interesting to track semantic changes like assumptions in a JIT, in order to recompile or deoptimize functions compiled under outdated semantics. The system would need to make cell reads and writes more explicit so they can be tracked and adapted. In the user language, cell operations are currently implicit, while in the host language, cells can be freely manipulated. Currently, it is only in the compilation language that reads and writes are explicit.

• Stage-polymorphism has the overhead of applying the lifted operations (even in the case of a NoRep instantiation). Could we generate two versions of the code, one for interpretation and one for compilation? What about nested functions and continuations in this case? How would they maintain their two modes, independently?

• What about tail-recursion and tail-reflection? The stage-polymorphic interpreter, even without meta-applications, is not tail-recursive, because _if has a recursive call in each branch. When generating code, we do want to explore both branches.

• We duplicate the continuation in each branch of an _if expression. Collapsing also duplicate code through inlining, for example. Can we better control the generated code blow-up?

• Could we add common (re-)definitions, to all levels of the towers at once, from the user language?

• What guarantees can we make with respect to the semantics of compiled functions?

• Would it make sense to make the process of compilation and optimizations under the control of the user language?

By showing that it is possible to compile under user-modified semantics in a tower of interpreters, it is our hope that this work fosters further explorations in systems that combine compilation and reification/reflection.

10. 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.

10.1 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:

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 ((1) (a b c)) (((1 a b c))

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

10.3 Refilers

In towers of interpreters, a refiler is a way to go up the tower and get a reified structure for the current computation from below. From level n, the expression (delta (e r k) body ... args ...) evaluates the expression body... with the environment from level n + 1, with b 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:

> (+ 1 (call/cc (lambda (k) k)))  ;; 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 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:

> (+ 1 (call/cc (lambda (k) (k (meaning (call/cc (lambda (v) v)) 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.

> (+ 1 (call/cc (lambda (k) (if (pair? e) (if (pair? (car e)) (eq? ’delta (car (car e))) #f) #f)))

Using our special form on a particular instance of the problem, we see what happens when we go There And Back Again (TABA). As we walk down the first list, we push elements on the stack, that
10.4 Meta-Level Undo

We can implement undo at the meta-level, so that it’s easy to experiment with changes.

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

```
(define old-eval-set! eval-set!)
(set! eval-set! (_dummy (_dummy _dummy))
(lambda (e r k)
  (if (null? _env)
      (begin
        (old-eval-set! _env)
        (list 'set! (car _env))
        (list 'quote (cdr _env)))
      r
      (lambda (v) v))
  (set! undo-list (cdr undo-list))))
```

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

```
(define undo! (_dummy _env))
```

10.5 Semantics of Continuations

As pointed out in the code for static_apply, applying a continuation jumps out of the surrounding context, ignoring it. We can imagine alternative behaviors. Let’s re-define eval-var to explore some alternatives.

```
(define old-eval-var eval-var)
(set! eval-var (_dummy _dummy)
(lambda (e r k)
  (if (eq? '_dummy e) (begin (k 1) (k 2) (k 3))
    (old-eval-var e r k))))
```

What should happen when we evaluate _dummy? In our system, the default behavior is to jump out at the first call of k, so the result is 1.

> _dummy ;; => 1

In a “classic” tower system, (k 1) returns 1 to the user-level. When the user exits to the meta-level, the meta-continuation resumes, and (k 2) returns 2 back to the user-level (and “in time”). So there’s a ping-pong between the user-level and the meta-level until the meta-continuation is back to the top one, for the meta-level REPL. In Black:

> _dummy ;; => 1
> (exit 'up) ;; => 2
> (exit 'up) ;; => 3
> (exit 'up) ;; => up (meta-level)

Our system lacks meta-continuations, so we cannot exactly reproduce this behavior. Still by re-defining base-apply at the meta-level, we can explore alternative semantics for how continuations at the meta-level should behave.

**Auto-Continue**  One alternative is to simply continue with the outer continuation, after the inner one completes. Let’s change the meta-level:

```
(define old-base-apply base-apply)
(set! base-apply (_dummy _dummy)
(lambda (fa r k)
  (if (continuation? (car fa))
    (k (old-base-apply fa r (lambda (x) x)))
    (old-base-apply fa r k))))
```

Now, by the semantics of begin, we execute the first call to k, ignore the result, and continue with the second, and finally third one:

> _dummy ;; => 3

**Manually Resume**  To simulate something like the “classic” tower, but with only one program counter, we can record the pending outer context in a stack instead of ignoring it. For this, we call the continuation, record the resulting value to know how to resume with the outer continuation, and return the value.

```
(define old-base-apply base-apply)
(set! base-apply (_dummy _dummy)
(lambda (fa r k)
  (if (continuation? (car fa))
    (let ((v (old-base-apply fa r (lambda (x) x))))
      (lambda (v) v))
    (old-base-apply fa r k))))
```

At the user-level, we provide a convenient function to resume!

```
(define resume! (lambda ()
               (if (null? (EM EM pending-thunks))
                 'done
                 (EM EM (let ((thunk (car pending-thunks))
                            (set! pending-thunks (cdr pending-thunks))
                            (thunk)))))))
```

Resuming behaves more or less like we expect, except that 3 appears twice: once for the final call to k, and once after that because of the id_cont that we use in the host implementation for an empty outer continuation.

> _dummy ;; => 1
> (resume!) ;; => 2
> (resume!) ;; => 3
> (resume!) ;; => 3 // id_cont :
> (resume!) ;; => done

10.6 Collapsing User-Level Interpreters

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

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

We can generate code for matching a particular prefix as follows:
11. Related Work

Partial Evaluation Partial evaluation [31] 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 [24], an annotation-directed specializer for C. Jspec/Tempo [43], the JSC Java Supercompiler [33], and Civet [44].

Partial evaluation has addressed higher-order languages with state using similar let-insertion techniques as discussed here [7, 26, 34, 52]. Further work has studied partially static structures [36] and partially static operations [51], and compilation based on combinations of partial evaluation, staging and abstract interpretation [47, 12, 32]. Two-level languages are frequently used as a basis for describing binding-time annotated programs [31, 37].

Multi-level binding-time analysis extends binding-time analysis from two stages to more levels [22]. Polymorphic binding-time analysis extends binding-time analysis so that some expressions can be assigned multiple stages [27]. Our kernel language complements such binding-time analyses: the stages are explicit, but can be abstracted over, just like in polymorphism. However in the kernel language, we still want to have an API instead of doing things purely automatically.

The “writing cogen by hand approach” is due to Birkedal et al. [6], and was picked up by Thiemann [50]. Jones made a case for program transformation via interpreter specialization [30]. Glück has shown how optimizing specializers can be deriving by layering and specializing interpreters [23, 21].

Multi-stage programming Multi-stage programming (MSP, staging for short), as established by Taha and Sheard [49] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [9] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [40] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code. LMS draws inspiration from earlier work such as TaskGraph [4], a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [41, 8, 42, 35, 48].

Reflective Towers Smith introduced reflective towers in seminal papers on 3-Lisp [45, 46]. The motivation stems from enabling processes to inspect on their computation arbitrarily. Friedman and Wand distill the essence of reflection in Brown [18, 54], explaining reflection and reification in a self-contained semantics, which does not re-allude to reflection. Later, Friedman and Jefferson also give a simplified account for a finite tower, \( L_\infty \) [28]. Danvy and Malmkjær present a denotational semantics of Blond [15]. Their account justifies the use of meta-continuations for a compositional semantics. As discussed earlier, our Purple reflective tower is inspired chiefly by Asai’s Black [3, 2].

Program Generators A number of high-performance program generators have been built, for example ATLAS [55] (linear algebra), FFTW [19] (discrete fourier transform), and Spiral [38] (general linear transformations). Other systems include PetaBricks [1], and CVXgen [25].

12. Conclusions

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

More philosophically – if we were to view programming as a science, it would be the science of controlling processes that – deep down – blindly manipulate symbols. We need to explore conceptual toolkits that shifts the perspective, which enables concise reasoning by projecting away other concerns. Lisp is one example: linguistically, it shifts the lens from syntax to semantics. How can we explore deeper levels (e.g. pragmatics)? Reflective towers of interpreters are an intriguing concept to bridge across arbitrary levels of shifting. Naming is blessing, granting another “tool for thought”. As engineers, we like to build hierarchical systems. As scientists, we like to study processes from different angles, which yields various inconsistent hierarchies. This is why reflective towers are so intriguing: a lens for lens-ing.

References


