Collapsing a Reflective Tower:
Staging under User-Modified Semantics

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Abstract
In a reflective programming language such as Black, user 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.

In this project, we show that, using staging, it is possible to compile a user program under modified, possibly also compiled, semantics – a question raised in previous work (Asai 2014).

How? All functions are polymorphic as to whether they generate code or run. Generated code, when compiled, is also polymorphic in this way so that there’s no difference between built-in and compiled functions.

Compiling a function in a tower of interpreters effectively collapses the tower.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors–Code Generation, Optimization, Compilers; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages–Operational semantics, Partial evaluation

Keywords Reflection, metacircular interpreter, partial evaluation, binding-time analysis, staging, Scala

1. Introduction
In a 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.

We answer a question left open in previous work (Asai 2014), which used staging to specialize a function with respect to the built-in semantics of the tower: can we specialize a function with respect to the current, possibly user-modified, semantics of the tower?

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

```lms-black
(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. The special form EM shifts the tower up, so that its argument executes at the meta-level. By changing the definition of the function eval-var, we modify the meaning of evaluating a variable one level down. The interpreter function takes three arguments: the expression, environment and continuation from the level below.

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

We can compile a function by defining it with clambda instead of lambda (as was done for eval-var). Our goal is that the behavior of a clambda matches that of a lambda when applied, assuming the current semantics are fixed.

```lms-black
(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.

```lms-black
(set! eval-var old-eval-var)
> (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.

```lms-black
{(k, xs) => _app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_1 => _cell_set(<cell counter>, c_1)...
```

Copyright notice will appear here once ‘preprint’ option is removed.
In the generated code, mutable cells are directly referenced, outside the concepts (such as environments) explicitly represented in the user-accessible interpreters. The modified interpreter function from the meta-level is inlined into code compiled for the user-level. Compilation collapses the levels of the tower.

1.2 Roadmap

• We define a tower of interpreters, whose semantics is well-suited to compilation (Section 2).

• We apply staging to compile a program in a tower of interpreters, combining several generative patterns: turning interpreters into compilers, abstracting over staging decisions, generating code that can also be staged (Section 3).

• We show several examples in our system, confirming that compilation has reasonable behavior (Section 4).

• We offer a parting perspective (Section 5).

2. Tower of Interpreters

In a “classic” tower of interpreters (Smith 1984; Wand and Friedman 1986; Danvy and Malmkjær 1988; Asai et al. 1996; Asai 2014), 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.

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

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

Below is the code for building the tower level structure. 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 interpreter functions during frame initialisation. The meta-environment constructor takes an environment eagerly, and the next meta-environment lazily; thus, the stream of environments fixes infinitely many upper levels, potentially.

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

In a “classic” tower, a continuation eventually returns an Answer type, which is usually different from a regular Value type. In our system, continuations just return regular values, instead of answers.

3. Compilation via Staging

In the partial evaluation and staging literature (Futamura 1999; Consel and Danvy 1993; Rompf and Amin 2015), it is well-known how to turn an interpreter into a compiler: make the program itself static (known during code generation) and the input to the program dynamic (unknown during code generation).

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

In LMS (Rompf and Odersky 2012), staging is driven by types. An expression of type Rep[T] is dynamic (known during the first stage), while an expression of bare type T is static (known during the first stage). Scala’s local type inference performs binding-time inference.

In LMS, we can abstract over staging decisions by abstracting over the higher-kinded type Rep (Ofenbeck et al. 2013). Given a parameter higher-kinded type R[_], and a type class Ops[R[_]], which defines the operations on R[_] types, we instantiate R to NoRep (defined as type NoRep(A) = A) and Rep to either interpret or compile.

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).

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

The result is 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].

The evaluation of variables shows why the result needs to be dynamic during code generation. If the variable points to a cell in the environment, then we need to generate code to read the cell. We most likely don’t want to read the cell during code generation, because the result of the read depends on writes to the cell. If the variable points to a code value in the environment, the dynamic value represents a symbolic value, such as an argument of a clambda for which we are generating code.

```scala
def eval_var[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  env_get(env, exp, match {
    case Code(v: R[Value]) => apply_cont[R](cont, _lift(v))
    case v@Cell(_) =>
      apply_cont[R](cont, _cell_read(v))
    case _ =>
      apply_cont[R](cont, _lift(v))
  })
}
```

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.

```scala
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 “unlift” the continuation argument k, because meta_apply like other interpreter functions expects a known continuation.

```scala
def eval_clambda[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  eval_body[RF[_]](op)(OpsRep)
  val body = env_get(env, exp) match {
    case case P(S("clambda"), _) =>
      meta_apply[R](m, S("eval-clambda"), exp, env, cont)
    case _ =>
      meta_apply[R](m, S("eval-application"), exp, env, cont)
  }
}
```

```scala
def eval_EM[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  env_get(env, exp, match {
    case Code(v: R[Value]) => apply_cont[R](cont, _cell_read(v))
    case v@Cell(_) =>
      apply_cont[R](cont, _cell_read(v))
    case _ =>
      apply_cont[R](cont, _lift(v))
  })
}
```

```scala
def eval_var[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  env_get(env, exp, match {
    case Code(v: R[Value]) => apply_cont[R](cont, _lift(v))
    case v@Cell(_) =>
      apply_cont[R](cont, _cell_read(v))
    case _ =>
      apply_cont[R](cont, _lift(v))
  })
}
```

```scala
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 “unlift” the continuation argument k, because meta_apply like other interpreter functions expects a known continuation.

```scala
def eval_clambda[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  eval_body[RF[_]](op)(OpsRep)
  val body = env_get(env, exp) match {
    case case P(S("clambda"), _) =>
      meta_apply[R](m, S("eval-clambda"), exp, env, cont)
    case _ =>
      meta_apply[R](m, S("eval-application"), exp, env, cont)
  }
}
```

```scala
def eval_EM[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  env_get(env, exp, match {
    case Code(v: R[Value]) => apply_cont[R](cont, _cell_read(v))
    case v@Cell(_) =>
      apply_cont[R](cont, _cell_read(v))
    case _ =>
      apply_cont[R](cont, _lift(v))
  })
}
```

```scala
def eval_var[R[_]:Ops](m: MEnv,
  val o = implicitly[Ops[R]]; import o._
  env_get(env, exp, match {
    case Code(v: R[Value]) => apply_cont[R](cont, _lift(v))
    case v@Cell(_) =>
      apply_cont[R](cont, _cell_read(v))
    case _ =>
      apply_cont[R](cont, _lift(v))
  })
}
```

```scala
case class Code[R[_]](c: R[Value]) extends Value
```
3.1 Stage Polymorphism for First-Class Values

The type of a compiled function is Value ⇒ R[Value]. The type of a
continuation is R[Value] ⇒ R[Value]. The continuation or function
is created in a certain stage, say W[\_\_]:Ops. If W = NoRep, then we
might need to lift NoRep[Value] = Value to R[Value]. If W = Rep, then
we are generating textual code, which will abstract over the stage,
onece more — so let’s think of it as just W. So, we need to convert
W[Value] to R[Value] in order to use values from the context. These
contexts can be nested, so we need to transitively convert from any
outer W[Value] to any inner R[Value].

trait Convert[W[\_\_], R[\_\_]] {  
  implicit def convert(v: W[Value]): R[Value]  
}  
 implicit def convertNoRep[R[\_\_]:Ops] = new Convert[NoRep, R] {  
  val o = implicitly[Ops[R]]  
  def convert(v: Value) = o._lift(v)  
}  
 implicit def convertSame[R[\_\_]] = new Convert[R, R] {  
  def convert(v: R[Value]) = R(v)  
}  
 def convertTrans[R1[\_\_], R2[\_\_], R3[\_\_]] {  
  implicit ev1: Convert[R1, R2] = ev1  
  implicit ev2: Convert[R2, R3] = ev2  
  def convert(v: R1[Value]) = ev2:convert(ev1:convert(v))  
}  
 def convert(v: R[Value]) = v  
 o = implicitly[Ops[R]]  
 new Convert[R1, R2, R3] {  
  implicit ev1: Convert[R1, R2] = ev1  
  implicit ev2: Convert[R2, R3] = ev2  
  def convert(v: R1[Value]) = ev2:convert(ev1:convert(v))  
}  

Now, the type of a compiled function or a continuation is with
respect to a stage context, W. The actual function is stage-
polyomorphic, given evidence that we can convert from W to R.

abstract class Fun[W[\_\_]:Ops] {  
}  
 abstract class FunC[W[\_\_]] {  
  def fun[R[\_\_]:Ops][implicit ev: Convert[W,R]]: R[Value] ⇒ R[Value]  
}  

In the host language, this style of polymorphism is unfortu-
nately cumbersome to use.

First, we turn interpreter function definition into actual first-
class functions to put in the initial environment for a meta-level.
Here is the wrapping for base_eval, given evidence that we can convert from W to R.

val base_eval_fun(m: ⇒ MEnv): Fun[NoRep] = new Fun[NoRep] {  
  def fun[R[\_\_]:Ops][implicit ev: Convert[NoRep,R]] = (  
    (vc: Value) ⇒  
      val P(m, P(ev, P(env, P(kv, cont)))) = kc  
      apply_cont(R)(m, ev, env, cont)  
  )  
}  

Notice that we apply the active continuation mcont passed into
the function. This ensures that interpreter functions can be used in non-
tail positions too.

Here is the missing part from eval_clambda:

/*@ ... */  
} else {  
  /* already in compilation mode */  
  /* need to create a poly-stage function */  
  _fun(new Fun[R]) { def fun[R[\_\_]:Ops] {  
    implicit ev0: Convert[R,RF] = {  
      (kv: R[Value]) ⇒ (val kv = ev0:convert(kv0))  
      eval_body[RF](kv)  
    })  
  }  
 /*@ ... */  

Here is the lifting of the continuation, which delegates to a
regular lifted application:

def _lift(k: ⇒ R[Value]): Value = {  
  val o = implicitly[Ops[R]]  
  o._lift(new FunC[R] { def fun[R[\_\_]:Ops] {  
    implicit ev: Convert[R1,R1] = (  
      (v ⇒  
        apply_lifted_cont[R1](ev:convert(k), v)  
      ))  
    })  
  apply_lifted_cont[R](cont, R[Value], v: R[Value]) = {  
    val o = implicitly[Ops[R]]; import o._app(cont, _cons(v, N), id_cont[R])  
  }  

In the generated code, we want to delegate the conversion to
the Scala compiler. So we generate principled implicit conversions
between any mode W from an outer context, and a mode R in an inner
context of W.

3.2 Meta-Applying Application

Application requires some “unlifting” to fit into our setting, be-
cause the result of evaluation is dynamic, so both the evaluated
function and arguments are dynamic values. The default implementa-
tion for base-apply delegates to a lifted operation _app, so that
generated code contains code for _app calls by default.

def eval_application[R[\_\_]:Ops](m: MEnv,  
  exp: Value, env: Value, cont: Value) = {  
    val o = implicitly[Ops[R]]; import o._app(cont, _apply(fun, args))  
  
  meta_apply[R](m, S("base-apply"), fun, env, _cont(  
    new Func[R] { def fun[R[\_\_]:Ops] {  
      implicit ev: Convert[R1,R1] = (  
        val o1 = implicitly[Ops[R1]]  
        meta_apply[R1](m, S("eval-list"), args, env, o1._cont(  
          new Func[R1] { def fun[R2[\_\_]:Ops] {  
            implicit ev12: Convert[R1,R2] = (  
              val o2 = implicitly[Ops[R2]]  
              meta_apply[R2](m, S("base-apply"),  
                cons(o2._unlift(ev12:convert(v)), o2._unlift(vs)),  
                env, cont)  
          )  
        )  
      )  
    )  
  )  
  )  
})  
})  
  
  base_apply[R[\_\_]:Ops](m: MEnv, fun: R[Value], args: R[Value],  
    env: Value, cont: Value) = {  
    val o = implicitly[Ops[R]]  
    o._app(fun, args, cont)  
}  

In Ops[NoRep]:

def _app(fun: Value, args: Value, cont: Value) =  
  static_apply[NoRep](fun, args, cont)

For application, we _unlift then re_-lift the function and ar-
arguments in order to package them as expected through the
various functions. For Ops[NoRep], these operations are just iden-
tityes, so there is nothing to worry about. However, for Ops[Rep],
we have to ensure that we don’t over-do the wrappings, so that
_lift(_unlift(v)) = v.

3.3 Optimizations

We use smart constructors from LMS to perform some optimiza-
tions. For example, _lift and _unlift pattern-match on their
argument to avoid over-wrapping. In addition, we inline applications
where function and arguments are more or less known. We also
discard cells that are locally allocated, and then only read – this requires some extra book-keeping too.

For _lift, we can simply inspect whether the Value passed in is already some unlifted code. For _unlift and _app, we delegate to smart constructors that know about the implementation layer of LMS.

```scala
def _lift(v: Value) = v match {
  case Code(r: Rep[Value]) => r
  case _ => unit(v)
}

def _unlift(v: Rep[Value]) = unlift_rep(v)
def _app(f: Rep[Value], args: Rep[Value], cont: Value) = app_rep(f, args, cont)
```

For _app, the cases are more involved, and more ad-hoc too. They influence what semantics can be considered fixed at compilation. Here, we consider primitives, and other compiled (or built-in – no difference in representation) functions fair game. By considering functions that are read from a cell, we enable more inlining: for example, inlining the old-eval-var call from the modified eval-var interpreter function. Because the function meta_appl apply already silently reads interpreter function cells, this is a natural next step, that doesn’t fundamentally alter the model.

```scala
def app_rep(f: Rep[Value], args: Rep[Value], cont: Value): Rep[Value] = (f, args) match {
  case Const(fprim@Prim(p), Const(vs@P(_, _))) =>
    if !effectful_primitives.contains(fprim) && hasCode(vs) =>
      apply_cont[Rep](cont, OpsRep._lift(r))
    case Def(Reflect(CellReadRep(Const(Cell(InCell(
      Evalfun(ekey)))), _, _)), Const(vs@P(_, _))) =>
      val efn = funs(ekey).fun[Rep]
      efn(p(cont, vs))
    case (Const(Evalfun(ekey)), Const(vs@P(_, _))) =>
      val efn = funs(ekey).fun[Rep]
      efn(p(cont, vs))
    case (Const(fcont), Const(P(a, N))) if isCont(fcont) =>
      apply_cont[Rep](fcont, OpsRep._lift(at))
    case (Const(fcont), Def(ConsRep(a, Const(N)))) if isCont(fcont) =>
      apply_cont[Rep](fcont, a)
    case (_, _) =>
      val x = fresh[Value]
      val y = reifyEffects{
        apply_cont[Rep](cont, x)
      }
      reflectEffect(AppRep(f, args, x, y))
  }
```

### 3.4 Effects

Mutation is explicit at the host level through cells. This is not exposed in the user language, but it is exposed in the compilation language. These cell operations (allocations, writes, reads) are marked as effectful by the LMS framework. Similarly, the display primitive is marked effectful so that we don’t display at code generation time, but defer to generating code.

### 3.5 Values in Text

With LMS, the generated code is then printed as text, which a regular compiler (like Scala’s) can then load in. Hence, it is essential that the textual representation of values such as cells is semantic-preserving. Hence, a cell in the generated code must reference some dynamic cell in the running system.

Here is how values are represented in the host language. The overriding of toString is to ensure that the textual representation in generated code is syntactically valid Scala that can be meaningfully re-incorporated.

```scala
sealed trait Value
  case class I(n: Int) extends Value
  case class B(b: Boolean) extends Value
  case class S(sym: String) extends Value {
    override def toString = "S\\"+sym+\\"
  }
  case class Str(s: String) extends Value {
    override def toString = "Str\\"+s+\\"
  }
  case class N extends Value
  case class P(car: Value, cdr: Value) extends Value
  case class Prim(p: String) extends Value {
    override def toString = "Prim\\"+p+\\"
  }
  case class Clo(params: Value, body: Value, env: Value, menu: MEnv) extends Value
  case class Evalfun(key: Int) extends Value
  case class Code[R[_]](c: R[Value]) extends Value
  case class Cont(key: Int) extends Value
  case class CodeCont[R[_]](f: FunC[R], thunk: () => R[Value]) extends Value {
    lazy val force: R[Value] = thunk()
  }
  case class Cell(key: String) extends Value {
    override def toString = "Cell\\"+key+\\"
  }
```

### 3.6 Compile-Time Continuations

At compile time, we typically only need the continuation to generate code following an _app call. We don’t really need a first-class representation of the continuation until it is used as such, escaping the context of an _app call.

```scala
def apply_cont[R[_]:Ops](cont: Value, v: R[Value]): R[Value] = cont2fun[R](cont) match {
  case Some(f) => f(v)
  case None =>
    val o = implicitly[Ops[R]]
    o._static_apply[R](cont, P_unlift(v), N, id_cont[R])
}
```

Still, a continuation should be a Value by our staging discipline, so what do we do? We create a continuation code that can be forced to a R[Value] only when actually used in a first-class position.

```scala
def apply_cont[R[_]:Ops](cont: Value, v: R[Value]): R[Value] = cont2fun[R](cont) match {
  case Some(f) => f(v)
  case None =>
    val o = implicitly[Ops[R]]
    o._static_apply[R](cont, P_unlift(v), N, id_cont[R])
}
```

### 3.7 Partially Symbolic Environments

Environments introduce cells for enabling mutation via set!. When extending an environment, we create a new frame, in which we bind each parameter to an argument. The args value might be symbolic, wrapped as a Code.

```scala
def env_extend[R[_]:Ops](env: Value, params: Value, args: Value) = {
  val o = implicitly[Ops[R]]
  val frame = make_pairs[R](params, args)
  cons(if (o.inRep) frame else cell_new(frame, "frame"), env)}
```

There is a cute trick (Danvy and Goldberg 2002) 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.
4.3 Reifiers

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

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

\[
\begin{align*}
\text{(define call/cc)} \\
\lambda(f) \\
\delta(e r k) \\
(k (\text{meaning} 'f r (\lambda v) v) k)))))
\end{align*}
\]

\[
\begin{align*}
> (+ 1 (\text{call/cc} (\lambda(k) 0))) & \quad ; ; 1 \\
> (+ 1 (\text{call/cc} (\lambda(k) (k 0))))) & \quad ; ; 1 \\
> (+ 1 (\text{call/cc} (\lambda(k) (\text{begin} (k 1) (k 3)))))) & \quad ; ; 2 \\
> (+ 1 (\text{call/cc} (\lambda(k) (\text{begin} (k 3) (k 5)))))) & \quad ; ; 2 \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{(define call/cc)} \\
\lambda(f) \\
\delta(e r k) \\
(k (\text{meaning} 'f r (\lambda v) v) k)))))
\end{align*}
\]

\[
\begin{align*}
> (+ 1 (\text{call/cc} (\lambda(k) 0))) & \quad ; ; 1 \\
> (+ 1 (\text{call/cc} (\lambda(k) (k 0))))) & \quad ; ; 1 \\
> (+ 1 (\text{call/cc} (\lambda(k) (\text{begin} (k 1) (k 3)))))) & \quad ; ; 2 \\
> (+ 1 (\text{call/cc} (\lambda(k) (\text{begin} (k 5) (k 3)))))) & \quad ; ; 2 \\
\end{align*}
\]

4.4 Meta-Level Undo

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

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:

\[
\begin{align*}
\text{(define undo-list '())} \\
\text{(define old-eval-set! eval-set!)} \\
\text{(set! eval-set! \(\lambda(e r k)\)}) \\
\text{(let ((name (car (cdr e)))))} \\
\text{(eval-var name \(\lambda(v)\) \(\text{eval-var} e r k\)))} \\
\text{(old-eval-set! e r k))))))))
\end{align*}
\]

\[
\begin{align*}
\text{(define reflect-undo!)} \\
\text{(lambda (r))} \\
\text{(if (null? undo-list) '())} \\
\text{(begin)} \\
\text{(old-eval-set! \(\lambda(v)\) \(\text{eval-set!} (\text{car undo-list})\))} \\
\text{(list 'quote cdr (car undo-list))))} \\
\text{(r \(\lambda(v)\)))} \\
\text{(set! undo-list (cdr undo-list)))}) \\
\text{))}
\end{align*}
\]

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

\[
\begin{align*}
\text{(begin)} \\
\text{(define undo! (\lambda() \(\text{(define reflect-undo!)} (\_env))))}
\end{align*}
\]

4.5 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.

\[
\begin{align*}
\text{(define apply-delta)} \\
\lambda(e r k) \\
\text{(let ((operator (car e)) \(\text{operand} cdr e\)))} \\
\text{(let ((delta-params (car (cdr operator)))))} \\
\text{(delta-body (cdr (cdr operator))))} \\
\text{(eval-begin)} \\
\text{delta-body} \\
\text{(extend _env delta-params (list operand r k))} \\
\text{id-cont))))))))
\end{align*}
\]

\[
\begin{align*}
\text{(define old-eval-application eval-application)} \\
\text{(set! eval-application \(\lambda(e r k)\)}) \\
\text{(if (delta? e) \(\text{apply-delta} e r k\) \(\text{old-eval-application} e r k\)\)))}) \\
\text{(define meaning base-eval) \\
\text{))}
\end{align*}
\]

4.5.1 Auto-Continue

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

\[
\begin{align*}
\text{(define undo-list '())} \\
\text{(define old-eval-set! eval-set!)} \\
\text{(set! eval-set! \(\lambda(e r k)\)}) \\
\text{(let ((name (car (cdr e)))))} \\
\text{(eval-var name \(\lambda(v)\) \(\text{eval-var} e r k\)))} \\
\text{(old-eval-set! e r k))))))))
\end{align*}
\]

\[
\begin{align*}
\text{(define reflect-undo!)} \\
\text{(lambda (r))} \\
\text{(if (null? undo-list) '())} \\
\text{(begin)} \\
\text{(old-eval-set! \(\lambda(v)\) \(\text{eval-set!} (\text{car undo-list})\))} \\
\text{(list 'quote cdr (car undo-list))))} \\
\text{(r \(\lambda(v)\)))} \\
\text{(set! undo-list (cdr undo-list)))}) \\
\text{))}
\end{align*}
\]

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

\[
\begin{align*}
\text{(define undo! (\lambda() \(\text{(define reflect-undo!)} (\_env))))}
\end{align*}
\]

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

4.5.2 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.

(EXIT (begin
  (define pending-thanks '())

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

At the user-level, we provide a convenient function to resume! from the pending thunks of the meta-meta-level.

(define resume! (lambda ()
  (if (null? EM (EM (EM (EM (let ((thank (car pending-thanks))
        (set! pending-thunks (cdr pending-thunks))
        (thank)))))))))

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 // id_cont :(
> (resume!) ;; => done

4.6 Collapsing User-Level Interpreters

There’s nothing special about meta-level collapsing: we can do it entirely at the user-level too. For example, we can define a user-level interpreter for a little language and use the interpreter as a compiler, since any compiled function is stage-polymorphic.

As a tiny example, here is a generic list matcher that ensures that 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))))))

Now, we can generate code for matching a particular prefix:

(define start_ab ((lambda () (matches 'a b)]))

The code generated for the lambda of start_ab just has 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. Conceptually, the generated code is equivalent to the one generated for the following user-level program:

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

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. We can also generate code that will then just have extra counter increments like in the initial example of section 1.1.

> ((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)

5. Perspective

In this work, we showed that it is possible to compile a function in a tower of interpreters, under current, possibly user-modified, semantics. We took several 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 decision too. Because the body 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:

- What about compilation in a “classical” tower of interpreters? Shifting levels up and down would need to be made explicit (and 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.

```
(begin (define where_am_i 'user)
  (DM (define where_am_i 'meta))
  (DM (let ((old-eval-var eval-var)
             (_k (lambda (x) x)))
       (set! eval-var (lambda (e r k)
                      (if (eq? e '__k) (k __k)
                           (begin
                            (if (eq? e '_) (set! _k k)
                             '())
                           (old-eval-var e r k))))))

(define _ 0)
> (+ _ 1) ;; => 1
> where_am_i ;; => user
> (DM where_am_i) ;; => meta
> (_k 2) ;; => 3
> where_am_i ;; => user
> (DM where_am_i) ;; => meta (in Black: user!)
```

It’s strange (counterintuitive) that calling the continuation _k pushed another user-level in Black.

Some of our examples could be implemented more seamlessly because we didn’t need to reason about inadvertently shifting or pushing levels.

• 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?

• The use of Code is not very principled. Is there a better more principled way to mix dynamic parts in otherwise mostly static values?

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

By showing that it’s 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.

References


http://github.com/namin/lms-black