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.

```
(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.

```
(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
```

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.

```
(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.

```
{(k, xs) =>
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_1 =>
_cell_set(<cell counter>, c_1)
_app(‘<, _cons(_car(xs), ‘(2)), _cont{v_1 =>
_if(_true(v_1),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_2 =>
_cell_set(<cell counter>, c_2)
_app(k, _cons(_car(xs), ‘()), _cont{v_2 =>
v_2})}),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_3 =>
```

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

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

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-recursive interpreter functions, but look up such references in the meta-environment (via `meta_apply`).

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 initialization. The meta-environment constructor takes an environment eagerly, and the next meta-environment lazily; thus, the stream of environments fixes infinitely many upper levels, potentially.

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

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

Finally, when we shift to the meta-level, the continuation from the lower-level becomes an argument to the meta-level function, so we need a new outer continuation – usually, it’s a trivial continuation because there’s nothing to do after the shift. The `meta_apply` function delegates to `static_apply` to actually apply the interpreter function looked up in the meta-environment. If we have a closure (from an uncompiled `lambda`), then we recursively call `meta_apply` again possibly going further up in the levels. If we have a compiled or built-in function, then we just apply it as a black-box, thus breaking out of an infinite-tower regress.

```
def meta_apply[R[_]:Ops](m: MEnv, s: Value,
  exp: Value, env: Value, cont: Value): R[Value] = {
  val MEnv(meta_env, meta_menv) = m
  val o = implicitly[Ops[R]]; import o._
  val fun = env_get(meta_env, s) match {
    case c@Cell(_) => 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
  }
}
```

Before we explain the `R` markings next, let us give 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 e = exp match {
    case P(_, P(e, N)) => e
  }
  val MEnv(meta_env, meta_menv) = m
  meta_apply[R](meta_menv, S("base-eval"), e, meta_env, cont)
}
```

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

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 (not 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`).

```
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 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.

```
def eval_var[R[_]:Ops](m: MEnv,
  exp: Value, env: Value, cont: Value): R[Value] = {
  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 v => 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.

```
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.

```
def eval_clambda[R[_]:Ops](m: MEnv,
  exp: Value, env: Value, cont: Value): R[Value] = {
  val o = implicitly[Ops[R]]; import o._
  val (params, body) = exp match {
    case P(_, P(params, body)) => (params, body)
  }
  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)),
      unlift_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))
  } else {
    /* already in compilation mode */
    /* need to create a poly-stage function */
    /* ... */
  }
  apply_cont[R](cont, f)
}
```

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, once 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]) = v
}
def convertTrans[R1[_],R2[_],R3[_]](
  implicit ev12: Convert[R1, R2], ev23: Convert[R2,R3]) =
  new Convert[R1,R3] {
  def convert(v: R1[Value]) = ev23.convert(ev12.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-polymorphic, given evidence that we can convert from `W` to `R`.

```
abstract class Fun[W[_]:Ops] {
  def fun[R[_]:Ops](implicit ev: Convert[W,R]): W[Value] => R[Value]
}
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 unfortunately 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`; the others are similar:

```
def base_eval_fun(m: => MEnv): Fun[NoRep] = new Fun[NoRep] {
  def fun[R[_]:Ops](implicit ev: Convert[NoRep,R]) =
  { (vc: Value) =>
    val P(mcont, P(exp, P(env, P(cont, N)))) = vc
    apply_cont[R](mcont, base_eval[R](m, exp, 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[RF[_]:Ops](
      implicit ev0: Convert[R,RF]) =
      { ((kv0: R[Value]) => { val kv = ev0.convert(kv0)
        eval_body[RF](kv)
      })}})
  }
/* ... */
```

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

```
def unlift_cont[R[_]:Ops](k: => R[Value]): Value = {
  val o = implicitly[Ops[R]]
  o._cont(new FunC[R] {def fun[R1[_]:Ops](
    implicit ev: Convert[R,R1]) =
    { v =>
      apply_lifted_cont[R1](ev.convert(k), v)
    }})
}
def apply_lifted_cont[R[_]:Ops](cont: R[Value], v: R[Value]):
  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, because the result of evaluation is dynamic, so both the evaluated function and arguments are dynamic values. The default implementation 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._
  val P(fun, args) = exp
  meta_apply[R](m, S("base-eval"), fun, env, _cont(
    new FunC[R] { def fun[R1[_]:Ops](
    implicit ev: Convert[R,R1]) = { v =>
    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]) = { vs =>
      val o2 = implicitly[Ops[R2]]
      meta_apply[R2](m, S("base-apply"),
        cons(o2._unlift(ev12.convert(v)), o2._unlift(vs)),
        env, cont)
    }}))
  }}))
}
def base_apply_fun(m: => MEnv): Fun[NoRep] = new Fun[NoRep] {
  def fun[R[_]:Ops](implicit ev: Convert[NoRep,R]) =
  { (vc: Value) =>
    val o = implicitly[Ops[R]]
    val P(mcont, P(P(fun, args), P(env, P(cont, N)))) = vc
    apply_cont[R](mcont, base_apply[R](m,
    o._lift(fun), o._lift(args), env, cont))
  }
}
def 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 arguments in order to package them as expected through the various functions. For `Ops[NoRep]`, these operations are just identities, 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 optimizations. 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.

```
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 unlift_rep, we pattern-match on the internal representation of a Rep, which is either a constant Const or a symbol Sym, with an associated definition.

```
def unlift_rep(r: Rep[Value]) = r match {
  case Const(v: Value) => v
  case _ => Code(r)
}
```

For app_rep, 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 also 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_apply already silently reads interpreter function cells, this is a natural next step, that doesn't fundamentally alter the model.

```
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) =>
    val r = apply_primitive(p, 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(a))
  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.

```
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 object 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,
  menv: 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.

```
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]]; import 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.

```
case class CodeCont[R[_]](f: FunC[R], thunk: () => R[Value])
      extends Value {
  lazy val force: R[Value] = thunk()
}
```

In Ops[Rep]:

```
def _cont(f: FunC[Rep]) = {
  lazy val k: CodeCont[Rep] = CodeCont[Rep](f, () => cont_rep(k, f))
  k
}
```

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.

```
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)
}
def make_pairs[R[_]:Ops](ks: Value, vs: Value):
```

```
  Value = (ks, vs) match {
  case (N, N) => N
  case (N, Code(_)) => N
  case (S(s), _) => cons(cons(ks, vs), N)
  case (P(k@S(s), ks), P(v, vs)) =>
    val o = implicitly[Ops[R]]
    cons(cons(k, o._unlift(o._cell_new(o._lift(v), s))),
      make_pairs[R](ks, vs))
  case (P(k@S(s), ks), Code(c : R[Value])) =>
    val o = implicitly[Ops[R]]
    cons(cons(k, o._unlift((o._cell_new(o._car(c), s)))),
      make_pairs[R](ks, Code(o._cdr(c))))
}
```

3.8 Three Languages

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 `Ops[R]`.

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

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

4. 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.

4.1 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)))))))
)
```

4.2 Stack Inspection

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`.

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

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

```
(EM (begin
(define eval-taba-call
  (lambda (add! original-eval-application)
    (lambda (exp env cont)
      (eval-list
       (cdr exp) env
       (lambda (ans-args)
         (original-eval-application
          exp env
          (lambda (ans)
            (add! ans-args ans)
            (cont ans))))))))

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

(add-app-hook!
 'taba
 (lambda (exp env cont)
   ((eval-taba (car (cdr exp))) (car (cdr (cdr exp))) env cont)))
 ))
```

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

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

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

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) 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)
))
```

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:

```
(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)))
)
```

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.

```
(EM (begin
(define old-eval-var eval-var)
(set! eval-var (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-meta level, we can explore alternative semantics for how continuations at the meta-level should behave.

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:

```
(EM (EM (begin
(define old-base-apply base-apply)
(set! base-apply (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
```

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.

```
(EM (EM (begin
(define pending-thunks '())

(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 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
```

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 (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
```

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

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

The code generated for the clambda 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 (clambda (s)
  (if (null? s) #f
    (if (eq? 'a (car s))
     ((clambda (s)
       (if (null? s) #f
         (if (eq? 'b (car s))
           ((clamda (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 "classic" 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)
(EM (define where_am_i 'meta))
(EM (let ((old-eval-var eval-var)
          (__k (lambda (x) x)))
      (set! eval-var (lambda (e r k)
                       (if (eq? e '__k) (k __k)
                           (begin
                                  (if (eq? e '_) (set! __k k)
                                      '())
                              (old-eval-var e r k)))))))
(define _ 0))
> (+ _ 1)         ;; => 1
> where_am_i      ;; => user
> (EM where_am_i) ;; => meta
> (__k 2)         ;; => 3
> where_am_i      ;; => user
> (EM where_am_i) ;; => meta (in Black: user!)
```

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

K. Asai. Compiling a reflective language using metaocaml. In *GPCE*, 2014.

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

C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, 1993.

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

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

Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 1999.

G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *GPCE*, 2013.

T. Rompf and N. Amin. Functional pearl: A SQL to C compiler in 500 lines of code. In *ICFP*, 2015.

T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *CACM*, 2012.

B. C. Smith. Reflection and semantics in lisp. In *POPL*, 1984.

M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Lisp and Functional Programming*, 1986.