Intensions and Extensions in a Reflective Tower

Olivier DANVY[*]& *Karoline MALMKJÆR*
DIKU – University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, DENMARK
uucp: danvy@diku.dk & karoline@diku.dk

Abstract

This article presents a model of the reflective tower based on the formal semantics of its levels. They are related extensionally by their mutual interpretation and intensionally by reification and reflection.

The key points obtained here are: a formal relation between the semantic domains of each level; a formal identification of reification and reflection; the visualisation of intensional snapshots of a tower of interpreters; a formal justification and a generalization of Brown's meta-continuation; a (structural) denotational semantics for a compositional subset of the model; the distinction between making continuations jumpy and pushy; the discovery of the tail-reflection property; and a Scheme implementation of a properly tail-reflective and single-threaded reflective tower.

Section 1 presents the new approach taken here: rather than implementing reification and reflection leading to a tower, we consider an infinite tower described by the semantics of each level and relate these by reification and reflection. Meta-circularity then gives sufficient conditions for implementing it. Section 2 investigates some aspects of the environments and control in a reflective tower. An analog of the *funarg* problem is pointed out, in relation with the correct environment at reification time. Jumpy and pushy continuations are contrasted, and the notions of ephemeral level and proper tail-reflection are introduced. Our approach is compared with related work and after a conclusion, some issues are proposed.

Keywords
Procedural reflection, reification, meta-continuation, Blond, tail-reflection, single-threadedness, jumpy and pushy continuations

Introduction

For a number of reasons reflective towers [Smith 82] [Smith 84] have not led to a general understanding of what they are, nor what to do with them and only to some extent [des Rivières & Smith 84] [Wand & Friedman 88] how to implement them. This article attempts to contribute to the field by investigating the extensional and intensional aspects of a reflective tower. The results obtained are a non-reflective, more general and more precise description (that is: requiring less intuition) and an implementation running in Scheme [Rees & Clinger 86], a little bigger than the Brown reflective tower [Wand & Friedman 86] [Wand, Friedman & Duba 86], but richer and more consistent with respect to the formal description presented here.

The terms "extensional" and "intensional" are taken in their general sense, as in [Talcott 85], that is: the first addresses the abstract "what", in the usual mathematical sense, and the latter covers the constructive "how", that is the mechanisms fulfilling the extension.

The basic idea of a reflective tower is to have a series of interpreters[1] interpreting each other, and connected by two meta-level operations: "reification" and "reflection" [Friedman & Wand 84]. Each interpreter processes the one below, and the tower is run by an "ultimate machine" [Smith 84] at its top:

```
 .         .
 .         .
+---------+         △
|    2    |    Reification
+---------+
|    1    |    Reflection
+---------+         ▽
|    0    |
+---------+
```

Reification makes the current state of the computation (expression to be interpreted, environment and continuation) available. It is achieved with procedures whose bodies are processed one level higher than their application: *reifiers*. Conversely, reflection spawns and

[*]LITP – Université de Paris 6 (45-55, 2e ét.), 4 place Jussieu, 75252 Paris Cedex 05, FRANCE (...!mcvax!inria!litp!od)

[1]For now, we relax the constraint that they are meta-circular.

activates a new level down in the tower, given the state in which the computation starts.

We propose not to focus immediately on the meta-level operations, but first to consider only one level. Denotational semantics provides a convenient framework to describe the language of this level[2]. To distinguish the descriptions of different levels, we index each semantic domain with its level in the tower. This gives the formal framework to identify and study the meta-level connections. We then establish the relation between all the semantic descriptions:

$$\mathcal{E}_n[\![int_n]\!]\rho_n\kappa_n \simeq \mathcal{E}_{n-1}$$

This isomorphism expresses that each level is created by an interpreter at the level above. It is general and does not assume the interpreter to be meta-circular.

With this extensional study, and constraining the interpreters to all interpret the same language[3] (that is, to be meta-circular) we get a simple intensional description using a meta-continuation pairing the environments and continuations of all the levels above. This, in turn, proves well-suited for implementing the tower. *En passant*, this approach justifies some design choices of Brown and in particular its single-threadedness – that is, there is one unique interpreter active at any time and thus no order-of-magnitude overhead every time a level is spawned. In addition, our implementation is more consistent than Brown since, for example, reification occurs in a correct environment, without any variable capture. This study does not use reflection to explain reflection and neither does our implementation.

We call our system *Blond*[4]. Its concrete syntax resembles Scheme, with numbers, strings, identifiers, λ-abstractions and reifiers (represented as γ and δ-abstractions): the idea of a γ-reifier is that it extends the environment at the level above its definition, whereas a δ-reifier extends the environment at the level above its application. Blond is applicative, call by value, lexically scoped and reflective. It is thus higher-order and offers full-fledged continuations. It has no special forms nor keywords, since each identifier can be (re-)bound and since most of the predefined control structures can be redefined reflectively (`quote`, `if`, *etc.*) – the others are the rock bottom semantic objects implementing the tower (`meaning`, `openloop`, *etc.*).

[2]The notations from [Schmidt 86] will be used.

[3]But this does not have to be so: [Sturdy 88] describes a reflective architecture, *Platypus*, in which the languages are distinct at each level.

[4]This is a bit of a joke and a wink at Brown, as Blond has been built in Scandinavia.

Blond is largely comparable to Brown, but differs on several points:
— the operator `make-reifier` has disappeared[5], for reasons developed in section 2.1, and reifiers are represented as ternary γ and δ-abstractions;
— numbers need not be quoted[6];
— each level has its own environment;
— the actual representation of the reflective tower (the meta-continuation in Brown) holds explicitly a current continuation and a current environment; this is required for a correct implementation of reification, as pointed out in section 2.1;
— when opening a new top level loop one can specify its environment;
— a Blond prompt displays both the current level in the tower and the current number of iterations in the top level loop[7]. The latter has proven very useful to visualize a point of reactivation:

```
>>> (blond)
0-0: bottom-level
0-1> (load "cwce.bl")
call-with-current-environment exit
0-1: loaded
0-2> (call-with-current-environment
         (lambda (r)
             (openloop "foo" r)))
foo-0: bottom-level
foo-1> (exit "bar")
0-2: bar
0-3> (exit "bye")
1-0: bye
1-1>
```

This illustrates a first scenario with Blond. The tower is started up from Scheme. The bottom level is labelled 0, below an infinity of others, labelled 1, 2, 3, *etc.*. At the iteration 1 of level 0, the file `"cwce.bl"` is loaded (in verbose mode). Two Blond reifiers are defined: one captures environments and the other exits the current level. At the iteration 2 of level 0, a new level labelled *foo* is spawned with the same environment as level 0. In that environment the identifier `exit` is bound. The only thing we do at the level *foo* is to exit from it. We are back to finish iteration 2 of level 0. During the iteration 3, we exit from level 0 to arrive at level 1. Exiting further would lead to levels 2, 3, 4, and so on.

[5]In Brown, reifiers are made by applying `make-reifier` to a ternary procedure.

[6]In the Brown expression `(add1 '3)`, `add1` designates a Brown primitive which designates the successor function; `'3` designates the same as `3` in Scheme, whereas `3` does not designate anything in Brown. In the Blond expression `(add1 3)`, `add1` still designates a Blond primitive function; `3` designates an internal structure which designates the number 3.

[7]This is analogous to a prompt in the command interpreter of an operating system, displaying the name of the host machine and the number of the next command in the session history.

What we get with Blond is a tower simulator processing one program point at any time, at the edge where the program stands. In that respect, control is single-threaded in Blond.

What we lose is any side effect above in the tower. This is easy to understand intuitively too, because the intuitive explanation of why the levels above do not need to be really processed is that they all perform operations on identical and well-known data (the interpreter at the level below). If one interpreter also performs side effects (such as a trace, *etc.*), these are neither well-known nor identical to anything else. They are lost in a single-threaded implementation.

The *effect* of a level is to process the level below, until the edge. In other words: even knowing only how the level below is behaving, one knows how and what the level above is doing. Of course, this only holds because the code of the level above is known, and so on.

This statement does not hold for *side effects* at the level above but only for its known effects. For example, let us consider one level displaying information about its own computation. This could be tuned by swapping the tracing level with the level above it (or below it) in the tower – the resulting trace varying by orders of magnitude (one for each level of interpretation). This cannot be done in Blond, because of its single-threadedness, even though we can illustrate how to swap levels.

For example let us permute the two levels above the current one. The idea is to reify three times and then to reflect back, exchanging environments and continuations and restoring the initial ones. To express it graphically:

```
.         .                     .         .
.         .                     .         .
+---------+                     +---------+
|    3    |                     |    3    |
+---------+                     +---------+
|    2    |                     |    1    |
+---------+  --permute!->       +---------+
|    1    |                     |    2    |
+---------+                     +---------+
|    0    |                     |    0    |
+---------+                     +---------+
```

The corresponding code figures in appendix 1, and the following scenario illustrates its effect. In an environment where `permute!` is bound to the permuter and `exit` exits the current level, we get:

```
>>> (blond)
0-0: bottom-level
0-1> (load "swap-2.bl")
permute! exit
0-1: loaded
0-2> (permute!)
0-2: done
0-3> (exit "bye")
2-0: bye
2-1> (begin (mute-load "exit.bl") (exit "ibid."))
1-0: ibid.
1-1> (begin (mute-load "exit.bl") (exit "again"))
3-0: again
3-1> (begin (mute-load "exit.bl") (exit "more"))
4-0: more
4-1>
```

Above level 0 are the two levels 1 and 2. Their permutation is witnessed by exiting from the level 0 to the level 2 and from the level 2 to the level 1, below the levels 3, 4, and so on. To conclude this example, let us point out that conceptually, what was syntax at one level has become semantics at the level above, and what was semantics at that level has become syntax at the level below.

Finally let us take a closer look to the meta-level connections: we already know how to spawn a new level with `openloop`. The general way to reflect is the function `meaning`, where one specifies an expression to be evaluated, an environment that has been reified and any function to be applied to the result:

```
>>> (blond)
0-0: bottom-level
0-1> (meaning 100 (reify-new-environment)
                  (lambda (a) a))
0-1: 100
0-2> (meaning 100 (reify-new-environment) add1)
0-2: 101
0-3>
```

The expression `100` is evaluated one level below and the result is passed to the function specified as a continuation. Since this function is specified at level 0, the result is transmitted at level 0. As we can see, `openloop` is a particular instance of `meaning` where the continuation loops and the expressions are read. The function `reify-new-environment` returns an initial environment that has just been reified.

Reification is performed by applying a reifier. A reifier has three parameters. The first is bound to the list of all its arguments (unevaluated), the second to a reified version of the current environment, and the third to a reified version of the current continuation. The body of the reifier is processed at the level above the current one:

```
>>> (blond)
0-0: bottom-level
0-1> (define baz (delta (e r k) e))
0-1: baz
0-2> (baz x y z)
1-0: (x y z)
1-1>
```

What we can do with a reified expression is to treat it as any denotable value. We can also convert it back to a syntactic object and evaluate it:

```
1-1> ((delta (e r k)
        (map add1 e)) 0 1 2 3)
2-0: (1 2 3 4)
2-1> ((delta (e r k)
        (meaning (car e) r k)) (map list '(1 2)))
2-1: ((1) (2))
2-2>
```

The latter reifier is the usual reflective definition of the identity function.

What we can do with a reified environment is to consult it. It is a function mapping identifiers to their value, if any. We can extend it too, out of a list of identifiers and a parallel list of values:

```
2-2> (let ((x "anything"))
        ((delta (e r k) (r 'x))))
3-0: anything
3-1> (define env
      (extend-reified-environment
       '(x y z) '(1 2 3) (reify-new-environment)))
3-1: env
3-2> (env 'y)
3-2: 2
3-3>
```

What we can do with a reified continuation is to apply it, since it is reified as an applicable object. Applying it, by default, replaces the current continuation, as in Scheme. To that respect the definition:

```
(define call-with-current-continuation
    (lambda (f)
        ((delta (e r k)
            (k ((r 'f) k))))))
```

is unsatisfactory since it would lose levels. The following definition is correct and summarizes what we know so far:

```
(define call-with-current-continuation
   (lambda (f)
      ((delta (e r k)
          (let ((env (extend-reified-environment
                        '(g c) (list (r 'f) k) r)))
              (meaning '(g c) env k))))))
```

Numerous other examples can be found in [Danvy & Malmkjær 88] and in the rest of this paper, which is organized as follows: section 1 develops on the semantics of each level in a reflective tower and their meta-level connection. This leads to design a reflective architecture for Blond. Section 2 investigates some aspects of environments and control in Blond. Section 3 compares the present approach with related work.

1 Extensional and intensional aspects of reflection

Let us first consider a tower of (possibly identical) interpreters, each processing the level below, but without any meta-level facilities. The language at each level is an expression language comparable to Scheme but they are not necessarily identical. Each of them is described by a usual continuation semantics manipulating expression, environment, and continuation but without a store – that is without side effects. In order to distinguish the domains and valuation functions at each level n, we subscript them:

Abstract syntax:
  $\mathtt{B}, \mathtt{E}, \mathtt{F}, \mathtt{R}, \mathtt{K} \in \mathit{Expression}_n$
  $\mathtt{C} \in \mathit{Constant}_n$
  $\mathtt{I}, \mathtt{x}, \mathtt{y}, \mathtt{z} \ldots \in \mathit{Identifier}_n$
  $\mathtt{E} ::= \mathtt{C} \mid \mathtt{I} \mid (\mathtt{F}\ \mathtt{E}^*)$
  etc.
Domains:
  $a, b, c, v \in \mathit{DenotableValue}_n = \mathit{Identifier}_n + \mathit{Function}_n + \mathit{List}_n + \ldots$
  $\rho_n \in \mathit{Environment}_n = \mathit{Identifier}_n \rightarrow \mathit{DenotableValue}_n$
  $\kappa_n \in \mathit{Continuation}_n = \mathit{DenotableValue}_n \rightarrow \mathit{Answer}_n$
  etc.
Valuation functions:
  $\mathcal{E}_n : \mathit{Expression}_n \times \mathit{Environment}_n \times \mathit{Continuation}_n \rightarrow \mathit{Answer}_n$
  $\mathcal{E}_n[\![\mathtt{I}]\!]\rho_n\kappa_n = \kappa_n(\rho_n[\![\mathtt{I}]\!])$
  etc.

In such a tower, all the domains are distinct, but strongly related because each level is created by the level above. All the objects at one level are represented at the level above: expressions, environments, continuations, denotable values, and so on.

One way to obtain this is to represent an object as a compound structure with the object and some identification tag.

1.1 Digression

For example an element of the domain Number_n could be represented in a domain Pair_{n+1} at the level above, the first element of the pair being a tag mimicking the injection tags of the domain $\mathit{DenotableValue}_n$.

Thus the number 7 at level 0 (let us call it 7_0) would at level 1 be represented as $(\nu_0,\ 7_1)$ and, leaving aside how to represent pairs and tags at the level above the number, 7_0 would be represented at level 21 as:

$$(\nu_0, (\nu_1, \ldots (\nu_{20},\ 7_{21}))\ldots))$$

At the "top" of the tower, 7_0 would appear as:

$$(\nu_0, \ldots (\nu_n, (\nu_{n+1}, \ldots, \texttt{0000 0111}) \ldots)) \ldots)$$

This gives a pleasing intensional view of the whole tower even for the most elementary objects. [Danvy 87] presented an analogous intensional view holding only for primitive functions.

Another alternative is just to represent a number (and any other elementary object) as the corresponding object at the level above. For example the number 7 at level n is represented as the number 7 at level $n+1$ and thus at all levels. Then the subscripts can be avoided and for the sake of simplicity this is done in the following.

Being familiar with these representations of objects of one level in another has proven helpful to understand meta-level facilities. Having distinguished the levels, we are now in a better position to describe reification and reflection.

1.2 The semantics of going up and down

The point of having reification in a system is to have full access to the state of the computation. Since each level in the tower is described by a semantics manipulating expression, environment, and control, full access can here be understood as access to expression, environment, and control. Similarly reflection (with the function `meaning`) can specify a state from which computation will proceed, by specifying the expression, environment and continuation to be in effect.

In the reflective tower this is realized by letting reifiers and `meaning` operate on more than one level. This easily leads to confusion and it is this confusion we can avoid with our subscripted model.

Now the operation of a reifier, `(delta (e r k) B)`, can be explained as a relation between valuation functions from the semantics of two levels[8]. The reifier has a body `B` and three formal parameters `e`, `r` and `k` to be bound respectively to the reified expression, environment, and continuation of the level where it is applied.

$$\begin{aligned}&\mathcal{E}_n[\![\texttt{((delta (e r k) B) E}^*\texttt{)}]\!]\rho_n\kappa_n\\&= \mathcal{E}_{n+1}[\![\texttt{B}]\!]([[\![\texttt{e}]\!] \mapsto (map_n\ exp_n^\wedge[\![\texttt{E}^*]\!]),\\&\qquad\qquad [\![\texttt{r}]\!] \mapsto (env_n^\wedge\rho_n),\\&\qquad\qquad [\![\texttt{k}]\!] \mapsto (cont_n^\wedge\kappa_n)]\rho_{n+1})\ \kappa_{n+1}\end{aligned}$$

The reification relation

where ρ_{n+1} and κ_{n+1} are the environment and continuation of level $n+1$, $exp_n^\wedge$, $env_n^\wedge$ and $cont_n^\wedge$ are operations[9] of the following types:

$$\begin{aligned}&exp_n^\wedge : Expression_n \to DenotableValue_{n+1}\\&env_n^\wedge : Environment_n \to DenotableValue_{n+1}\\&cont_n^\wedge : Continuation_n \to DenotableValue_{n+1}\end{aligned}$$

and $(map_n f)$ maps a sequence of expressions into a list of values by applying f to each of the expressions. It has the following functionality:

$$\begin{gathered}(Expression_n \to DenotableValue_{n+1}) \to\\ Expression_n^* \to DenotableValue_{n+1}\end{gathered}$$

One can note that $exp_n^\wedge$ is not applied to `B`, the body of the reifier, since by hypothesis `B` is written in the language of level $n+1$.[10] This could be compared to an `asm` statement in a C program: the argument is written in assembly language, not in C nor in anything else.

Similarly reflection can be described by:

$$\begin{aligned}&\mathcal{E}_n[\![\texttt{(meaning E R K)}]\!]\rho_n\kappa_n =\\&\quad \mathcal{E}_n[\![\texttt{E}]\!]\rho_n(\lambda a.\mathcal{E}_n[\![\texttt{R}]\!]\rho_n\\&\qquad\qquad (\lambda b.\mathcal{E}_n[\![\texttt{K}]\!]\rho_n\\&\qquad\qquad\quad (\lambda c.\mathcal{E}_{n-1}(exp_n^\vee a)(env_n^\vee b)(cont_n^\vee c))))\end{aligned}$$

The reflection relation

where $exp_n^\vee$, $env_n^\vee$ and $cont_n^\vee$ have the types:

$$\begin{aligned}&exp_n^\vee : DenotableValue_n \to Expression_{n-1} \cup \{error\}\\&env_n^\vee : DenotableValue_n \to Environment_{n-1} \cup \{error\}\\&cont_n^\vee : DenotableValue_n \to Continuation_{n-1} \cup \{error\}\end{aligned}$$

Applying $exp_n^\vee$ to the argument a actually makes it a syntactic object at level $n-1$ so it is not necessary to enclose it in semantic brackets.

It is important to notice that this is not a denotational definition of `meaning` but a relation between two denotational semantics. If it was to be interpreted as a denotational definition it would clearly violate the compositionality principle.

This is at the crossroad of the criticisms against the non-compositionality of Lisp [Muchnick & Pleban 80], as a denotable value is mapped to an expression. For now, let us say that this criticism does not apply (yet), since here the operation $exp_n^\vee$ relates two distinct semantics.

About the $^\wedge$ and $^\vee$ operations it is not in general possible to say much except for their functionality. Since they are to support meta-level connections, however, they must satisfy these relations:

(1) $cont_{n+1}^\vee \circ cont_n^\wedge = identity_{Continuation_n}$
and
(2) $cont_n^\wedge \circ cont_{n+1}^\vee \sqsubseteq identity_{DenotableValue_{n+1}}$

[8] For simplicity we consider the definition directly rather than a reifier bound in the environment.

[9] Named compatibly with [Wand & Friedman 88].

[10] This assumption is of importance if the languages at the different levels are different, because `B` would have to be compiled by $exp_n^\wedge$ if it was written in the language of another level.

(1) should hold because when a previously reified object is reinvoked, we want it to be the same object (actually having a similar behaviour is sufficient in an implementation – but probably not the most efficient).

(2) states that $cont_n^{\wedge} \circ cont_{n+1}^{\vee}$ is less defined than $identity_{DenotableValue_{n+1}}$, *i.e.*, $(cont_n^{\wedge}(cont_{n+1}^{\vee} c))$ is either c or *error*. It is c when $(cont_{n+1}^{\vee} c)$ is defined because we only want one legal representation at level n for each object at level $n-1$. It is *error* when $(cont_{n+1}^{\vee} c)$ is *error* because we don't want to take something undefined to something defined.

Similar relations hold for the other operations: $exp_n^{\wedge}$, $exp_n^{\vee}$, $env_n^{\wedge}$ and $env_n^{\vee}$.

Though the reification and reflection relations are enlightening they do not take full advantage of the fact that this happens in a tower of interpreters interpreting each other. This makes it possible to integrate the loose ends: how to find the correct ρ_{n+1} and κ_{n+1} in the reification relation and how to find the correct $\mathcal{E}_{n-1}$ in the reflection equation.

What happens in the intuitive model when `meaning` is called is that the interpreter int_n (which is defined at level n, the currently lowest level of the tower) is invoked with some arguments `E`, `R` and `K`. Since int_n implements the language described by $\mathcal{E}_{n-1}$ we have:

$$\mathcal{E}_n [\![int_n]\!] \rho_n \kappa_n \simeq \mathcal{E}_{n-1}$$

Similarly, when a reifier is applied, ρ_{n+1} and κ_{n+1} are present in the valuation function $\mathcal{E}_n$. This isomorphism expresses that the meaning of an interpreter applied to an expression is the meaning of that expression. It corresponds to the fact that partially evaluating an interpreter with respect to a program leads to an equivalent residual program [Jones *et al.* 88].

This also makes sense in terms of domains since, in the model of the reflective tower, an interpreter at level n takes three $DenotableValue_n$-arguments – an expression, an environment, and a continuation at level $n-1$ – and returns an $Answer_n$. Thus it has the type:

$$Expression_{n-1} \times Environment_{n-1} \times Continuation_{n-1} \rightarrow Answer_n$$

which is the same type as $\mathcal{E}_{n-1}$ if we assume that all the *Answer* domains are equal.

To examine this relation a little further, let us assume that all the levels are identical and that the interpreter is a program, `<int`$_n$`>`, which does not use reification nor reflection. We can then evaluate a level $n-1$ expression[11], $[\![\texttt{E}]\!]_{n-1}$, at level n in a level n environment ρ_n binding `e`, `r` and `k` to the expression and structures representing the environment $\rho_{init_{n-1}}$ and the continuation $\kappa_{init_{n-1}}$ of level $n-1$.

$$\begin{aligned}
&\mathcal{E}_n [\![\texttt{(<int}_n\texttt{> e r k)}]\!]_n \rho_n \kappa_n \\
&= \mathcal{E}_n [\![\texttt{<int}_n\texttt{>}]\!]_n \rho_n (\lambda f.(f(exp^{\wedge} [\![\texttt{E}]\!]_{n-1})(env^{\wedge} \rho_{init_{n-1}}) \\
&\qquad\qquad (cont^{\wedge} \kappa_{init_{n-1}}) \kappa_n)) \\
&\simeq \mathcal{E}_{n-1} [\![\texttt{E}]\!]_{n-1} \rho_{init_{n-1}} \kappa_{init_{n-1}}
\end{aligned}$$

Since the levels are identical the $^{\wedge}$ and $^{\vee}$ functions at all levels are identical and it is no longer necessary to subscript them.

This describes, in some sense, the meaning of two levels in the semantics of the upper one. The meaning of an expression $[\![\texttt{E}]\!]$ at level $n-1$ can be found as

$$\mathcal{E}_n [\![\texttt{(<int}_n\texttt{> e r k)}]\!]_n ([[\![\texttt{e}]\!]_n \mapsto (exp^{\wedge} [\![\texttt{E}]\!]_{n-1})] \rho_n) \kappa_n$$

where κ_n is the initial continuation in the semantics. It is not a (structural) denotational semantics for level $n-1$, since the meaning of an expression is not a composition of the meanings of its subexpressions.

However it is possible to express the reflection relation in the semantics of one level assuming that `meaning` is bound to the denotation of the interpreter in the environment.

To reduce the compositionality problem somewhat (if not completely) we can choose to make the interpreter (*i.e.*, `meaning`) a special form in the language and introduce a second valuation function in the semantics, $\mathcal{E}_n^{-1}$. $\mathcal{E}_n^{-1}$ would take three arguments: a syntactic expression of level $n-1$ (which is where the compositionality problem remains) and two values to serve as level $n-1$ environment and continuation. However since $\mathcal{E}_n^{-1}$ is a valuation function in the level n semantics it also needs the continuation of level n as an argument. For reifiers it is also necessary to save the environment ρ_n (see section 2.1 for a closer discussion). Then we can define

$$\begin{aligned}
&MetaContinuation_{n-1} = \\
&\quad (Environment_n \times Continuation_n) \\
&\mathcal{E}_n^{-1} : Expression_n \times DenotableValue_n \times \\
&\qquad DenotableValue_n \times MetaContinuation_n \rightarrow \\
&\qquad Answer \\
&\mathcal{E}_n [\![\texttt{(meaning E R K)}]\!] \rho_n \kappa_n \\
&= \mathcal{E}_n [\![\texttt{E}]\!] \rho_n \\
&\qquad (\lambda a. \mathcal{E}_n [\![\texttt{R}]\!] \rho_n \\
&\qquad\quad (\lambda b. \mathcal{E}_n [\![\texttt{K}]\!] \rho_n \\
&\qquad\qquad (\lambda c. \mathcal{E}_n^{-1} (exp^{\vee} a)(env^{\vee} b)(cont^{\vee} c)(\rho_n, \kappa_n))))
\end{aligned}$$

It can be noticed that in this definition, the "continuation" of $\mathcal{E}_n^{-1}$ is a function at the level n, so it takes the continuation of level n as argument. Correspondingly a unary function at level n takes two arguments: the actual parameter and the continuation, while a unary

[11] Subscripting the semantic brackets for clarity.

function at level $n-1$ takes three arguments: the actual, the "continuation" of level 1 and the continuation of level n.

Now we can even describe reification at level $n-1$:

$$
\begin{aligned}
&\mathcal{E}_n^{-1}[\![\texttt{((delta (e r k) B) E}^*\texttt{)}]\!]\, r\, k\, (\rho_n, \kappa_n) \\
&= \mathcal{E}_n[\![\texttt{B}]\!]([[\![\texttt{e}]\!] \mapsto (\mathit{map}\ \mathit{exp}^\wedge [\![\texttt{E}^*]\!]), \\
&\qquad\quad [\![\texttt{r}]\!] \mapsto (\mathit{env}^\wedge r), \\
&\qquad\quad [\![\texttt{k}]\!] \mapsto (\mathit{cont}^\wedge k)]\rho_n)\kappa_n
\end{aligned}
$$

To describe the whole tower this way in the level n semantics we just need infinitely many valuation functions, all alike but operating on different domains and all taking this extra argument:

$$
\begin{aligned}
&\mathit{MetaContinuation}_{n-1} = \\
&\quad ((\mathit{Environment}_n \times \mathit{Continuation}_n) \times \\
&\qquad ((\mathit{Environment}_{n-1} \times \mathit{Continuation}_{n-1}) \times \ldots)) \\
&\mathcal{E}_n^m : \mathit{Expression}_{n+m} \times \mathit{Environment}_{n+m} \times \\
&\qquad \mathit{Continuation}_{n+m} \times \mathit{MetaContinuation}_{n+m} \rightarrow \\
&\qquad \mathit{Answer} \\
&\mathcal{E}_n^m[\![\texttt{(meaning E R K)}]\!]\rho_{n+m}\kappa_{n+m}\tau_{n+m} \\
&= \mathcal{E}_n^m[\![\texttt{E}]\!]\rho_{n+m}(\lambda a\tau_{n+m}. \\
&\qquad \mathcal{E}_n^m[\![\texttt{R}]\!]\rho_{n+m}(\lambda b\tau_{n+m}. \\
&\qquad\quad \mathcal{E}_n^m[\![\texttt{K}]\!]\rho_{n+m}(\lambda c\tau_{n+m}. \\
&\qquad\qquad \mathcal{E}_n^{m-1}(\mathit{exp}^\vee a)(\mathit{env}^\vee b)(\mathit{cont}^\vee c) \\
&\qquad\qquad\quad ((\rho_{n+m}, \kappa_{n+m}), \tau_{n+m}))\tau_{n+m})\tau_{n+m})\tau_{n+m}
\end{aligned}
$$

$\mathcal{E}_n^m$ expresses the meaning of level $n+m$ in the semantics of level n (m can be compared to an offset).

With an infinity of levels, the functions at all levels take infinitely many arguments. For simplicity that is rather written as three, of which the third is infinite: the actual, the continuation and the meta-continuation.

This means (since a continuation is a function at the level above) that a continuation rather than one argument takes two: a value and a meta-continuation. This again means that if we want to use a function at one level as a continuation at the level below (with `meaning`) $\mathit{cont}^\vee$ has to transform it from a 3-argument function to a 2-argument function. This is no problem since the three arguments the function should take are contained in the two which are available: continuation of level n and meta-continuation of level n are contained in the meta-continuation of level $n-1$.

Now the whole tower is described in one semantics. So far we have carefully maintained subscripts to clarify level shifts. But since all the levels are identical the subscripts were not necessary and simply removing them from the domains the *MetaContinuation* reveals to have the type $(\mathit{Continuation} \times \mathit{Environment}) \times \mathit{MetaContinuation}$, *i.e.*, all the superscripted valuation functions have the same type - and are thus identical. So we can remove the super- and subscripts giving:

$$
\begin{aligned}
&\mathit{MetaContinuation} = \\
&\quad (\mathit{Environment} \times \mathit{Continuation}) \times \mathit{MetaContinuation} \\
&\mathcal{E} : \mathit{Expression} \times \mathit{Environment} \times \mathit{Continuation} \times \\
&\quad \mathit{MetaContinuation} \rightarrow \mathit{Answer} \\
&\mathcal{E}[\![\texttt{(meaning E R K)}]\!]\,\rho\,\kappa\,\tau \\
&= \mathcal{E}[\![\texttt{E}]\!]\rho(\lambda a\tau.\mathcal{E}[\![\texttt{R}]\!]\rho(\lambda b\tau.\mathcal{E}[\![\texttt{K}]\!]\rho(\lambda c\tau. \\
&\quad \mathcal{E}(\mathit{exp}^\vee a)(\mathit{env}^\vee b)(\mathit{cont}^\vee c)((\rho, \kappa), \tau))\tau)\tau)\tau \\
&\mathcal{E}[\![\texttt{((delta (e r k) B) E}^*\texttt{)}]\!]\rho\kappa((\rho', \kappa'), \tau) \\
&= \mathcal{E}[\![\texttt{B}]\!]([[\![\texttt{e}]\!] \mapsto (\mathit{map}\ \mathit{exp}^\wedge [\![\texttt{E}^*]\!]), \\
&\qquad\quad [\![\texttt{r}]\!] \mapsto (\mathit{env}^\wedge \rho), \\
&\qquad\quad [\![\texttt{k}]\!] \mapsto (\mathit{cont}^\wedge \kappa)]\ \rho')\kappa'\tau
\end{aligned}
$$

Given these two central definitions it should be clear how to describe the reflective tower completely in this system. However it should be noticed that the compositionality problem is still not solved, but remains in the part "$\mathcal{E}(\mathit{exp}^\vee a)\ldots$".

We are now sufficiently equipped to specify a reflective tower simulator. As we want everything to be redefinable and first class, everything will be held in the environment. As we want to reify over expressions by applying reifiers, the expressions in function position will be evaluated prior to their arguments. As we want to reify over environments, the semantics will be environment-based rather than a set of rewrite rules with substitution. As we want to reify over continuations, it will be in continuation passing style. And finally in order to handle reflection, a meta-continuation will be added.

1.3 The design of a procedurally reflective architecture

The architecture is described with denotational semantics but it is sufficiently close to Scheme to allow a straightforward transcription.

The general design is based on a continuation-passing style interpreter without special forms. Because it has no special forms, its architecture is classically based on a dichotomy eval/apply and when a compound form is encountered, the object in functional position is evaluated. The apply module realizes a dispatch on the injection tag of the result, the domain of applicable objects being a direct sum of abstractions, reifiers, and so on.

To make it reflective it is extended with a meta-continuation which is passed around by all the valuation functions and the continuations, which have the functionality:

$$\kappa : \mathit{DenotableValue} \times \mathit{MetaContinuation} \rightarrow \mathit{Answer}$$

The abstract syntax is essentially the same as specified earlier. It comprises constants (numerals, booleans, strings), identifiers and parentheses. The semantic algebras for primitive domains are standard:

Identifiers
  Domain $i \in \mathit{Identifier} = \mathit{Identifier}$
Constants
  Domain $c \in \mathit{TruthValue} + \mathit{Number} + \mathit{String}$
  Operations (omitted)

Omitting the valuation function $\mathcal{C}$ for constants, and leaving aside the store, we can already write:

$\mathcal{E}: \mathit{Expression} \times \mathit{Environment} \times \mathit{Continuation} \times$
$\quad \mathit{MetaContinuation} \rightarrow \mathit{Answer}$
$\mathcal{E}[\![\mathtt{C}]\!]\,\rho\,\kappa\,\tau = \kappa\,(\mathit{inConstant}(\mathcal{C}\,[\![\mathtt{C}]\!]))\,\tau$
$\mathcal{E}[\![\mathtt{I}]\!]\,\rho\,\kappa\,\tau = \kappa\,(\rho\,[\![\mathtt{I}]\!])\,\tau$
$\mathcal{E}[\![\mathtt{(F\ E^*)}]\!]\,\rho\,\kappa\,\tau = \mathcal{E}[\![\mathtt{F}]\!]\,\rho\,(\lambda\,a\,\tau.\mathit{apply}\,a\,[\![\mathtt{E}^*]\!]\,\rho\,\kappa\,\tau)\tau$

The domain of applicable objects is a direct sum of all the objects that can occur in function position: abstractions, reifiers, predefined control structures, reified environments and reified continuations. The auxiliary function for applying objects dispatches them according to their injection tag:

$\mathit{apply}: \mathit{ApplicableObject} \times \mathit{Expression}^* \times$
$\qquad \mathit{Environment} \times \mathit{Continuation} \times$
$\qquad \mathit{MetaContinuation} \rightarrow \mathit{Answer}$
$\mathit{apply}\ f\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$= \mathit{cases}\ f\ \mathit{of}\ \mathit{isSubr}(s) \rightarrow\ \mathit{applySubr}\ s\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$\qquad [\!]\ \mathit{isFsubr}(f) \rightarrow\ \mathit{applyFsubr}\ f\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$\qquad [\!]\ \mathit{isAbstraction}(\alpha)$
$\qquad\quad \rightarrow\ \mathit{applyAbstraction}\ \alpha\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$\qquad [\!]\ \mathit{isDeltaReifier}(d)$
$\qquad\quad \rightarrow\ \mathit{applyDeltaReifier}\ d\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$\qquad [\!]\ \mathit{isContinuation}(c)$
$\qquad\quad \rightarrow\ \mathit{applyContinuation}\ c\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$\qquad [\!]\ \ldots\ \mathit{end}$

To apply a primitive function (a *subr* in the Lisp terminology – such as *car*, *cdr*, *etc.*), one checks that the number of formal and actual parameters coincide and evaluates the actuals before applying it. To apply a predefined control structure (an *fsubr* – *if*, *quote*, *meaning*, *etc.* with a fixed arity, and *and*, *or*, *begin* with a variable arity) one does something comparable. As can be expected, the initial environment maps the names of primitive functions and predefined control structures to their associated semantic function.

$\mathit{applyFsubr}\ f\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau = f\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$

To apply an abstraction, all its arguments are evaluated and put in a list (for example). This is performed by an auxiliary function *evlis*. The abstraction is then applied to the list, the current continuation (to ensure proper tail-recursion) and the new meta-continuation (to keep track of possible level shifts).

$\mathit{applyAbstraction}\ \alpha\ [\![\mathtt{E}^*]\!]\ \rho\ \kappa\ \tau$
$= \mathit{evlis}\ [\![\mathtt{E}^*]\!]\ \rho\ (\lambda\,l\,\tau\,.\,\alpha\ l\ \kappa\ \tau)\ \tau$

So far the parameter τ has been passively transmitted. It participaters more actively when reifiers are applied.

Applying a reifier is done by reifying its arguments and the current environment and continuation. Then the environment and continuation of the level above are extracted from the meta-continuation, together with a new meta-continuation. Finally the body of the reifier is evaluated, in effect at the level above its application, since it is in the environment and with the continuation of that level.

$\mathit{applyDeltaReifier}\ r\ [\![\mathtt{E}^*]\!]\rho_n\kappa_n((\rho_{n+1},\kappa_{n+1}),\tau_{n+1})$
$= r\ (\mathit{map}\ \mathit{exp}^\wedge\ [\![\mathtt{E}^*]\!])\ (\mathit{env}^\wedge\rho_n)\ (\mathit{cont}^\wedge\kappa_n)\rho_{n+1}\kappa_{n+1}\tau_{n+1}$

To apply a γ-abstraction is quite similar:

$\mathit{applyGammaReifier}\ g\ [\![\mathtt{E}^*]\!]\rho_n\kappa_n((\rho_{n+1},\kappa_{n+1}),\tau_{n+1})$
$= g\ (\mathit{map}\ \mathit{exp}^\wedge[\![\mathtt{E}^*]\!])\ (\mathit{env}^\wedge\rho_n)\ (\mathit{cont}^\wedge\kappa_n)\kappa_{n+1}\tau_{n+1}$

To make this description more complete, here are three central examples of semantic functions bound in the initial environment. We give them as if they were special forms for the sake of simplicity[12]. Checking whether the number of formal and actual arguments coincide is omitted here.

$\mathcal{E}[\![\mathtt{(lambda\ (I^*)\ B)}]\!]\,\rho\,\kappa\,\tau$
$= \kappa(\mathit{inAbstraction}(\lambda\,l\,\kappa\,\tau.$
$\qquad \mathcal{E}\ [\![\mathtt{B}]\!]\ \mathit{extendEnvironment}([\![\mathtt{I}^*]\!]\ l\ \rho)\ \kappa\,\tau))\tau$

where the function *extendEnvironment* straightforwardly extends an environment from of a sequence of identifiers and an isomorphic list of values.

$\mathcal{E}[\![\mathtt{(delta\ (e\ r\ k)\ B)}]\!]\,\rho\,\kappa\,\tau$
$= \kappa(\mathit{inDeltaReifier}(\lambda\,e\,r\,k\,\rho\,\kappa\,\tau.$
$\qquad \mathcal{E}[\![\mathtt{B}]\!]\ ([\ [\![\mathtt{e}]\!] \mapsto e,$
$\qquad\qquad [\![\mathtt{r}]\!] \mapsto r,$
$\qquad\qquad [\![\mathtt{k}]\!] \mapsto k\ ]\ \rho)\ \kappa\,\tau))\ \tau$

The operational value of a δ-abstraction closes its formal parameters and its body.

$\mathcal{E}[\![\mathtt{(gamma\ (e\ r\ k)\ B)}]\!]\,\rho\,\kappa\,((\rho',\kappa'),\tau)$
$= \kappa(\mathit{inGammaReifier}(\lambda\,e\,r\,k\,\kappa\,\tau.$
$\qquad \mathcal{E}[\![\mathtt{B}]\!]\ ([\ [\![\mathtt{e}]\!] \mapsto e,$
$\qquad\qquad [\![\mathtt{r}]\!] \mapsto r,$
$\qquad\qquad [\![\mathtt{k}]\!] \mapsto k\ ]\ \rho')\ \kappa\,\tau))\ ((\rho',\kappa'),\tau)$

[12]In an implementation the identifier `lambda`, for example, would be bound to a predefined control structure producing the operational value of an λ-abstraction: a closure.

These specifications are compositional: by introducing the meta-continuation in the semantic functions, we have expressed the denotation of reifiers and their application without breaking the denotational assumption. The only new point is that the body of, *e.g.*, the δ-abstraction is evaluated both in another environment and with another continuation than the current one (since the current has been reified). Comparing this with the classical specification of a λ-abstraction, one can see that the body of the λ-abstraction is evaluated with the current continuation, but in another environment, *i.e.*, its environment of definition, extended.

What remains to do now is to specify the function *meaning* and how to initialize the meta-continuation.

In the initial environment, the identifier `meaning` is bound to the semantic function *meaning*, in the domain $Fsubr$. A naive definition of *meaning* would let it evaluate its three arguments, convert them down and stack the current environment and continuation in the meta-continuation – that is, assuming that *meaning* is actually applied to three arguments:

$$\begin{aligned}&\mathit{meaning}\;[\![\texttt{E R K}]\!]\,\rho\,\kappa\,\tau\\&= \mathcal{E}\;[\![\texttt{E}]\!]\,\rho\\&\quad(\lambda\,a\,\tau\,.\,\mathcal{E}\;[\![\texttt{R}]\!]\,\rho\\&\qquad(\lambda\,b\,\tau\,.\,\mathcal{E}\;[\![\texttt{K}]\!]\,\rho\\&\qquad\quad(\lambda\,c\,\tau\,.\,\mathit{reflect}\;a\;b\;c\;\rho\;\kappa\;\tau)\;\tau)\;\tau)\;\tau\\&\mathit{reflect}\;a\;b\;c\;\rho\;\kappa\;\tau\\&= \mathcal{E}\;(\mathit{exp}^{\vee}a)\,(\mathit{env}^{\vee}b)\,(\mathit{cont}^{\vee}c)\,((\rho,\,\kappa),\,\tau)\end{aligned}$$

This definitional clause is not compositional: the right-hand side contains an expression: $(\mathit{exp}^{\vee}a)$ that is not a proper subpart of the left-hand side, which violates the denotational assumption[13].

There exists an alternative to the function *meaning*, slightly less powerful but compositional. We call it `meaning-prime` – shortened `meaning'` – and the idea is that it behaves as `meaning` but without evaluating its first argument[14]:

$$\begin{aligned}&\mathcal{E}[\![\texttt{(meaning' E R K)}]\!]\,\rho\,\kappa\,\tau\\&= \mathcal{E}[\![\texttt{R}]\!]\,\rho\,(\lambda\,b\,\tau\,.\,\mathcal{E}[\![\texttt{K}]\!]\,\rho\,(\lambda\,c\,\tau\,.\,\mathit{reflect'}\,[\![\texttt{E}]\!]\;b\;c\;\rho\;\kappa\;\tau)\;\tau)\,\tau\end{aligned}$$

$$\begin{aligned}&\mathit{reflect'}\,[\![\texttt{E}]\!]\;b\;c\;\rho\;\kappa\;\tau\\&= \mathcal{E}\;[\![\texttt{E}]\!]\,(\mathit{env}^{\vee}b)\,(\mathit{cont}^{\vee}c)\,((\rho,\,\kappa),\,\tau)\end{aligned}$$

This definitional clause is compositional. It satisfies the fairly large domain of application where one would give an expression and the binding of its free variables instead of building a new expression. Of course, one could say that the problem of compositionality is not solved but just pushed away, since extending a reified environment transforms a list of denotable values into a list of syntactic identifiers. However, most of the time this list is static, so introducing the operator `extend-reified-environment-prime` – shortened `extend-reified-environment'` – non-strict in its first argument, fulfills both requirements of usefulness and compositionality. With

$$\begin{aligned}&\mathcal{E}[\![\texttt{(extend-reified-environment' (I}^{*}\texttt{) L R)}]\!]\rho\kappa\tau\\&= \mathcal{E}[\![\texttt{L}]\!]\,\rho\,(\lambda\,l\,\tau.\mathcal{E}[\![\texttt{R}]\!]\,\rho\,(\lambda\,r\,\tau.\\&\quad\kappa(\mathit{env}^{\vee}(\mathit{extendEnvironment}\,[\![\texttt{I}^{*}]\!]\,l\,(\mathit{env}^{\vee}r)))\tau)\tau)\tau\end{aligned}$$

one can write compositionally

```
0-3> (let ((f (lambda (x) x)) (d "hello world"))
       (meaning' (f d)
                 (extend-reified-environment'
                     (f d)
                     (list f d)
                     (reify-new-environment))
                 (lambda (a) a)))
0-3: hello world
0-4>
```

Extending the environment rather than building a piece of program originates in [Wand & Friedman 86]. It turns out not to be only a trick for avoiding the use of `quote`, but a general way to preserve compositionality. Thus in Blond one can reflect non-compositionally, with `meaning`, or compositionally (and as generally, provided `extend-reified-environment'`) with `meaning'`.[15]

Another way to specify a reflective program compositionally is to take it as a whole and use the algebraic properties of operators over reified expressions. The reflective description of identity:

$$\mathcal{E}[\![\texttt{((delta (e r k) (meaning (car e) r k)) baz)}]\!]\,\rho\,\kappa\,\tau$$

reduces to

$$\mathcal{E}\;(\mathit{exp}^{\vee}(\mathit{exp}^{\wedge}[\![\texttt{baz}]\!]))\,\rho\,\kappa\,\tau$$

which does not break the denotational assumption, as

$$\mathit{exp}^{\vee}\circ\mathit{exp}^{\wedge} = \mathit{identity}_{\mathrm{Expression}}$$

Finally, the meta-continuation is initialized to

$$\mathit{fix}\;\lambda\,\tau\,.\,((\rho,\,\kappa),\,\tau)$$

where ρ is the initial environment and κ the initial continuation. In an implementation κ could be a toplevel loop.

[13] But is the whole point of Kleene's second recursion theorem.

[14] In that respect it is dual to `if` which is strict in its first argument and non-strict in each of the others. `meaning'` is non-strict in its first argument and strict in the two others.

[15] `ASET'` in [Steele & Sussman 78] has a comparable genesis.

2 Environments and Control in a Reflective Tower

This section analyzes the consequences of pairing the environment and continuation of the levels above in the meta-continuation. The first is that reification occurs in a correct environment, which it did not in Brown. The second consequence is the question of the current continuation when a reified continuation is applied: should it be abandoned as in Scheme or stacked in the meta-continuation as in Brown? Finally, we discuss ephemeral levels and proper tail-reflection.

2.1 Reification: extending which environment?

The goal of this section is to point out in which environment the body of a reifier should be evaluated. This question is not trivial, as in Brown it is the environment from the level of definition of the reifier[16] and the result is to mask variables, in a way very similar to the old *funarg* problem [Moses 70]. Let us take the Brown definition of `exit`:

```
(lambda (x)
    ((make-reifier
        (lambda (e r k) x))))
```

This function returns a result from one level to the level above it. For one thing, this conflicts with the referential transparency of the level above: that is, since `x` is bound in the reified environment `r`, its value should be found there. A correct Brown definition would be:

```
(lambda (x)
    ((make-reifier
        (lambda (e r k) (r 'x)))))
```

since an environment is reified as a function of type:

$$Identifier \rightarrow DenotableValue$$

The point is better communicated by adding semantic brackets and subscripting them with their level:

$[\![$`(lambda (x) ((make-reifier`
`(lambda (e r k)` $[\![$`x`$]\!]_{n+1}$`))))`$]\!]_n$

In the following Blond scenario, $[\![\mathtt{x}]\!]_{n+1}$ is evaluated in the correct environment ρ_{n+1}.

```
0-1> (let ((x "foo"))
        (meaning' (let ((x "bar"))
                       ((delta (e r k) x)))
                  (reify-new-environment)
                  (lambda (a) "whatever")))
0-1: foo
0-2>
```

In the environment ρ_0, extended with the binding of `x` to the value `"foo"`, a new (and anonymous) level is spawned with a fresh initial environment and an arbitrary continuation. At that new level, the environment is extended to map `x` to the value `"bar"` and reification occurs, leading to:

$$\begin{aligned}\mathcal{E}\,[\![\mathtt{x}]\!]\;([[\![\mathtt{e}]\!] &\mapsto (map\,exp^{\wedge}\;[\![\;]\!]),\\ [\![\mathtt{r}]\!] &\mapsto (env^{\wedge}([[\![\mathtt{x}]\!] \mapsto \mathcal{C}[\![\texttt{"bar"}]\!]\,\rho)),\\ [\![\mathtt{k}]\!] &\mapsto (cont^{\wedge}\kappa)]\\ [[\![\mathtt{x}]\!] &\mapsto \mathcal{C}[\![\texttt{"foo"}]\!]\,\rho_0)\;\kappa_0\end{aligned}$$

and the continuation κ_0 is applied to the expressible value $\mathcal{C}[\![\texttt{"foo"}]\!]$.

This shows that a reifier cannot be defined as a λ-abstraction, and has made us rule out `make-reifier` in Blond. Instead, the body of a reifier should be evaluated either in the environment of the level above its definition or in the environment of the level above its application. Blond supplies both possibilities: the first with γ-abstractions and the second with δ-abstractions. At first glance, this looks surprisingly close to the question of having a lexical or dynamic scope in a non-reflective language. However, the question can be identically put for continuations: one could imagine the body of a reifier to be evaluated with a continuation above its level of application as well as its level of definition[17]. In a reflective tower, though, the actual environment and continuation above the application of a reifier would be lost[18]. This shows the limits of the comparison: again, further experiments are needed.

2.2 Jumpy vs. pushy continuations

In Scheme, when a captured continuation is applied, the context of its application is thrown away. This is often referred to as a "black hole" behaviour and [des Rivières 88] classifies such continuations as *jumpy*. In contrast, continuations have been made *pushy* in Brown: whenever a reified continuation is applied, the current continuation is stacked on the meta-continuation, rather than being thrown away[19]. This

[16]Although it could be any other, since in Brown one can make a reifier out of an abstraction already defined, with something like `(make-reifier foo)`. This leads to evaluate the body of the reifier in the closure environment of `foo`, lexically extended with the reified expression, environment and continuation.

[17]This is not irrealistic: procedures returning in their context of definition rather than application exist and are used as exceptions.

[18]To some extent, it justifies why debuggers are *called* from an error point.

[19]This phenomenon is distinct from the *functional* continuations of [Felleisen *et al.* 87]: applying compositional continua-

is illustrated in the following Brown scenario, adapted from [Wand & Friedman 86]:

```
0-> ((make-reifier (lambda (e r k)
                    (cons (k 'a) (k 'd)))))
0:: a
0-> (exit 'foo)
0:: d
0-> (exit 'bar)
1:: (foo . bar)
1->
```

The body of the reifier is processed at level 1. The reified continuation is the top level loop at level 0. It is applied twice. Each time, the current continuation is stacked on the meta-continuation. Exiting twice from level 0, one obtains a result at level 1.[20]

Having jumpy continuations, the scenario becomes:

```
0-1> (load "exit.bl")
exit
0-1: loaded
0-2> ((delta (e r k)
        (cons (k "a") (k "d"))))
0-2: a
0-3> (exit "foo")
2-0: foo
2-1>
```

and the levels 1 and 0 are irremediably lost – level 1 because it has been replaced by level 0 and level 0 because it has been exited. Thinking about the environment extensions in a procedural language, one might make an analogy with dynamic and static scoping: in the former, extensions are managed in a stack, whereas in the latter, they are thrown away when a procedure is applied tail-recursively and its lexical extension takes place.

We argue here that the pushy feature muddies the meta-continuation, because the current continuation is not only stacked at reflection time, but also when a reified continuation is applied. In fact, the point is not whether a continuation is pushy or jumpy, since this is not a property of the continuation itself, but of how it is applied. Applying a reified continuation in a jumpy way is modelled by:

$$applyContinuation\ c\ [\![\mathrm{E}]\!]\ \rho\ \kappa\ \tau = \mathcal{E}[\![\mathrm{E}]\!]\ \rho\ c\ \tau$$

whereas applying it in a pushy way would be:

tions returns a result to the point of their application. Thus they can be composed. In Brown, applying a continuation stacks the current one on the meta-continuation, so that reifying or exiting from the level below reactivates it – the point is that there are top level loops at each level.

[20]One can note that Brown evaluates the arguments of `cons` from left to right.

$$applyContinuation\ c\ [\![\mathrm{E}]\!]\ \rho\ \kappa\ \tau = \mathcal{E}[\![\mathrm{E}]\!]\ \rho\ c\ ((\rho, \kappa), \tau)$$

These equations underline the problem that stacking a pushy continuation is done together with an environment – here the current one, implicitly. In Blond, where continuations are jumpy by default, a pushy behaviour is obtained regularly via `meaning`, where it is explicitly specified which environment should be stacked together with the continuation:

```
0-2> ((delta (e r k)
         (cons (meaning' "a" r k)
               (meaning' "d" r k))))
0-2: a
0-3> (exit "foo")
0-2: d
0-3> (exit "bar")
1-0: (foo . bar)
1-1>
```

Rather than applying a pushy continuation, it is specified that the continuation should be pushed by using `meaning`. The results are identical.

However, this session carries more information than the Brown one, as the Blond prompt makes it explicit that the second iteration of the top level loop at level 0 is captured and activated twice.

But clearly again, further experiments are needed, so we have provided a toggle in Blond for switching the application mode of reified continuations between jumpy and pushy, the default mode being jumpiness. This is only for convenience since one can anyway jumpify a pushy continuation with:

```
(define jumpify
   (lambda (pc)
      (lambda (a)
         ((delta (e r k)
             (meaning' a r (r 'pc)))))))
```

and pushify a jumpy continuation with

```
(define pushify
   (lambda (jc)
      (lambda (a)
         (let ((env (extend-reified-environment'
                        (dummy) (list a)
                        (reify-new-environment))))
            (meaning' dummy env jc)))))
```

using call by value – that is, if reification occurs while evaluating the argument, the reified continuation is the current one and not the one to be applied. The other way around is specified in appendix 2.

2.3 Ephemeral levels and proper tail-reflection

This section introduces a phenomenon analogous to tail-recursion for reflective programming: *tail-reflection*. Here is a tail-reflective program:

```
(delta (e r k)
  (meaning (car e) r
    (lambda (p)
      (meaning (ef p (cadr e) (caddr e)) r k))))
```

This reifier follows the usual reflective definition of `if` in 3-Lisp or in Brown, using the conditional function `ef`, strict on all its arguments. The test part of the form is evaluated and according to its result one of the two alternatives are evaluated. Translating this reifier directly would give a denotational specification flirting with non-compositionality:

$$\mathcal{E}\llbracket\texttt{(if P T E)}\rrbracket\,\rho\,\kappa = \mathcal{E}\llbracket\texttt{P}\rrbracket\,\rho\,(\lambda\,p.\mathcal{E}\,(p \to \llbracket\texttt{T}\rrbracket \;\|\; \llbracket\texttt{E}\rrbracket)\,\rho\,\kappa)$$

but meaning the same as the correct:

$$\mathcal{E}\llbracket\texttt{(if P T E)}\rrbracket\,\rho\,\kappa = \mathcal{E}\llbracket\texttt{P}\rrbracket\,\rho\,(\lambda\,p.p \to \mathcal{E}\llbracket\texttt{T}\rrbracket\,\rho\,\kappa \;\|\; \mathcal{E}\llbracket\texttt{E}\rrbracket\,\rho\,\kappa)$$

Let us retrace the essential steps of the computation: when the reifier is applied, its unevaluated arguments, the current environment and the current continuation are reified; the top-most environment and continuation are popped off the meta-continuation; the body of the reifier is processed in that environment, extended with its formal parameters, and that continuation; this body spawns a new level to evaluate the test; the continuation popped from the meta-continuation and the extended environment are pushed back; the specified continuation `(lambda (p) ...)` is applied to the result of the test; in its body a new level is spawned to evaluate one of the alternatives; and the initial continuation will eventually be applied[21].

As a whole, two new levels are created for evaluating the test and one alternative. The first level is ephemeral because it only exists for evaluating the test part. Both calls to `meaning` are tail-reflective. Retracing the essential steps of a derivation:

$$\begin{aligned}
&\mathcal{E}\,\llbracket\texttt{((delta (e r k)}\\
&\qquad\quad\texttt{(meaning (car e) r (lambda (p) ...)))}\\
&\qquad\texttt{\#t "hello" "world")}\,\rrbracket\,\rho_0\,\kappa_0\,((\rho_1,\kappa_1),\tau_1)\\
&= \mathcal{E}\,\llbracket\texttt{(meaning (car e) r (lambda (p) ...))}\rrbracket\,\rho_1'\,\kappa_1\,\tau_1\\
&\text{where } \rho_1' = [\llbracket\texttt{e}\rrbracket \mapsto (map\,exp^{\wedge}\,\llbracket\texttt{\#t "hello" "world"}\rrbracket)\\
&\qquad\qquad\;\; \llbracket\texttt{r}\rrbracket \mapsto (env^{\wedge}\rho_0),\\
&\qquad\qquad\;\; \llbracket\texttt{k}\rrbracket \mapsto (cont^{\wedge}\kappa_0)]\,\rho_1\\
&\text{and, using the algebraic property}\\
&\qquad\quad car(map\,exp^{\wedge}\,\llbracket\texttt{(E E}^{*}\texttt{)}\rrbracket) = exp^{\wedge}\llbracket\texttt{E}\rrbracket\\
&= \mathcal{E}(exp^{\vee}(exp^{\wedge}\llbracket\texttt{\#t}\rrbracket))\\
&\qquad(env^{\vee}(env^{\wedge}\rho_0))\\
&\qquad(\lambda\,v\,((\rho_1'',\kappa_1),\tau)\,.\\
&\qquad\quad\mathcal{E}\llbracket\texttt{(meaning (ef p (cadr e) (caddr e)) r k)}\rrbracket\\
&\qquad\quad\;([\llbracket\texttt{p}\rrbracket \mapsto v]\,\rho_1')\,\kappa_1\,\tau_1)\\
&\qquad((\rho_1',\kappa_1),\tau_1)
\end{aligned}$$

[21] As one may note, this specification is properly tail-recursive: the initial continuation is transmitted for evaluating the selected alternative.

The next reduction is properly tail-reflective: because `(lambda (p) ...)` is a function at the level subscripted with 1 and a continuation at the level subscripted with 0, its body is evaluated at level 1 with the corresponding meta-continuation. In particular, it is evaluated in the environment ρ_1' so the environment ρ_1'' is not used.

$$\begin{aligned}
&= \mathcal{E}\,\llbracket\texttt{(meaning (ef p (cadr e) (caddr e)) r k)}\rrbracket\\
&\qquad([\llbracket\texttt{p}\rrbracket \mapsto \mathcal{C}\llbracket\texttt{\#t}\rrbracket]\,\rho_1')\,\kappa_1\,\tau_1\\
&= \mathcal{E}\,(exp^{\vee}(exp^{\wedge}\llbracket\texttt{"hello"}\rrbracket))\,(env^{\vee}(env^{\wedge}\rho_0))\\
&\qquad(cont^{\vee}(cont^{\wedge}\kappa_0))\,((([\llbracket\texttt{p}\rrbracket \mapsto \mathcal{C}\llbracket\texttt{\#t}\rrbracket]\,\rho_1',\,\kappa_1),\,\tau_1)\\
&= \kappa_0\,(\mathcal{C}\llbracket\texttt{"hello"}\rrbracket)\,((([\llbracket\texttt{p}\rrbracket \mapsto \mathcal{C}\llbracket\texttt{\#t}\rrbracket]\,\rho_1',\,\kappa_1),\,\tau_1)
\end{aligned}$$

Maybe to some surprise, the top-most environment in the meta-continuation has been altered. Assuming `rift` to be bound to the present reifier:

```
>>> (blond)
0-0: bottom-level
0-1> (load "rift.bl")
rift
0-1: loaded
0-2> (rift #t "hello" "world")
0-2: hello
0-3> ((delta (x y z) e))
1-0: (#t hello world)
1-1>
```

Level 1 memorizes in its environment the bindings of all the formal parameters of the reifiers applied at level 0. This extension concerns only the top-most environment in the meta-continuation and not the environment of any later continuation, since they close environments of their own. At iteration 1 of level 1 in the scenario, the variable `e` is unbound.

There are a couple of lessons that we can learn from this example. About the environment: the phenomenon is inherent to the model of a reflective tower – reifiers are processed in the interpreter above. If one reifies and reflects back, the altered environment reminds him that he *already* has reified and reflected back. Thus the extension of the environment at the level above.

About the control: the description is naturally tail-reflective since a continuation like

$$(\lambda\,a\,((\rho,\kappa),\tau)\,.\,\kappa\,a\,\tau)$$

does not occur when a reified continuation or a function is given as a continuation to `meaning`. This would be the typical continuation of an ephemeral level. Whereas a properly tail-reflective specification would be

$$\begin{aligned}
&\mathcal{E}\llbracket\texttt{(meaning' E R (lambda (a) B))}\rrbracket\,\rho\,\kappa\,\tau\\
&= \mathcal{E}\llbracket\texttt{R}\rrbracket\,\rho\,(\lambda\,r\,\tau.\,\mathcal{E}\llbracket\texttt{E}\rrbracket\,(env^{\vee}r)\\
&\qquad\qquad\qquad(\lambda\,a\,((\rho,\kappa),\tau).\,\mathcal{E}\llbracket\texttt{B}\rrbracket\,([\llbracket\texttt{a}\rrbracket \mapsto a]\rho)\\
&\qquad\qquad\qquad\qquad\qquad\kappa\,\tau)\,((\rho,\kappa),\tau))\,\tau
\end{aligned}$$

a non-properly tail-reflective specification could be:

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(meaning' E R (lambda (a) B))}]\!]\,\rho\,\kappa\,\tau\\
&= \mathcal{E}[\![\texttt{R}]\!]\,\rho\\
&\quad (\lambda\, r\,\tau.\,\mathcal{E}[\![\texttt{E}]\!]\,(env^{\vee} r)\\
&\qquad\qquad (\lambda\, a\,\tau.\,\mathcal{E}[\![\texttt{B}]\!]\,([[\![\texttt{a}]\!] \mapsto a]\rho)\\
&\qquad\qquad\qquad (\lambda\, v((\rho,\kappa),\tau).\,\kappa\, v\,\tau)\,\tau)\,((\rho,\kappa),\tau))\,\tau
\end{aligned}$$

The pattern is comparable to a call followed by a return in assembly language, the difference being that the last thing done at one level is to create a new one.

To summarize: by definition of a reflective tower, the environment above any reification followed by a reflection is extended with the formal parameters of the reifier. An ephemeral level is created each time a level is spawned with, as a continuation, a function which is not a reified continuation. This corresponds to a composition of functions and can be treated properly, the idea being that as a proper tail-recursion requires a constant continuation, a proper tail-reflection requires a constant meta-continuation.

3 Comparison with related work

3.1 3-Lisp

[Smith 82] introduces the model of the reflective tower together with an implementation: 3-Lisp. It addresses a number of the central issues about reflection such as its philosophical background, the limits of machine representation, the semantic framework, the finite implementation of the infinite tower, the possible mathematical characterizations, *etc.*.

3-Lisp is based on a statically scoped Lisp-like language. Since it is reflective, it is also higher-order and supports macros. Furthermore it is semantically rationalized, *i.e.*, expressions are not dereferenced but only normalized to a co-designating normal form, unless explicitly stated otherwise. Thus the levels in the tower correspond to different levels of designation and going up and down also changes the level of designation.

The 3-Lisp implementation creates the levels above on demand and only keeps a stack of the already created levels. The internal structures of the interpreter are managed as data structures and reified as such, rather than being curried and untyped abstractions, as in Brown, which uses an applicative-order Y combinator to create levels. In Blond, levels are also defined recursively. They rely on the recursive capabilities of the definition language, rather than on an explicit Y combinator.

3.2 Brown

[Friedman & Wand 84] describes an interpreter for Brown: a language with access to internal structures of the interpreter, *i.e.*, expressions, environments and continuations. This process is referred to as reification. By symmetry, structures can be (re-)installed as new current expression, environment and continuation of the interpreter. This process is referred to as reflection. Finally, the various possibilities of keeping or throwing away the current continuation at reification and reflection time are analyzed.

[Wand & Friedman 86] essentially describe a tightened Brown. A new structure is introduced as the spine of the model: the meta-continuation. At reflection time, the current continuation is pushed on it, and at reification time, the new current continuation is popped from the top of the meta-continuation. The meta-continuation represents the tower, and this view is acknowledged in [des Rivières 88].

The essence of Brown is to provide a direct interpretation of any reflective program, managing reflective procedures with the meta-continuation. Brown is thus single-threaded. The implementation of Brown in 1984 was flat in the sense of towerless, while the one of 1986 is towerful, with the meta-continuation. It is an impressively neat piece of Scheme programming, where basically all functions are curried and untyped abstractions. [Wand, Friedman & Duba 86] attempts to circumvent the cost of shifting environments and continuations up and down by coercing them.

We have described the effect of a couple of design choices in Brown, which have occurred to us while establishing the extensional aspects of the model described in this paper: pushy continuations interfere with the management of the meta-continuation; at reification time, an evaluation occurs in an incorrect environment.

But environments are a complex problem, and as a proof we take the fact that their conversion to Brown functions is different in [Wand & Friedman 86] and [Wand, Friedman & Duba 86]: this results in distinct behaviours when studying "pathologies for shifting". The idea is that one specifies a conversion without level-shifting while the other shifts levels. Therefore, exiting at that point, one leaves from two different levels and thus arrives at two different levels above.

[Wand & Friedman 88] refines the article from 1986, essentially about the algebraic aspects of upping and downing objects at reification and reflection time.

To compare Brown with Blond: Brown starts from a flat, towerless reflective model to build a reflective tower. Blond starts from a non-reflective tower to make it reflective.

Conclusion and issues

This paper reports an extensional and intensional study of the reflective tower. A new approach has been taken: first to describe the infinite tower in itself, and then consider the impact of meta-level connections on it. Based on this and the additional constraint that all the interpreters are meta-circular, we have designed and implemented a reflective tower simulator. The result is Blond, a reflective dialect of Scheme.

This study shows why a meta-continuation is necessary and sufficient to create a reflective dialect of a language which can be defined by a semantics manipulating expressions, environments and continuations. We have pointed out not only how but why a reflective tower can be implemented using only one running interpreter, that is without order of magnitude overhead for each downwards level shift.

With this more clear view of the reflective tower we can see some of its possibilities and limits. For example, the following reflective definition of `quote`:

```
(delta (e r k) (k (car e)))
```

requires the reified continuation to be pushy. Also, as a reifier, it extends the environment at the level above with the bindings of `e`, `r` and `k`. The former can be solved with:

```
(delta (e r k)
  (meaning-prime dummy
                 (extend-reified-environment-prime
                   (dummy) (list (car e)) r)
                 k))
```

the `-prime` operators being non-strict, but at the meta-level and compositionally. Another limitation is that the model does not have a store and thus no side effects. This explains why side effects above the current level in the tower are lost in the implementation, as discussed in the introduction.

A natural extension would be reification over the store, *e.g.* to give a formal model of garbage collection. Another extension could be variadic reifiers, able to reify over the meta-continuation, the consequent meta-meta-continuation, and so on. This would need to maintain one extra dimension of meta-ness and could be used to model distributed systems.

An immediate perspective is to explore Blond programming and more generally, to continue investigating the intriguing extensional similarities between reflective towers and self-applicable partial evaluators.

Acknowledgements

To Bruce F. Duba for the intense hours introducing the first author to Brown; to Neil D. Jones and Jacqueline Signorini for discussing issues about reflection; and to Anders Bondorf, Andrzej Filinski, Kristoffer Høgsbro Holm, Torben Mogensen and Peter Sestoft for their optimizing rereading.

References

[Danvy 87] Olivier Danvy: *Accross the Bridge between Reflection and Partial Evaluation*, Proceedings of the Workshop on *Partial Evaluation and Mixed Computation*, Dines Bjørner, Andrei P. Ershov and Neil D. Jones (eds.), North-Holland (to appear), Gl. Avernæs, Denmark (October 1987)

[Danvy & Malmkjær 88]
Olivier Danvy, Karoline Malmkjær: *A Blond Primer*, draft, DIKU, University of Copenhagen, Copenhagen, Denmark (February 1988)

[des Rivières & Smith 84] Jim des Rivières, Brian C. Smith: *The Implementation of Procedurally Reflective Languages*, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming pp 331–347, Austin, Texas (August 1984)

[des Rivières 88] Jim des Rivières: *Control-Related Meta-Level Facilities in LISP*, from *Meta-Level Architectures and Reflection*, Patti Maes & Daniele Nardi (eds.), North-Holland (1988)

[Felleisen *et al.* 87] Matthias Felleisen, Daniel P. Friedman, Bruce F. Duba, John Merrill: *Beyond Continuations*, Technical Report No 216, Computer Science Department, Indiana University, Bloomington, Indiana (February 1987)

[Friedman & Wand 84] Daniel P. Friedman, Mitchell Wand: *Reification: Reflection without Metaphysics*, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming pp 348–355, Austin, Texas (August 1984)

[Jones *et al.* 88] Neil D. Jones, Peter Sestoft, Harald Søndergaard: *MIX: a Self-Applicable Partial Evaluator for Experiments in Compiler Generation*, to appear in the International Journal *LISP and Symbolic Computation*, (1988)

[Moses 70] Joel Moses: *The Function of FUNCTION in LISP, or Why the FUNARG Problem should be called the Environment Problem*, MIT-AIL, AI Memo No 199, Cambridge, Massachusetts (June 1970)

[Muchnick & Pleban 80] Steven S. Muchnick, Uwe F. Pleban: *A Semantic Comparison of Lisp and Scheme*, Conference Record of the 1980 LISP Conference pp 56-64, Stanford, California (August 1980)

[Rees & Clinger 86] Jonathan Rees, William Clinger (eds): *Revised^3 Report on the Algorithmic Language Scheme*, Sigplan Notices, Vol. 21, No 12 pp 37-79 (December 1986)

[Schmidt 86] David A. Schmidt: *Denotational Semantics: a Methodology for Language Development*, Allyn and Bacon, Inc. (1986)

[Smith 82] Brian C. Smith: *Reflection and Semantics in a Procedural Language*, Ph. D. thesis, MIT/-LCS/TR-272, Cambridge, Massachusetts (January 1982)

[Smith 84] Brian C. Smith: *Reflection and Semantics in Lisp*, Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages pp 23-35, Salt Lake City, Utah (January 1984)

[Steele & Sussman 78] Guy L. Steele Jr., Gerald J. Sussman: *The Revised Report on SCHEME, a Dialect of LISP*, MIT-AIL, AI Memo No 452, Cambridge, Massachusetts (January 1978)

[Sturdy 88] John C. G. Sturdy: Ph. D. thesis (forthcoming), University of Bath, School of Mathematical Sciences, Bath, England (1988)

[Talcott 85] Carolyn Talcott: *The Essence of $\mathcal{R}$um: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*, Ph. D. thesis, Department of Computer Science, Stanford University, Stanford, California (August 1985)

[Wand & Friedman 86] Mitchell Wand, Daniel P. Friedman: *The Mystery of the Tower Revealed: a Non-Reflective Description of the Reflective Tower*, Conference Record of the 1986 ACM Symposium on LISP and Functional Programming pp 298–307, Cambridge, Massachusetts (August 1986)

[Wand, Friedman & Duba 86] Mitchell Wand, Daniel P. Friedman, Bruce F. Duba: *Getting the Levels Right (Preliminary Report)*, Preprints of the *Workshop on Meta-Level Architectures and Reflection*, Patti Maes & Daniele Nardi (eds.), Alghero, Sardinia (October 1986)

[Wand & Friedman 88] Mitchell Wand, Daniel P. Friedman: *The Mystery of the Tower Revealed: a Non-Reflective Description of the Reflective Tower*, to appear in the International Journal *LISP and Symbolic Computation* (1988)

Appendix 1 – Swapping levels in Blond

```
(define permute!
 (delta (e0 r0 k0)
  ((delta (e1 r1 k1)
    ((delta (e2 r2 k2)
      (let ((R2 (extend-reified-environment'
                  (R0 K0)
                  (list ((r2 'r1) 'r0)
                        ((r2 'r1) 'k0))
                  r2))
            (K2 k2))
       (let ((R1 (extend-reified-environment'
                   (R2 K2)
                   (list R2 K2)
                   (r2 'r1)))
             (K1 (r2 'k1)))
        (meaning' (meaning' (meaning' "done"
           R0 K0)    R2 K2)    R1 K1)))))))))
```

Appendix 2 – Pushifying in Blond

Defining `pushify` can follow three paths: `(pushify (k <exp>))` `(pushify k <exp>)` `((pushify k) <exp>)` and the third one has been chosen here. The following specifies how to make a pushy continuation out of a jumpy one. The point here is that if reification occurs while evaluating the arguments of the pushified continuation, that continuation will be found rather than the current one.

```
(define pushify-bis
  (lambda (jc)
    (meaning'
      (gamma (e r k)
        (meaning'
          (meaning' dummy env cont)
          (extend-reified-environment'
            (env cont)
            (list (extend-reified-environment'
                    (dummy) (list (car e)) r)
                  jc)
            r)
          k))
      (reify-new-environment)
      (lambda (pc) pc))))
```

The idea of using both `meaning'` and a γ-abstraction is that we want to capture the binding of `pc` in a reifier – of course at the price of an environment above.