DIKU Rapport 88/21, October 1988

A Blond Primer

Olivier Danvy & Karoline Malmkjær

DIKU – University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, DENMARK
uucp: danvy@diku.dk & karoline@diku.dk

Abstract

This report describes how to use the reflective tower Blond. A reflective tower is a computational architecture where programs are given access to representations of the current state of computation. This models an infinite tower of interpreters interpreting each other, meta-circularly.

Blond is a Scheme interpreter extended to be reflective. This report concentrates on its reflective extension rather than on the standard Scheme characteristics. Reification, reflection, reified environments, and reified continuations are described in detail. Each key point is illustrated with scenarios. The first entries in a Blond library are assembled, and finally Blond is run in Blond, achieving orthogonal reflective towers. A glossary and the Scheme source code are provided in appendix.

This report informally describes its 1988 implementation, as specified in the article *"Intensions and Extensions in a Reflective Tower"*, presented at the 1988 ACM Symposium on Lisp and Functional Programming, where Blond is formally described.

Keywords

Procedural reflection, reflective towers, reification, reflection, reifiers.

Contents

Introduction

Blond is the name of a reflective tower simulator developed at DIKU, the Institute of Datalogy at the University of Copenhagen.

The abstract model is described in [Danvy & Malmkjær 88] and [Malmkjær 88]. This report intends to be a manual. It is an informal, but self-contained presentation of Blond.

The basic idea of a reflective tower [Smith 82] is to have a series of interpreters 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" at its top. The tower is conceptually infinite.

At the beginning of a Blond session, all levels are interpreting the level below at the first iteration of a print-eval-read loop. The final result of their efforts is the lowest level, with its interactive print-eval-read loop, iterating over the input and output streams. We refer to it as the bottom level loop.

```
.         .
.         .
.         .
+---------+        △                 (<reifier><argument>*)
|    2    |    Reification
+---------+
|    1    |    Reflection
+---------+        ▽          (meaning <expression><expression><expression>)
|    0    |
+---------+
```

At each level, the computation is determined by the expression being evaluated, its environment of evaluation, and the continuation to apply to the result.

Reification is achieved by applying a reifier: a ternary procedure which, when it is applied at one level, is given the specified arguments, the current environment and the current continuation as actual parameters (the arguments are reified in a list). The body of a reifier is evaluated at the level above.

Reflection is obtained by applying the function `meaning` to three arguments. They are respectively an expression to be evaluated, an environment in which to evaluate the expression and a continuation to apply to the result of the evaluation.

When a reified continuation from level m is applied at level n, the computation of level m continues with a result from level n. As the continuation is a closure, neither expression nor environment are necessary – they are specified in the continuation itself.

Blond offers the third implementation of a reflective tower we know of. It comes after 3-Lisp [Smith & des Rivières 84] and Brown [Friedman & Wand 84] [Wand, Friedman & Duba 86] [Wand & Friedman 88]. 3-Lisp is implemented in InterLisp-D and runs on a Lisp Machine. It is a complete system on its own. Brown is implemented in Scheme 84 and is a minimal reflective system. With Blond, we have chosen to offer basically the language Scheme at each level of the tower, and to enrich it with reification and reflection.

Blond is currently implemented in Scheme, meta-circularly to be able to run Blond in Blond.

With the Blond project, we hope to contribute to the general understanding of the reflective tower: extensionally by describing it formally [Danvy & Malmkjær 88] [Danvy & Malmkjær 89]; and intensionally by implementing it and document the implementation [Danvy & Malmkjær 88'].

Section 1 presents a session with Blond. At each level stands a Scheme interpreter with a print-eval-read interactive loop. That language is briefly presented in section 2. Section 3 describes the reflective extensions: reifiers and how to spawn new levels. Sections 4 and 5 address reified environments and reified continuations. Section 6 describes the standard Blond library and how to program in Blond. Section 7 discusses Blond in Blond and the construction of orthogonal reflective towers. After a conclusion, a Blond glossary and the Scheme source code of Blond are provided in appendix.

Blond has been developed in Scandinavia and is named after Brown, the reflective tower of Indiana University.

1 A Session with Blond

The following scenario consists of (1) loading a file `exit.bl` defining a function to exit from the current level with a result and (2) applying it. The most visible manifestation of the tower is the prompt system: at each bottom level loop, a prompt indicates both the current level and the current iteration. The latter point proves very useful when applying continuations that come back to an earlier iteration of a bottom level loop.

The session is started at the level 0. Above it there is (conceptually) an infinity of levels processing each other. Exiting from one level one arrives at the level above.

```
> (blond)
0-0: "bottom-level"
0-1> (load "exit.bl")
exit
0-1: "exit.bl"
0-2> (exit "good bye")
1-0: "good bye"
1-1> (load "exit.bl")
exit
1-1: "exit.bl"
1-2> (exit "farewell!")
2-0: "farewell!"
2-1> (blond-exit)
"farvel!"
>
```

Concretely:

- the first prompt is `0-0:` to signal that we are at level 0 and that the number of iterations since the beginning of the bottom level loop is 0: the level 0 has just started up;
- the next prompt is `0-1>` to signal that we (still) are at level 0, that the current iteration in the bottom level loop is 1 and that the system is ready to interact;

- we type `(load "exit.bl")` in order for the system to load the file `exit.bl`, containing the definition of the function `exit`;
- the system loads the file, and gracefully displays some informations witnessing what is currently loaded;
- the prompt `0-1:` signals that the following is the result of the iteration 1 of level 0; that result is the name of the file;
- at the iteration 2 of level 0 we apply the function `exit` to some random expression `"good bye"`; the net effect is to leave the level 0 and arrive to the level immediately above; as this is a reflective tower, the level above is an identical Blond interpreter, which manifests itself with the prompt `1-0:`, signifying that we are now at level 1, that the number of iterations since the beginning of the bottom level loop is 0 and that the current result is `"good bye"`;
- the next prompt tells that this is the iteration number 1 of level 1;
- as we are at level 1, there is no reason to believe that the function `exit` is defined as it was at level 0;[1] we load the file `exit.bl` and exit from level 1;
- at level 2, we end the session with the predefined function `blond-exit`.

What do we learn in this session (apart that "good bye" is "farvel" in Danish)?

Essentially that we face a bottom level loop with a prompt displaying the level in the tower and the number of iterations since the beginning of the bottom level loop. Otherwise, it is Scheme: we can load a file and apply functions.

More fundamentally we learn that a session starts at level 0, and that above it there is (potentially) an infinity of other identical levels, that preexist.

The next scenario illustrates that we can spawn new levels below the current one. We use the predefined function `openloop`:

```
> (blond)
0-0: "bottom-level"
0-1> (openloop "hal")
hal-0: "bottom-level"
hal-1> (openloop "shalmaneser")
shalmaneser-0: "bottom-level"
shalmaneser-1> (load "exit.bl")
exit
shalmaneser-1: "exit.bl"
shalmaneser-2> (exit "cdb")
hal-1: "cdb"
hal-2> (blond-exit)
"farvel!"
>
```

What we learn here is that we can spawn new levels in the tower, and come back to them. This points out that above the current level, there are pre-existing levels, and below, one can create as many levels as he likes (or needs)[2].

[1]Though it may be, as developed in section 4.

[2]This indicates that a reflective tower is actually a reflective tree with the root at the top, in the same way as a

2 Blond as a Reflective Extension of Scheme

At each level of the tower there is an interpreter. It is essentially a Scheme interpreter, that is it processes an untyped λ-calculus applied to integers, strings and lists. It is higher-order and properly tail-recursive.

In addition, Blond is properly tail-reflective [Danvy & Malmkjær 88]: tail-reflective calls are performed iteratively. This extends Scheme's proper tail-recursion.

Contrasting with Scheme, there is no implicit sequence of evaluation in Blond – it must be made explicit with `begin` – and no "internal define": local definitions are made using `letrec`, for simplifying local definitions, global definitions and definitions common to all the levels in the tower. Finally there are no special forms: functions, primitives, control structures, reifiers and by extension reified environments and continuations are first-class – they can be passed as arguments and returned as values. This design has been chosen on an experimental basis, to enforce the idea that control structures actually are compiled reifiers.

3 Reification and Reflection in Blond

In Blond, reification is achieved with reifiers, that are abstractions. Reflecting spawns a new level and is performed with the functions `meaning` and `openloop`.

3.1 Reification

The Blond reifiers are represented as ternary δ or γ-abstractions:

```
(delta (e r k) <body>)
(gamma (e r k) <body>)
```

Applying them has the effect to reify their arguments (in a list), the environment and the continuation. Their body is evaluated at the level above, in an environment lexically extended with the bindings of the three formal parameters to the list of reified expressions, the reified environment and the reified continuation.

```
> (blond)
0-0: "bottom-level"
0-1> (add1 (openloop "marvin"))
marvin-0: "bottom-level"
marvin-1> ((delta (e r k) 41))
0-1: 42
0-2>
```

In this scenario, a new level is started as an argument of the successor function. At that level a reifier is applied, its body consisting of the number 41. It is evaluated at the level above in place of the expression `(openloop "marvin")` and, not that surprisingly, the answer is 42.

conventional control stack in an Algol or Lisp-like implementation merely represents the currently active branch in the control tree.

The difference between δ and γ-abstraction is that the body of a δ-abstraction is evaluated in the environment of the level above its application while the body of a γ-abstraction is evaluated in the environment of the level above its definition[3].

```
0-2> ((lambda (x) (openloop "foo")) 0)
foo-0: "bottom-level"
foo-1> ((delta (e r k) x))
0-2: 0
0-3>
```

In this scenario, the variable `x` is bound in the level above *foo* and can be referred to in the body of a δ-reifier. Conversely, a γ-reifier defined at the level *foo* and applied at any other level would remember of the level 0 environment. Blond offers these two sorts of reifiers which generalize the concepts of static and dynamic scope to multiple levels of interpretation.

3.2 Reflection

Spawning a new level requires an expression to evaluate, an environment in which to evaluate it, and a continuation to apply to the result of the evaluation.

We already know `openloop`: it takes a name (a string, a number, *etc.*) and spawns a new level. We also know that it is the continuation that realizes the bottom level loop of the new level. Thus there is not a single expression to evaluate at one level but all the expressions that will be read along the bottom level loop. Finally, using `openloop` as above gives a new instance of the initial environment to the new level. If one wants to spawn a new level with a predefined environment (obtained by reification), he adds a second (optional) argument to `openloop`:

```
0-3> (let ((x 1))
         ((delta (e r k)
              (openloop "fox" r))))
fox-0: "bottom-level"
fox-1> x
fox-1: 1
fox-2>
```

Note: the function `reify-new-environment` provides a new reified instance of the initial environment, so that the form

```
(openloop "name")
```

is equivalent to

```
(openloop "name" (reify-new-environment))
```

But one does not always want to spawn a new level with a bottom level loop. The function `meaning` offers a general way of spawning a new level: it has as arguments an expression, an environment and a continuation. The expression is a standard Blond expression, the environment must be a reified environment, and the continuation can be any unary applicable object. A standard example is the identity function, where the reified environment and continuation are reinstalled:

[3]The name δ holds for *dynamic*. The logical letters for *lexical* or *static* would be λ or σ, which are already used in [Church 41] and [Felleisen & Friedman 87]. Since Blond is already an applied λ-calculus, and because we can define a σ-abstraction with a reifier to implement the language Λ_σ, we have choosen γ.

```
fox-2> ((delta (e r k)
          (meaning (car e) r k)) "hello world")
fox-2: "hello world"
fox-3>
```

In this scenario, the argument of the δ-abstraction has been reified in a list. The body of the reifier consists of querying the meaning of the argument in the reified environment, with the reified continuation – that is to say, the reified expression, environment and continuations are reflected back. This realizes the usual reflective definition of identity.

In the following interaction, the expression that is reified is not a string, but an identifier:

```
fox-3> (let ((x "hello world"))
         ((delta (e r k)
            (meaning (car e) r k)) x))
fox-3: "hello world"
fox-4>
```

In that scenario, the argument of the δ-abstraction is `x`, which is bound to the string `"hello world"` in the reified environment. The result is what we can expect from applying the identity function.

The first argument of `meaning` must evaluate to a reified expression. For simplicity, we chose to reify expressions as S-expressions, just as is expected in a Lisp-type setting:

- numerals and numbers are identified;
- syntactic and semantic strings are identified;
- identifiers and symbols are identified;
- any composed form written between parentheses is reified as the corresponding S-expression.

However, because of Scheme's and Blond's abstract syntax, we consider a restricted set of S-expressions, without dotted pairs. Expressions are reified as pure polymorphic lists, built out of the empty list.

This is for simplicity, because Blond is implemented in Scheme, where the identification between numerals and numbers, between syntactic and semantic strings, and between identifiers and symbols, is already effective. Further, because Blond programs are actually Scheme lists for the interpreter, reifying them as Blond lists is basically for free.

Thus in general, the first argument of `meaning` can be any well-formed list of atomic values[4]. The second has to be a reified environment. But the third may be any unary function:

```
fox-4> (meaning 1 (reify-new-environment) (lambda (x) x))
fox-4: 1
fox-5>
```

In this scenario, the expression is the number 1, the environment is a fresh one and the continuation is the identity function. The latter could be as well a primitive function:

[4]Examples of non-atomic values are function values, and more generally the operational value of any abstraction: λ and δ-abstrations, reifiers and control structures.

```
fox-5> (meaning 1 (reify-new-environment) add1)
fox-5: 2
fox-6>
```

In Blond, the behaviour of the third argument of `meaning` obeys the following equation:

$$\mathit{meaning}\, \epsilon\, \rho\, \kappa \quad \simeq \quad \mathit{apply}\, \kappa\, \epsilon\, \rho$$

This makes it impossible to get "inside the implementation" (whatever this means) as in 3-Lisp and in Brown, typically by specifying a reifier as a continuation. For example:

```
(meaning 'foobarbaz (reify-new-environment) quote)
```

merely returns `foobarbaz` at the same level, as it is the effect of `quote`-ing this identifier.

In 3-Lisp or in Brown, however, the expression is evaluated, and some continuation

$$\lambda v\, .\, \mathit{quote}(v)$$

is is applied to the result, leading to the bewildering result v in Brown and various other results in 3-Lisp, depending on the expression.

However this hygiene is restricted to environments and continuations. First-class reifiers and control structures offer a versatile access to expressions, as illustrated in the following scenario:

```
> (blond)
0-0: "bottom-level"
0-1> (define map
          (lambda (f l)  ; (Val -> Val) * List(Val) -> List(Val)
              ((rec self (lambda (l)
                             (if (null? l)
                                 '()
                                 (cons (f (car l)) (self (cdr l)))))) l)))
0-1: map
0-2> (map (lambda (x) x) '(1 2 3))
0-2: (1 2 3)
0-3> (map quote '(1 2 3))
0-3: ((car l) (car l) (car l))
0-4> (map (delta (e r k) e) '(1 2 3))
1-0: ((car l))
1-1>
```

where one gets what he deserves (remembering that reified expressions are values).

3.3 Conclusion

The design of Blond makes it possible to:

- describe reification and reflection formally, without fearing a reflective program that could get an insight in our semantics and process it (or why not side-effect it!);
- implement a reflective tower in something else than an interpreted Lisp-like language;
- finally it is a step towards compiling reflective programs.

This concludes the reflective capabilities offered in Blond. They are simple and formalizable (see [Danvy & Malmkjær 88]). Reification is offered through δ and γ-abstractions, and reflection is achieved by specifying either a one-shot evaluation, or a new interpreter with a bottom level loop. Successive reifications and reflections neutralize, as can be expected:

```
> (blond)
0-0: "bottom-level"
0-1> (let ((x 1))
         (meaning '((delta (e r k) x))
                  (reify-new-environment)
                  (reify-new-continuation "dummy")))
0-1: 1
0-2> (let ((x 1))
         ((delta (e r k)
              (meaning (car e) r k)) x))
0-2: 1
0-3> (blond-exit)
"farvel!"
>
```

This scenario illustrates that reflection followed by reification and reification followed by reflection leave the meta-continuation intact. [Danvy & Malmkjær 88] points out why this holds modulo an extension of the first environment in the meta-continuation. The reason is that such an environment memorizes the bindings of the formal parameters of any reifier that is applied.

[Danvy & Malmkjær 88] also discusses the non-compositionality of `meaning` and proposes a weaker, but compositional, function `meaning'`.

The two next sections analyze more precisely the properties of environments, continuations, and their reified representations.

4 Environments

There are conceptually three environments:

- a *common environment* mapping predefined or commonly defined identifiers to their value; this environment is called common because it is common to the levels: one common definition is visible from all the other levels;
- a *global environment* mapping the global identifiers of one level to their value; there is one global environment per level;
- the *lexical extension*, containing all the lexically bound identifiers – that is: the formal parameters of functions and the variables bound locally with the `let` construction and recursively with the `rec` and `letrec` constructions.

Note: variables defined in the common and in a global environment are recursively bound. Also, their scope is lexical.

Note': the global and the common environment are dynamic, as in Scheme.

4.1 Operations on the environments

These operations address creating or modifying a binding in the common environment, the global environment, and any lexical extension.

`(define <ide> <val>)` creates or modifies the binding of `<ide>` to `<val>` in the global environment of one level and returns `<ide>`.

`(common-define <ide> <val>)` creates or modifies the binding of `<ide>` to `<val>` in the common environment and returns `<ide>`.

`(let ((<ide-1> <val-1>) ... (<ide-n> <val-n>)) <body>)` extends the lexical environment and evaluates `<body>`.

`(letrec ((<ide-1> <val-1>) ... (<ide-n> <val-n>)) <body>)` extends the lexical environment recursively and evaluates `<body>`.

One can define a value recursively with `(rec <ide> <value>)`, that returns this value.

`(set! <ide> <val>)` modifies an already existing binding and returns the previous R-value. Worth to notice: `set!`-ting a binding of the common environment creates the modified binding in the global environment of the current level. It does not side-effect the binding in the common environment. Only `common-define` can create or modify a binding in the common environment of all the levels.

One can note that there is no "internal define" and that `define` operates on the global environment of one level. This is for the sake of simplicity, and also for symmetry with `common-define`, that operates on the common environment of all the levels.

4.2 Reified environments

They are obtained by applying a reifier, either a static one

```
(gamma (e r k) <body>)
```

or a dynamic one

```
(delta (e r k) <body>)
```

The second parameter is then bound to the reified environment, which is a functional object. Also, the zero-ary operator `reify-new-environment` returns a reified instance of the initial environment.

The rest of this subsection describes all the possible operations with a reified environment. They are similar in spirit to what environments are made for: they map identifiers to their R-value or their L-value; and they can be lexically extended.

Applying a reified environment to an identifier returns the value it is bound to in that environment or the symbol `***undefined***`:

$$\textit{reified-environment: Ide} \rightarrow \textit{Den-Val} \cup \{\texttt{***undefined***}\}$$

We can now define the function `exit` that exits from the current level. To define it simultaneously at all the levels of the tower, we use `common-define`:

```
(common-define exit     ; transmits a value from level n to level n+1
    (lambda (x)         ; Val_n -> Val_n+1
        ((delta (e r k)
            (r 'x)))))
```

`exit` is an unary function whose parameter is `x`. Its body consists of applying a reifier. The result is the value of the variable `x` in the reified environment `r`. This definition contrasts with Brown and [des Rivières 88] where the body of the reifier is evaluated in the environment of the level of definition: there is no variable capture (shadowing) in Blond.

One could remark that applying a reified environment and spawning a new level are redundant. This is true: when the identifier is bound, looking it up in the reified environment has the same effect as spawning a new level to evaluate that identifier.

```
> (blond)
0-0: "bottom-level"
0-1> ((delta (e r k) (common-define env r)))
1-0: env
1-1> (env 'x)
1-1: ***undefined***
1-2> (let ((x 'foobar))
         ((delta (e r k)
              (common-define env-x r))))
2-0: env-x
2-1> (env-x 'x)
2-1: foobar
2-2> (meaning 'x env-x (lambda (x) x))
2-2: foobar
2-3>
```

In this scenario, we start by commonly defining a reified environment. We find ourselves at level 1 with the variable `env` bound to a reified instance of the initial environment[5]. In this environment, the variable `x` is not bound. If it was evaluated at the bottom level, an error would occur. Here the reified environment is functional. Applying it to the identifier `x` returns the value `***undefined***`. We then commonly define a new reified environment, `env-x`, where the variable `x` is lexically bound to `foobar`. Applying `env-x` to `x` has the same effect as spawning a new level to query the current value of the variable `x` in the environment `env-x` with the identity continuation.

Reified environments are *variadic* [Strachey 67] in Blond: applying a reified environment to an identifier and a value will modify the binding of the identifier in that environment.

$$\textit{reified-environment: Ide} \times \textit{Den-Val} \rightarrow (\textit{Den-Val})_\perp$$

The effect is similar to spawning a new level and using `set!`:

```
2-3> (env-x 'x)
2-3: foobar
2-4> (env-x 'x 'foobarbaz)
2-4: foobar
2-5> (env-x 'x)
2-5: foobarbaz
2-6> (meaning '(set! x 'foo) env-x (lambda (x) x))
2-6: foobarbaz
2-7> (env-x 'x)
2-7: foo
2-8>
```

[5]For the same effect, but without leaving the level 0 we could have used the function `reify-new-environment`.

Note: `set!`-ing a common identifier will bind it globally, without changing its common binding. If one wants to redefine a common variable, he does it with `common-define`. This will not affect the value of the variable at the level where it has been `set!`.

The operations above allow to consult a reified environment and to modify it. To extend its global part, one needs an explicit new level:

```
2-8> (meaning '(define y x) env-x (lambda (x) x))
2-8: y
2-9>
```

The lexical extension of a reified environment is achieved with the operation:

```
(extend-reified-environment <list-of-ide> <list-of-val> <reified-env>)
```

Finally, for the better or for the worse, applying a reified environment to zero argument returns its underlying data structure: a list of dictionaries. Each element of the list pairs a list of names and an isomorphic list of values.

To summarize, the domain of reified environments is defined as:

$$(\mathit{Unit} \rightarrow (\mathit{Ide}^* \times \mathit{Den\text{-}Val}^*)^*) + (\mathit{Ide} \rightarrow \mathit{Den\text{-}Val} \cup \{\texttt{***undefined***}\}) + (\mathit{Ide} \times \mathit{Den\text{-}Val} \rightarrow (\mathit{Den\text{-}Val})_\perp)$$

5 Continuations

They represent the "rest of the computation" at one level, and can be reified, in the similar way as Scheme reifies continuations with `call-with-current-continuation` – [Talcott 85] calls that reification *noting* since it notes the current program point [Landin 65]: in another context [Smith & Hewitt 75] continuations were said to be *unpacked* or *captured*. This access to the implicit continuation contrasts with the well-known "continuation-passing style" [van Wijngaarden 66] [Fisher 72] [Reynolds 72] [Sussman & Steele 75] [Steele & Sussman 76] [Stoy 77] [Steele 78] where the continuation is kept explicit.

Reified continuations are obtained by applying a reifier, either a static one

```
(gamma (e r k) <body>)
```

or a dynamic one

```
(delta (e r k) <body>)
```

The third parameter is then bound to the reified continuation, which is functional and unary – this means that it can be applied to one argument, as in Scheme.

A major point: Blond keeps continuations "jumpy" [des Rivières 88]. That is, applying a reified continuation replaces the continuation that was active at that moment. This is the usual "black-hole" behaviour of Scheme, and contrasts with the usual "pushy" behaviour [Danvy & Malmkjær 88] taken in 3-Lisp and Brown. There, the local continuation, rather than being lost, is pushed onto the meta-continuation, to be reactivated next time the current level is abandoned.

This means that the tower is involved not only at reification or reflection time – that is, when a reifier is applied or when a new level is spawned – but also when a reified continuation is applied.

To separate clearly how the tower is managed and how reified objects are used, we have made continuations jumpy in Blond. A continuation is reified at some level L, and applying it merely substitutes the current level of processing with the level L.

But we do understand that this could be only a matter of taste. This is why, waiting for further results in the semantic investigation of reflective towers, a predicate and a toggle are provided in Blond:

```
(continuation-mode)
```

and

```
(switch-continuation-mode)
```

They return one of the two identifiers `pushy` or `jumpy` according to whether the mode is pushy or jumpy (resp. made pushy or jumpy).

Note: by default it is jumpy.

Note': changing of mode is of fundamental consequence for all the reflective programs that one has written. The most reasonable compromises that we have found so far practising Blond are:

- either to maintain two separate versions of each program according to the selected mode;
- or to make polling versions, that are compatible with the two continuation modes and continuously test the current mode;
- or to fix one's mind for a class of applications and simply program with one continuation mode; after all, this is complicated enough in itself without adding a further unstability factor.

To conclude: our main reason for keeping reified continuations jumpy is that we can push them with a regular call to `meaning`. It allows to achieve reification and reflection uniquely with reifiers and `meaning`.

Finally, for symmetry with having `reify-new-environment`, we provide the operator `reify-new-continuation`, whose functionality is:

$$\textit{Den-Val} + (\textit{Den-Val} \times \textit{Reified-Env}) \rightarrow \textit{Reified-Cont}$$

It reifies an initial bottom level loop, with an optional reified environment. The argument is the name of the new level. The optional reified environment will be the environment of the new bottom level loop. Let us illustrate it with a scenario:

```
> (blond)
0-0: "bottom-level"
0-1> ((reify-new-continuation "rock"
                               (extend-reified-environment '(foo)
                                                           '("bar")
                                                           (reify-new-environment))) "bottom")
rock-0: "bottom"
rock-1> foo
rock-1: "bar"
```

```
rock-2> ((reify-new-continuation "Multivac") "new bottom-level")
Multivac-0: "new bottom-level"
Multivac-1> ((delta (e r k) "bye"))
1-0: "bye"
1-1>
```

This session illustrates how level 0 is replaced successively by two other levels, by applying reified continuations realizing new bottom level loops. Exiting from the last ends up at level 1.

Let us define `openloop` with the operator `reify-new-continuation`.

```
(define openloop
    (delta (e r k)      ; List(RExp) * REnv * RCont -> Val
        (case (length e)
            (1
                (meaning (car e)
                         r
                         (lambda (level)
                             (meaning '(meaning "bottom-level"
                                                (reify-new-environment)
                                                (reify-new-continuation level))
                                      (extend-reified-environment '(level) (list level) r)
                                      k))))
            (2
                (meaning (car e)
                         r
                         (lambda (level)
                             (meaning (cadr e)
                                      r
                                      (lambda (env)
                                          (meaning '(meaning "bottom-level"
                                                             (reify-new-environment)
                                                             (reify-new-continuation level env))
                                                   (extend-reified-environment '(level env)
                                                                               (list level env)
                                                                               r)
                                                   k))))))
            (else
                (meaning "openloop: arity mismatch" r k)))))
```

By defining `openloop` as a reifier rather than a function, we can handle its variable number of arguments.

6 Elements for a Library

So far, there is no program written in Blond for any other purpose than practising it. So the Blond library is still growing. For example here are a couple of definitions, in both jumpy and pushy mode. The first is `call-with-current-continuation`, in jumpy mode:

```
(common-define call/cc
    (lambda (f)         ; (Val -> Val) -> Val
        ((delta (e r k)
            (meaning '(f dummy)
                     (extend-reified-environment '(dummy) (list k) r)
                     k)))))
```

Its argument is evaluated, and is expected to be a function. A reification followed by a reflection gives access to the continuation. The computation is continued with the application of the function to the reified continuation. The form to be evaluated is the application. The identifier `k` in the form is bound in a lexical extension of the reified environment. The identifier `f` is already bound in the reified environment. As one can note, this definition relies on the fact that reified continuations are already a proper sort of applicable values.

The following definition is only valid in pushy mode:

```
(common-define call/cc
    (lambda (f)         ; (Val -> Val) -> Val
        ((delta (e r k)
            (k ((r 'f) k))))))
```

This definition is more compact because the outermost application pushes back the current continuation on the meta-continuation, which is equivalent to spawning a new level. But it is not equivalent since the captured continuation is pushy. [Bawden 88] analyzes through a series of examples why pushy continuations are unsatisfactory, precisely because they push levels onto the meta-continuation, and how these levels interfere with later reifications and reflections.

The following scenario illustrates some aspects of programming with (jumpy) continuations, as in Scheme. What makes them quite clear are the Blond prompts: they witness the application of a continuation, when it starts a new instance of an earlier iteration in a bottom level loop.

```
> (blond)
0-0: "bottom-level"
0-1> (load "scheme.bl")
exit the-environment call/cc call/ce
0-1: "scheme.bl"
0-2> (continuation-mode)
0-2: jumpy
0-3> (add1 (call/cc (lambda (k) 3)))
0-3: 4
0-4> (add1 (call/cc (lambda (k) (k 3))))
0-4: 4
0-5> (add1 (call/cc (lambda (k) (sub1 (k 3)))))
0-5: 4
0-6> (call/cc (lambda (k) (common-define cont-0-6 k)))
0-6: cont-0-6
0-7> 'dummy   ; cont-0-6 is bound to the continuation of iteration 6 at level 0
0-7: dummy
0-8> (cont-0-6 "back to 0-6")
0-6: back to 0-6
0-7> (exit "exit from level 0")
1-0: "exit from level 0"
1-1> (cont-0-6 "back again to 0-6")
0-6: "back again to 0-6"
0-7> (exit "exit again from level 0")
2-0: "exit again from level 0"
2-1>
```

Let us analyze this interaction.

- First the noted continuation is not used. The effect is to return 3 to the function *add*1 – "to return" meaning "to apply the implicit continuation".
- Second the noted continuation is used. The effect is to explicitly apply the continuation to 3. The result again is 4. The only difference between these two expressions is in the implicit and the explicit applications of the continuation.

- At the iteration `0-5`, the noted continuation is composed with the function *sub*1. However the effect of applying a continuation is to drop the continuation active at that moment. Applying *sub*1 is not performed and 3 is directly passed to the continuation that starts with *add*1. Again the result is 4.
- At the iteration `0-6` the continuation is commonly bound to the identifier `cont-0-6`. This means that `cont-0-6` is simultaneously defined at all the levels of the tower, where not globally or lexically shadowed by an alias.
- Iteration `0-7` makes it explicit that the next iteration is `0-8`.
- At iteration `0-8` the continuation bound to `cont-0-6` is applied. The effect is to continue the bottom level loop of level 0 at iteration 6.
- At the new instance of iteration `0-7`, we exit from level 0. Because continuations are jumpy we arrive at level 1. If they were pushy, we would come back to the last point where a continuation has been applied, that is we would finish the iteration `0-8`.
- At level 1 the identifier `cont-0-6` is defined because it has been commonly defined and is not shadowed. We apply again its R-value, that is, the continuation of level 0, iteration 6. The effect is to continue the bottom level loop of level 0 at iteration 6.
- At the new instance of iteration `0-7`, we exit again from level 0. Because continuations are jumpy we arrive at level 2. If they were pushy, we would continue at level 0, iteration 9, and the same scenario with pushy continuations confirms this:

```
0-1> (mute-load "scheme.bl")
0-1: "scheme.bl"
0-2> (switch-continuation-mode)
0-2: pushy
0-3> (add1 (call/cc (lambda (k) 3)))
0-3: 4
0-4> (add1 (call/cc (lambda (k) (k 3))))
0-4: 4
0-5> (add1 (call/cc (lambda (k) (sub1 (k 3)))))
0-5: 4
0-6> (call/cc (lambda (k) (common-define cont-0-6 k)))
0-6: cont-0-6
0-7> 'dummy   ; cont-0-6 is bound to the continuation of iteration 6 at level 0
0-7: dummy
0-8> (cont-0-6 "back to 0-6")
0-6: "back to 0-6"
0-7> (exit "exit from level 0")
0-8: "exit from level 0"
0-9> (cont-0-6 "back again to 0-6")
0-6: "back again to 0-6"
0-7> (exit "exit again from level 0")
0-9: "exit again from level 0"
0-10> (exit 3)
0-5: 3
0-6> (exit 3)
0-4: 4
```

```
0-5> (exit "at last!")
1-0: "at last!"
1-1>
```

The point in this scenario is that each time a reified continuation is applied, the current continuation is stacked onto the meta-continuation, and that each time a level is exited, this ex-current continuation is popped from the meta-continuation and restored.

Jumping from branch to branch in the control tree can be compared with jumping from branch to branch in the environment tree – which is basic to programming in a lexically-scoped language, specially if it is higher-order (because abstractions close their environment of definition). In that sense, dynamic scoping consists of composing environment extensions and functional continuations [Felleisen *et al.* 87] consists of composing control extensions. Recent work proposes a more lexical composition of control extensions [Danvy & Filinski 88]. With the present analysis we try to relate the issues of control and environments, as [Landin 66] did for control and data.

Continuing to assemble elements for a Blond library, here is the function `the-environment` from CScheme (mitscheme), valid both in pushy and jumpy mode (it does not matter here because we do not apply any reified continuation):

```
(common-define the-environment
    (delta (e r k)      ; List(RExp) * REnv * RCont -> Val
        (meaning 'dummy
                 (extend-reified-environment '(dummy) (list r) r)
                 k)))
```

In pushy mode only, it looks like this:

```
(common-define the-environment
    (delta (e r k)      ; List(RExp) * REnv * RCont -> Val
        (k r)))
```

A more functional view of that function could be `call-with-current-environment`. In both jumpy and pushy mode:

```
(common-define call/ce
    (delta (e r k)      ; List(RExp) * REnv * RCont -> Val
        (meaning (car e)
                 r
                 (lambda (f)
                     (meaning '(f r)
                              (extend-reified-environment '(f r) (list f r) r)
                              k)))))
```

In pushy mode only:

```
(common-define call/ce
    (delta (e r k)      ; List(RExp) * REnv * RCont -> Val
        (meaning (car e) r (lambda (f)
                              (k (f r))))))
```

It clearly appears that definitions are shorter in pushy than in jumpy mode. This is of course due to the explicit call to `meaning` to spawn a level back. Time will show which one will prevail.

Part of the solution probably lies in the way pushy reified continuations are applied: (1) should their argument be evaluated with the current continuation stacked on the meta-continuation, as it would happen with a call to `meaning`? (2) or should they be evaluated with a constant meta-continuation and only when the reified continuation is applied, should the current continuation be stacked on the meta-continuation? Blond currently follows (1), but in its last developments (2) appears more appropriate [Danvy & Malmkjær 89].

Continuing to assemble elements for a Blond library, here is a function exiting as many levels as specified by its argument:

```
(common-define nexit
    (lambda (n)         ; Num_m -> Str_m'
        (if (<= n 0)
            "home"
            ((delta (e r k)
                (nexit (sub1 (r 'n))))))))
```

The functionality specifies that it takes an number n at level m and returns a string at level $m' = m + n$. The idea is to reify as many times as specified by its argument. It is interesting to stress how `nexit` iterates.

This function is recursively defined in the common environment, and thus is visible from all the levels of the tower. In particular it is defined at all the levels between the initial one and the target one.

At each iteration, there is a tail-recursive call. The argument is found *via* the reified environment. The formal parameter `n` is bound to the current value. By applying the reified environment to `n`, the current value is obtained, decremented, and passed to `nexit`.

Actually, these calls are tail-reflective, since `nexit` is applied tail-recursively at the level above.

The following scenario illustrates using `nexit`:

```
> (blond)
0-0: "bottom-level"
0-1> (load "nexit.bl")
nexit
0-1: "nexit.bl"
0-2> (nexit 256)
256-0: "home"
256-1> (nexit 64)
320-0: "home"
320-1> (nexit 8)
328-0: "home"
328-1> (nexit 0)
328-1: "home"
328-2>
```

Actually, we do not have to define `nexit` in the common environment. It is sufficient to define it at one level, and pass it around as argument. Of course, at the target level, the function is not defined.

This is achieved with the following definition:

```
(define lexit
    (lambda (n)         ; Num_m -> Str_m'
        (let ((self (lambda (self n)
                        (if (<= n 0)
                            "home"
                            ((delta (e r k)
                                ((r 'self) (r 'self) (sub1 (r 'n)))))))))
            (self self n))))
```

Now we are sufficiently acquainted to define a function that permutes two arbitrary levels above in the tower. It is best illustrated by a scenario:

```
> (blond)
0-0: "bottom-level"
0-1> (load "swap.bl")
swap! get-up! got-up! got-down! get-down! kwote bye nexit
0-1: "swap.bl"
0-2> (swap! 2 1)     ; permutes the first level and the second level above
0-2: "done!"
0-3> (bye)           ; exit from level 0
2-0: "bye"
2-1> (bye)           ; exit from level 2
1-0: "bye"
1-1> (bye)           ; exit from level 1
3-0: "bye"
3-1> (bye)           ; exit from level 3
4-0: "bye"
4-1>
```

given the reifier

```
(common-define bye
    (delta (e r k)   ; List(RExp) * REnv * RCont -> Str
        "bye"))
```

This scenario can be expressed graphically by:

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

The `swap!` operator generalizes the `permute!` operator from [Danvy & Malmkjær 88] since it can permute any arbitrarily levels. Let us use `nexit` to illustrate it:

```
> (blond)
0-0: "bottom-level"
```

```
0-1> (load "swap.bl")
swap! get-up! got-up! got-down! get-down! kwote bye nexit
0-1: "swap.bl"
0-2> (swap! 85 133)  ; permute the 85th level and the 133rd level above
0-2: "done!"
0-3> (bye)           ; above level 0 there is still level 1
1-0: "bye"
1-1> (nexit 83)      ; exit 83 levels
84-0: "home"
84-1> (bye)          ; exit one level more, to level 133
133-0: "bye"
133-1> (bye)         ; exit from level 133 to level 86
86-0: "bye"
86-1> (bye)          ; above level 86 there is level 87
87-0: "bye"
87-1> (nexit 45)     ; exit 45 levels
132-0: "home"
132-1> (bye)         ; exit one level more, to level 85
85-0: "bye"
85-1> (bye)          ; above level 85 there is level 134
134-0: "bye"
134-1> (nexit 166)   ; and 166 levels above level 300 is present
300-0: "home"
300-1>
```

Roughly, the idea is to iterate upwards and collect the environments and continuations, up to the higher level, and then to iterate downwards, restoring environments and continuations. We use a set of common definitions and check first the consistency of the arguments:

```
(common-define swap!
    (lambda (n o)       ; Num * Num -> Str
        (cond
            ((or (< n 0) (< o 0))
                "foobar")
            ((< n o)
                (get-up! n (- o n) '()))
            ((> n o)
                (get-up! o (- n o) '()))
            (else
                "already done"))))
```

The function `get-up!` reifies iteratively up to the first level to permute:

```
(common-define get-up!
    (lambda (n o l)     ; Num * Num * List(Pair(REnv, RCont)) -> Str
        (if (zero? n)
            ((delta (e r k)
                (got-up! (r 'o) (cons (cons r k) (r 'l)) '())))
            ((delta (e r k)
                (get-up! (sub1 (r 'n)) (r 'o) (cons (cons r k) (r 'l))))))))
```

The arguments of `get-up!` are transmitted using the same device as for `nexit`. They are: the number of levels to traverse for reaching the closest level to permute; the number of levels to traverse for reaching the farthest level; a list of pairs mimicking the meta-continuation, *i.e.*, holding the reified environments and continuations of all the levels down to the original one. When the closest level is reached, the function `got-up!` is applied to the number of other levels to traverse, the list of all the environments and continuations of the levels down, starting with the environment and continuation of the level to permute, and an empty list to hold all the environments and continuations of the level above, up to the farthest level to permute.

```
(common-define got-up!
    (lambda (o l ll)    ; Num * List(Pair(REnv, RCont)) * List(Pair(REnv, RCont)) -> Str
        (if (zero? o)
            (meaning (list 'got-down! (kwote (cons (car ll) (cdr l))) (kwote (cdr ll)))
                     (caar l)
                     (cdar l))
            ((delta (e r k)
                (got-up! (sub1 (r 'o)) (r 'l) (cons (cons r k) (r 'll))))))))
```

The function `got-up!` iterates upwards up to the farthest level to permute, using the same strategy as `get-up!`. When this level is reached, a new level is spawned with the environment and continuation of the lowest level. The expression to evaluate is built up to call the function `got-down!` with two arguments: the list of environments and continuations to restore down to the lowest level to permute; and the list of environments and continuations to restore down to the original level. We build the expression with quoted values for convenience:

```
(common-define kwote
    (lambda (x)         ; Val -> RExp
        (list 'quote x)))
```

but the environment spawned could as well be extended with dummy identifiers bound to these values. It does make a difference though since all these bindings will extend the environments in the meta-continuation, shadowing the other bindings. The corresponding problems of compositionality and referential transparency are discussed further in [Danvy & Malmkjær 89].

```
(common-define got-down!
    (lambda (l ll)      ; List(Pair(REnv, RCont)) * List(Pair(REnv, RCont)) -> Str
        (if (null? ll)
            (get-down! l)
            (meaning (list 'got-down! (kwote l) (kwote (cdr ll))) (caar ll) (cdar ll)))))
```

The function `got-down!` spawns levels down to the lowest level to permute, restoring their environments and continuations[6]. The function `get-down!` spawns levels down to the original level:

```
(common-define get-down!
    (lambda (l)         ; List(Pair(REnv, RCont)) -> Str
        (if (null? l)
            "done!"
            (meaning (list 'get-down! (kwote (cdr l))) (caar l) (cdar l)))))
```

This concludes some first elements for a Blond library. What we have learned assembling them is to practise meta-level facilities. Many other examples are to be found in [Danvy & Malmkjær 89], in an extended dialect of Blond.

[6]Yes, it would be more concise to use backquote.

7 Blond in Blond, or the Orthogonality of Reflective Towers

Blond is specified in Scheme, and defines a Scheme interpreter. We have specified it in the non-reflective part of the language it can interpret – that is, our specification of Blond is meta-circular.

Thus one can load the Blond interpreter in a Blond session, and start a new Blond session. The effect is to create a reflective tower orthogonal to the current one. The process can be iterated, of course at the price of speed.

However it illustrates the intensional view of primitive functions introduced in [Danvy 88]:

```
>>> (load "blond.scm")
t
>>> (blond)
0-0: "bottom-level"
0-1> (mute-load "scheme.bl")
0-1: "scheme.bl"
0-2> (mute-load "blond.scm")
0-2: "blond.scm"
0-3> (call/ce
         (lambda (r)
             (openloop "blond" r)))
blond-0: "bottom-level"
blond-1> (blond)
0-0: "bottom-level"
0-1> car
0-1: (subr 1 (subr 1 <CLOSURE>))
0-2> '(1 2 3)
0-2: (1 2 3)
0-3> (car '(1 2 3))
0-3: 1
0-4> (blond-exit)
blond-1: "farvel!"
blond-2> car
blond-2: (subr 1 <CLOSURE>)
blond-3> (blond-exit)
"farvel!"
>>> car
<CLOSURE>
>>>
```

This scenario illustrates a Blond session in a Blond session under Scheme.

- During a Scheme session, the file `blond.scm` is loaded. It contains the Scheme specification of Blond.
- A Blond session is then started, by applying the function `blond`. We load the file `scheme.bl`, defining in particular the `call-with-current-environment` described in the previous section.
- At the second iteration, we load the file `blond.scm` in Blond. It is possible to load it and even to process Blond in Blond because the system is written meta-circularly and non-reflectively.
- We start a new level `blond`, noting (or: capturing) the current environment, where Blond is defined. This has no other purpose than changing the prompt.

- At the new level, we start a Blond session by applying the function `blond`. The effect is to create a reflective tower orthogonal to the current one. Scheme, Blond and Blond form a non-reflective tower, because Blond is specified non-reflectively. It is a simple tower of interpreters whose height is three.
- We evaluate `car`. It is bound to the usual primitive function *car*, that is: the first projection of a Blond pair. In the Blond model, the domain of applicable objects is a direct sum of the domains of primitive functions, abstractions, *etc.* – *car* is a primitive function. The injection tag is `subr`. The domain of primitive functions is itself a direct sum of zero-ary, unary and binary functions. Presently, `1` is another injection tag indicating that *car* is unary. The third component is the function itself: it is the function *car* at the level above. Since that level is Blond, *car* belongs to the domain of unary primitive functions, which is mirrored by its injection tags. The next processor running Blond presently is Scheme 84: it represents primitive functions as `<CLOSURE>`.
- The session is ended by applying `blond-exit`. We come back to the first Blond interpreter. We check whether `car` is bound to the primitive function `car`, which it is. The Blond session is ended, to come back to Scheme.
- We finally check whether `car` is bound to the primitive function *car*, which it is.

This view of primitive functions alongside a tower of interpreters is intensional because it illustrates the operational connection between the levels of interpretation – that is:
(1) the request of processing a primitive operation from one interpreter to the one above; and
(2) the answer to that service.

The latter point is insured by the continuation of the level above, where it figures in the remaining part of the computation.

This intensional view, together with the tiling game of [des Rivières & Smith 84], points out the cost of a meta-interpretation. It justifies the current efforts in the domain of partial evaluation for breaking down the resulting orders-of-magnitude loss of efficiency. This is achieved by specializing the meta-interpreters with respect to their programs.

[Danvy 88] analyzes further the relationships between reflection and partial evaluation, on the basis that they both involve a program and its interpreter: reflective procedures in a program reify internal structures in the interpreter; partial evaluation of an interpreter consists of specializing it with respect to a program. The paper investigates the intriguing similarities between the identity of the meta-circular interpreters at each level of the reflective tower and the identity of the subject language and the base language in a partial evaluator. Both are essential: it is because the levels of the tower are all alike that simple and reflective procedures are expressed in the same language[7] and it is because a partial evaluator is written in the language it partially evaluates that it can be self-applied and thus generate compilers and compiler generators rather than merely perform compilations [Futamura 71] [Ershov 77] [Jones *et al.* 85] [Jones *et al.* 88].

Conclusion

Blond is a reflective dialect of Scheme, offering the standard, non-reflective, Scheme facilities and the expressive power of a reflective tower. This primer provides a brief description of the general

[7]On the other hand, it is also because the levels are all alike that it is possible to implement an infinite tower with a finite machine.

framework of meta-level capacities and a more detailed description of the Blond system and how to use it. Summarizing the characteristics of Blond:

- control structures are pre-defined, and thus do not need to be defined reflectively;
- environments are kept separated; there is one environment common to all the levels of the tower, one global environment per level, and as many lexical extensions as necessary;
- there are two sorts of reifiers: δ and γ-abstractions; their body is evaluated in the environment of the level above their application or their definition (resp.);
- the Blond semantics is closed: a program cannot "get inside the implementation" by specifying a reifier where functions could be expected;
- reification occurs in a correct environment, *i.e.*, of the level above;
- the specification is meta-circular: this allows to build orthogonal reflective towers;
- the specification is non-reflective: this means that Blond running in Blond creates a non-reflective tower of interpreters;
- the implementation is single-threaded [Schmidt 85], on a model similar to Brown;
- the specification is λ-lifted [Johnsson 85], whereas Brown is totally curried;
- Blond offers a system of prompts witnessing both the current level in the tower and the current iteration in the bottom level loop; this proves valuable when programming with continuations.

Currently, we are translating Blond to other systems than Scheme, and practising it. We also continue to investigate the semantics of reflective towers, in the line of [Danvy & Malmkjær 88]. In [Danvy & Malmkjær 89], for example, we propose a new approach for formalizing computational reflection.

Glossary

The following is a very sketchy review of the Scheme values that figure in the Blond initial environment. They are the usual Scheme primitive functions. In addition, control structures are present in the initial environment since they are first-class. The last lines concern the reflective extension.

```
nil,
car, cdr, caar, cadr, cdar, cddr, caddr, cdddr, list,
cons, last-pair, list-tail, set-car! set-cdr!,
null?, atom?, pair?, number?, string?, symbol?, equal?, boolean?, procedure?
zero?, add1, sub1, +, -, *, =, negative?, positive?, <, <=, >=, >,
not, length, member,
if, ef, case, and, or, begin, cond,
display, print, pretty-print, newline, flush-output,
load, mute-load, read,
open-input-file, eof-object?, close-input-port,
common-define, define, set!,
let, letrec, rec, let*,
lambda, quote, delta, gamma, meaning, openloop,
reify-new-continuation, reify-new-environment, extend-reified-environment,
continuation-mode, switch-continuation-mode,
blond-exit
```

References

[Bawden 88] Alan Bawden: *Reification without Evaluation*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming pp 342-351, Snowbird, Utah (July 1988)

[Church 41] Alonzo Church: *The Calculi of Lambda-Conversion*, Princeton University Press, Princeton, New Jersey (1941)

[Danvy 88] Olivier Danvy: *Across the Bridge between Reflection and Partial Evaluation*, in *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov and N. D. Jones (eds.), North-Holland (1988)

[Danvy & Filinski 88] Olivier Danvy, Andrzej Filinski: *A Functional Abstraction of Typed Contexts*, article submitted for publication, DIKU, University of Copenhagen, Copenhagen, Denmark (June 1988)

[Danvy & Malmkjær 88] Olivier Danvy, Karoline Malmkjær: *Intensions and Extensions in a Reflective Tower*, Proceedings of the 1988 ACM Symposium on LISP and Functional Programming pp 327-341, Snowbird, Utah (July 1988)

[Danvy & Malmkjær 88'] Olivier Danvy, Karoline Malmkjær: *A Blond Primer*, DIKU Rapport 88/21, DIKU, University of Copenhagen, Copenhagen, Denmark (October 1988)

[Danvy & Malmkjær 89] Olivier Danvy, Karoline Malmkjær: *Aspects of Computational Reflection in a Programming Language*, article submitted for publication, DIKU, University of Copenhagen, Copenhagen, Denmark (October 1988)

[des Rivières & Smith 84] Jim des Rivières and 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*, in *Meta-Level Architectures and Reflection*, Patti Maes and Daniele Nardi (eds.), North-Holland (1988)

[Ershov 77] Andrei P. Ershov: *On the Partial Computation Principle*, Information Processing Letters, Vol. 6, No 2 pp 38-41 (April 1977)

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

[Felleisen & Friedman 87] Matthias Felleisen, Daniel P. Friedman: *A Syntactic Theory of Sequential State*, Technical Report No 230, Computer Science Department, Indiana University, Bloomington, Indiana (October 1987)

[Fisher 72] Michael J. Fisher: *Lambda Calculus Schemata*, Proceedings of the ACM conference *Proving Assertions about Programs* pp 104-109, SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14 (January 1972)

[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)

[Futamura 71] Yoshihiko Futamura: *Partial Evaluation of Computation Process – an Approach to a Compiler-Compiler*, Systems, Computers & Control Vol. 2, No 5 pp 45-50 (1971)

[Johnsson 85] T. Johnsson: *Lambda Lifting: Transforming Programs to Recursive Equations*, Proceedings of the Conference on Functional Languages and Computer Architecture, Lecture Notes in Computer Science No 201 pp 190-203, Jean-Pierre Jouannaud (ed.), Springer-Verlag, Nancy, France (September 1985)

[Jones *et al.* 85] Neil D. Jones, Peter Sestoft, Harald Søndergaard: *An Experiment in Partial Evaluation: the Generation of a Compiler Generator*, Proceedings of the first International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science No 202 pp 124-140, Jean-Pierre Jouannaud (ed.), Springer-Verlag, Dijon, France (June 1985)

[Jones *et al.* 88] Neil D. Jones, Peter Sestoft, Harald Søndergaard: *MIX: a Self-Applicable Partial Evaluator for Experiments in Compiler Generation*, Vol. 1, Nos 3/4 of the International Journal *LISP and Symbolic Computation* (1988)

[Landin 65] Peter J. Landin: *A Correspondance between ALGOL 60 and Church's Lambda Notation*, CACM Vol. 8, No 2 pp 89-101 & No 3 pp 158-165 (February & March 1965)

[Landin 66] Peter J. Landin: *The Next 700 Programing Languages*, CACM Vol. 9, No 3 pp 157-166 (March 1966)

[Malmkjær 88] Karoline Malmkjær: *The Reflective Tower*, student project ???, Computer Science Department, University of Copenhagen, Copenhagen, Denmark (November 1988)

[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)

[Reynolds 72] John Reynolds: *Definitional Interpreters for Higher-Order Programming Languages*, Proceedings 25th ACM National Conference pp 717-740, New York (1972)

[Schmidt 85] David A. Schmidt: *Detecting Global Variables in Denotational Definitions*, ACM Transactions on Programming Languages and Systems, Vol. 7, No 2 pp 299-310 (April 1985)

[Smith & Hewitt 75] Brian C. Smith, Carl Hewitt: *A PLASMA Primer*, rough draft, MIT-AIL, Cambridge, Massachusetts (October 75)

[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)

[Smith & des Rivières 84] Brian C. Smith, Jim des Rivières: *Interim 3-LISP Reference Manual*, Intelligent Systems Laboratory, Xerox PARC, Palo Alto, California (1984)

[Steele & Sussman 76] Guy L. Steele Jr., Gerald Jay Sussman: *Lambda, the Ultimate Imperative*, MIT-AIL, AI Memo No 353, Cambridge, Massachusetts (March 1976)

[Steele 78] Guy L. Steele Jr.: *RABBIT: a Compiler for SCHEME (A Study in Compiler Optimization)*, MIT-AIL, TR 474, MIT, Cambridge, Massachusetts (May 1978)

[Stoy 77] Joseph E. Stoy: *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, The MIT Press (1977)

[Strachey 67] Christopher Strachey: *Fundamental Concepts in Programming Languages*, International Summer School in Computer Programming, Copenhagen, Denmark (1967)

[Sussman & Steele 75] Gerald Jay Sussman, Guy L. Steele Jr.: *SCHEME: an Interpreter for Extended λ-Calculus*, MIT-AIL, AI Memo No 349, Cambridge, Massachusetts (December 1975)

[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)

[van Wijngaarden 66] A. van Wijngaarden: *Recursive Definition of Syntax and Semantics*, from *Formal Languages Description Languages for Computer Programming* pp 13-24, T. B. Steel Jr. (ed.), North-Holland (1966)

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

[Wand & Friedman 88] Mitchell Wand, Daniel P. Friedman: *The Mystery of the Tower Revealed: a Non-Reflective Description of the Reflective Tower*, Volume 1, No 1 pp 11-38 of the International Journal *Lisp and Symbolic Computation* (June 1988)

Appendix – The Blond Listing in Scheme

The following is a Scheme-type interpreter in continuation-passing style. An extra argument – the meta-continuation – is carried all along. It holds both environments and continuations of all the levels above the current one. At reification time, the bottom-most ones are popped off and activated. At reflection time, the current ones are pushed on. The illusion of having an infinite meta-continuation is achieved by building it lazily.

Programs are represented as Scheme lists. Numbers, strings, and identifiers are represented as Scheme numbers, strings, and identifiers. Control structures, functions, primitive functions, reifiers, and reified environments and continuations are first-class applicable objects. The apply module dispatches on their injection tag.

The environment is a series of pairs of series of identifiers and of values – that is, it is a series of dictionaries. It is reified as its procedural abstraction. The common environment is represented with two lists of common names and common values. The initial environment is the common environment.

Syntactic correctness is generally not checked (apart the arities). It is assumed that all Blond expressions are well-formed.

There are two classes of errors: the ones caught by the Blond interpreter and the ones caught by the Scheme implementation. An example of Blond error is typically an arity error. An example of Scheme error is to take the `car` of the empty list.

In case of Scheme error, the Blond session is interrupted and one is back to Scheme. In case of Blond error, the function `wrong` is applied. It displays its complaints and stops the Blond session, back to Scheme.

Recovering from an error is simple: restarting Blond. Only the common environment and the continuation mode are persistent. The global environment of each level and the lexical extensions are lost.