FUNCTIONAL PEARL

Embedded Interpreters

NICK BENTON
Microsoft Research
7 J J Thomson Avenue
Cambridge CB3 0FB
United Kingdom
(*e-mail:* `nick@microsoft.com`)

Abstract

We describe a technique for embedding interpreters in statically-typed functional programs, by which (higher-order) values in the interpreting language may be embedded in the interpreted language and values from the interpreted language may be projected back into the interpreting one. The idea is introduced with SML code for the command-line interface to a tactical theorem prover applet and then extended to languages with recursive types and applied to elementary meta-programming. We then show how the method combines with Filinski's continuation-based monadic reflection operations to define an 'extensional' version of the call-by-value monadic translation and hence to allow values to be mapped bidirectionally between the levels of an interpreter for a functional language parameterized by an arbitrary monad. Finally, we show how SML functions may be embedded into, and projected from, an interpreter for an asynchronous pi-calculus via an 'extensional' variant of a standard translation from lambda into pi.

1 Introduction

Many programs incorporate one or more 'little languages' which they have to parse and interpret, typically to provide a top-level interactive loop or a scripting interface. Given that nearly all introductions to functional programming include both a parser combinator library and at least one evaluator for lambda expressions, one might reasonably think that nothing remains to be said about how to embed an interpreter for an application-specific little language into a functional program. But most interpreters in the literature keep values (and types) from the interpret*ing* and interpret*ed* languages quite separate. Typically, a few hardwired, first-order primitives for arithmetic or IO are implemented in terms of their counterparts in the interpreting (meta)language, but the assumption is that interpreted programs will essentially be entirely written in the interpreted (object) language. When the interpreter is for a domain-specific language, however, one usually has a rich collection of application-specific datatypes and higher-type values written in the interpreting language; the purpose of the little language being to allow the user to glue these complex things together flexibly at runtime. Then one needs a way to lift arbitrary values from the metalanguage in which the application is written into the domain-specific object language, which is particularly challenging when the metalanguage has strong static types. That is the problem addressed here. Our solution combines several known advanced functional programming techniques in a novel way.

We begin with a real example – the interface for a theorem-prover applet – for which the interpreted language is particularly simple.

2 The Hal Applet

Hal is a tactical theorem prover for first-order logic written by Paulson (1991). As one of the examples for MLj (Benton *et al.*, 1998), we have compiled Hal into an applet which runs in any Java-enabled web browser.[1] Like many ML programs, Hal does not include any code providing an interactive user interface: it is intended to be used from within the top-level read-eval-print loop of an interactive ML system. This, of course, is just how ML was originally intended to be used: as the Meta Language for theorem provers. But MLj does not have an interactive top-level loop and, in any case, it would seem a little excessive to download a complete ML environment into a browser just to provide an interface to a program of a few hundred lines.

So we had to add some simple user-interface code to Hal. One difficulty was that Java applets do not get a simple, scrolling 'green screen' text interface by default. We solved that problem by downloading a third-party terminal emulator written in Java and linking that to the ML code using MLj's interlanguage working extensions. The more interesting issue was how to design and implement a little command language by which the user could actually prove theorems. To give a flavour of the problem, here is a trivial interactive session with Hal:

```
- goal "P --> P & P & P & P & P";           (* initial goal *)
 1. empty  |-  P --> P & (P & (P & (P & P)))

- by (impR 1);                              (* implication right
 1. P  |-  P & (P & (P & (P & P)))             on 1st subgoal  *)

- by (conjR 1);                             (* conjunction right
 1. P  |-  P                                   to 1st subgoal
 2. P  |-  P & (P & (P & P))                   gives 2 subgoals *)

- by (repeat ((conjR 1) || (basic 1)));     (* repeat conj-right
P --> P & (P & (P & (P & P)))                  or axiom finishes
No subgoals left!                              the proof *)
```

The Hal code already pretty-prints its output imperatively, so we just have to deal with processing user input. Examining the session above, we seem to have a

[1] Admittedly, the number of people who *want* to do interactive theorem proving in their web-browser is probably quite small, but we still thought it made a nice demo.

simple combinatory calculus involving strings, integers and a rich collection of built in identifiers (some of which, like `||`, appear infix). Here are a few extracts from signatures of top-level structures in Hal:

```
signature RULE = sig
  type state
  type tactic = state -> state ImpSeq.t
  val basic: int -> tactic
  val conjL: int -> tactic
  val disjL: int -> tactic
  ...            end

signature COMMAND = sig
  val goal: string -> unit
  val by: Rule.tactic -> unit
  ...               end

signature TACTICAL = sig
  type ('a,'b) multifun = 'a -> 'b ImpSeq.t
  val || : ('a,'b) multifun * ('a,'b) multifun -> ('a,'b) multifun
  val repeat : ('a,'a) multifun -> ('a,'a) multifun
  ...                end
```

(`Rule.state` is the abstract type of proof states. `ImpSeq.t` is a type constructor for potentially-infinite lazy streams.)

Using Paulson's parser combinators (which are already used in Hal to parse logical formulæ), it takes less than twenty lines to implement a parser

```
val read : string -> Exp
```

which will take user input like that used in our example session to elements of the following AST datatype:

```
datatype Exp = EId of string   (* identifiers *)
             | EI of int       (* integer consts *)
             | ES of string    (* string consts *)
             | EApp of Exp*Exp (* application *)
             | EP of Exp*Exp   (* pairs *)
```

The problem is then to embed the thirty-or-so ML values which make up Hal's basic vocabulary into an interpreter for elements of the `Exp` datatype.

3 Embedding Values In An Interpreter

The types of Hal commands are built from `int`, `string`, `tactic` and `unit` by products and function spaces. We interpret expressions of type `Exp` as elements of a universal ML datatype modelling an untyped lambda calculus with pairing built over those basetypes:

```
datatype U = UF of U->U | UP of U*U | UUnit |
             UI of int | US of string | UT of tactic
```

The novel observation is that such a meta-circular interpreter (Reynolds, 1998), which uses functions to interpret functions, allows us to link the interpreted language and the interpreting language in a particularly neat way.

The trick is to define a type-indexed family of pairs of functions which *embed* ML values into the type `U` and *project* values of type `U` back into ML values. Here is the relevant part of the signature:

```
signature EMBEDDINGS =
sig
  type 'a EP
  val embed   : 'a EP -> ('a->U)
  val project : 'a EP -> (U->'a)

  val unit   : unit EP
  val int    : int EP
  val string : string EP
  val tactic : tactic EP
  val **     : ('a EP)*('b EP) -> ('a*'b) EP
  val -->    : ('a EP)*('b EP) -> ('a->'b) EP
end
```

and here is the matching part of the corresponding structure:

```
structure Embeddings :> EMBEDDINGS =
struct
  type 'a EP = ('a->U)*(U->'a)
  fun embed (e,p) = e
  fun project (e,p) = p

  fun cross (f,g) (x,y) = (f x,g y)
  fun arrow (f,g) h = g o h o f

  fun PF (UF(f))=f  (* : U -> (U->U) *)
  fun PP (UP(p))=p  (* : U -> (U*U)  *)
  (* similar definitions of PI, PS, PU and PT *)

  infixr --> infix **
  val unit   = (UUnit,PU)
  val int    = (UI,PI)
  val string = (US,PS)
  val tactic = (UT,PT)
  fun (e,p)**(e',p') = (UP o cross(e,e'), cross(p,p') o PP)
  fun (e,p)-->(e',p') = (UF o arrow (p,e'), arrow (e,p') o PF)
end
```

For any ML type A, an (A `EP`)-value is a pair of an embedding of type A`->U` and a projection of type `U->`A. The interesting part of the definitions of the combinators on embedding/projection pairs is the case for function spaces: given a function from A to B, we turn it into a function from `U` to `U` by precomposing with the projection for A and postcomposing with the embedding for B (this is why embeddings and projections are defined simultaneously). The resulting function can then be made into an element of `U` by applying the `UF` constructor. Projecting an (appropriate) element of `U` to a function type A`->`B does the reverse: first strip off the `UF` constructor and then precompose with the embedding for A and postcompose with the projection for B.

Note that the projection functions are partial – they will raise a match exception if given an argument in the wrong summand of the universal type `U`.[2] For any ML type A which is built from the chosen base types by products and function spaces, the embedding for A followed by the projection for A will be the identity. Going the other way, following the projection with the embedding, generally yields a more undefined (raising more `Match` exceptions) value than the one you started with.[3]

Using embedding/projection pairs, we can map the values from the top-level structures in Hal into the datatype `U`. The resulting values are then paired with their names in a top-level environment which is accessed by the interpreter:

```
val rules = map (cross (I, (embed (int-->tactic))))
             [("basic", Rule.basic), ("conjL", Rule.conjL),...]
val comms = [("goal", embed (string-->unit) Command.goal),
             ("by", embed (tactic-->unit) Command.by)]
val tacs = [("||", embed (tactic**tactic-->tactic) Tactical.||),
            ("repeat", embed (tactic-->tactic) Tactical.repeat),...]
val builtins = rules @ comms @ tacs
```

The interpreter itself, which takes a user expression of type `Exp` and evaluates it to a value of type `U`, is just a denotational semantics:

```
fun interpret e = case e of
   EI n => UI n
 | ES s => US s
 | EId s => lookup s builtins
 | EP (e1,e2) => UP(interpret e1,interpret e2)
 | EApp (e1,e2) => let val UF(f) = interpret e1
                         val a = interpret e2
                     in f a
                     end
```

Now, evaluating an expression like

[2] Although these exceptions *should* be caught, we will omit all error handling for reasons of space and clarity.

[3] If one replaced the exception-throwing with nontermination, then `embed o project` would be less than or equal to the identity in the conventional domain-theoretic sense.

```
interpret (read "goal \"P & Q --> Q & P\"")
```

will set the initial prover state, whilst

```
interpret (read "by (impR 1)")
```

takes a proof step. The Hal applet top level loop therefore just has to repeatedly read a line from the console, parse it to yield an `Exp`, call `interpret` and print the result (since Hal commands all change and print the proof state imperatively, the last stage is not strictly necessary). Using the applet is then almost identical to using Hal from within an interactive ML system, except that type errors result in 'runtime' exceptions, rather than being caught statically.

4 Extensions And Further Applications

We will now assume our object language has been extended with lambda abstraction, `let val`-bindings, recursive `let fun` function bindings, conditionals and infix operators. The revised abstract syntax datatype looks like this:

```
datatype Exp = ... | ELet of string*Exp*Exp | EIf of Exp*Exp*Exp
             | ELam of string*Exp | ELetfun of string*string*Exp*Exp
```

and we show some of the revised interpreter in Figure 1. This is essentially standard, though the reader will notice that the interpreter is written in a staged style, taking an expression and a *static* environment (a `string list`) and returning a function taking as input a matching *dynamic* environment (a `U list`) and returning a `U` value. This kind of *binding-time separation* is well-known in the partial evaluation community (Jones *et al.*, 1993), but is perhaps somewhat underappreciated elsewhere. Compared with a more naïve interpreter (taking a single environment of type `string*U list` or `string->U`), the staged version is both more informative and more efficient, since the abstract syntax tree is only traversed once to yield what amounts to a form of threaded interpreted code.[4]

4.1 Projection and Quoting

Embedding/projection pairs allow one to do considerably more than just run end-user commands which manipulate values from the application as part of a top-level loop. Being able to `project` as well as `embed` means that object level expressions may be interpreted and then projected back down to ML values for use in subsequent computation. At its simplest, this means that we can do something like

```
- let val eSucc = interpretclosed (read "fn x=>x+1")
      val succ = project (int-->int) eSucc
  in (succ 3) end;
val it = 4 : int
```

[4] This really does turn out to be worthwhile. For example, using SML/NJ 110, calculating `fib 27` using an explicitly-defined Y combinator is about five times faster using separated environments than it is with association-list environments and no staging.

```
type staticenv = string list
type dynamicenv = U list
fun indexof (name::names, x) = if x=name then 0 else 1+(indexof(names, x))

(* val interpret : Exp*staticenv -> dynamicenv -> U *)
fun interpret (e,static) = case e of
   EI n => K (UI n)
 | EId s => (let val n = indexof (static,s)
             in fn dynamic => List.nth (dynamic,n)
             end handle Match => let val lib = lookup s builtins
                                 in K lib
                                 end)
 | EApp (e1,e2) => let val s1 = interpret (e1,static)
                       val s2 = interpret (e2,static)
                   in fn dynamic => let val UF(f) = s1 dynamic
                                        val a = s2 dynamic
                                    in f a
                                    end
                   end
 | ELetfun (f,x,e1,e2) =>
                    let val s1 = interpret (e1, x::f::static)
                        val s2 = interpret (e2,f::static)
                        fun g dynamic v = s1 (v::UF(g dynamic)::dynamic)
                    in fn dynamic => s2 (UF(g dynamic)::dynamic)
                    end
 | ... other clauses elided ...

fun interpretclosed e = interpret (e,[]) []
```

Fig. 1. The Interpreter with Separated Environments (extract)

But projecting a *fixed* expression from the object language to the metalanguage is rarely useful. More interesting are the cases in which the object language expression is either constructed or read in at run-time.

In the case of the Hal applet, expressions typed by the user are commands: the values (`UUnits`) they return are not of further interest. Often, however, one does need to map the values of users' object language expressions back down to the metalanguage. For example, in the case of a simple embedded query language one might process queries using a function

```
val query : string * (record list) -> record list
```

mapping a query string and a list of records to a list of records matching the query, with an implementation like

```
fun query (qs, records) =
  let val pred = project (record-->bool) (interpretclosed (read qs))
  in filter pred records
  end
```

Projection can also be used to provide a simple form of metaprogramming or run-time code generation. Since ML code which constructs object-level expressions by directly manipulating either strings or elements of the `Exp` datatype is rather ugly, it is convenient to use the quote/antiquote mechanism provided by SML/NJ and Moscow ML. This allows one to write a parser (which we call `%`) for the object language into which ML values may be spliced (using the `^` operator).

The standard example of metaprogramming is a version of the power function `pow x y` which computes y^x by first building a specialised function `pow x` in which all the recursion has been symbolically unrolled, so `pow x` is essentially `fn y=>y*y*···*y`. In the unlikely event that one ever wished to raise many different numbers to the same exponent, reusing the specialised function can be more efficient than calling the general version many times. Using quote/antiquote, we can write a staged power function in a style which looks superficially like MetaML (Taha & Sheard, 2000):

```
- local fun mult x 0 = %`1`
          | mult x n = %`^x * ^(mult x (n-1))`
  in fun pow n = project (int-->int)
                (interpretclosed (%`fn y => ^(mult (%`y`) n)`))
  end;
val pow = fn : int -> int -> int
- val p5 = pow 5;
val p5 = fn : int -> int
- p5 2;
val it = 32 : int
- p5 3;
val it = 243 : int
```

Unsurprisingly, although this is significantly faster than calling an unspecialised interpreted function of two arguments, the overheads of our interpreter are non-trivial and it is still much faster to call a directly compiled ML version of the unspecialized function.

The function `mult` above has type `Exp->int->Exp`, so the ML values which we are splicing into the parse are bits of abstract syntax. We can obtain something even more useful by fusing the parser and interpreter to produce a new parser `%%` which constructs semantic objects of type

```
  staticenv -> dynamicenv -> U
```

without constructing any intermediate abstract syntax trees at all. The main advantage of this (apart from eliminating a datatype and some calls to `interpret`) is that we can now use antiquotation to splice ML values of type `U`, in particular ones obtained by `embed`, directly into object-language expressions (rather than having to give them names and add them to the environment):

```
- fun embedcl ty v = let val ev = embed ty v
                     in fn static => fn dynamic => ev
                     end;
```

```
- fun twice f n = f (f n);
- val h = %%`fn x => ^(embedcl ((int-->int)-->int-->int)
                              twice) (fn n=>n+1) x`;
val h = fn : staticenv -> dynamicenv -> U
- val hp = project (int-->int) (h [] []);
val hp = fn : int->int
- hp 2;
val it = 4 : int
```

Readers with an interest in metaprogramming may wish to consider extending the interpreter to allow more sophisticated versions of the following cunning trick:

```
- fun run s = interpret (read s,["run"]) [embed (string-->any) run];
val run = fn : string -> U
- run "let val x= run \"3+4\" in x+2";
val it = UI 9 : U
```

By embedding the interpreter itself, one can run object programs which manipulate second, and higher, level object programs.

4.2 Polymorphism

It is straightforward to embed and use polymorphic ML functions in the interpreted language. One only needs a single instantiation – the one where all type variables are mapped to `U` itself:

```
fun I x = x
fun K x y = x
fun S x y z = x z (y z)

val any : (U EP) = (I,I)

val combinators =
 [("I", embed (any-->any) I),
  ("K", embed (any-->any-->any) K),
  ("S", embed ((any-->any-->any)-->(any-->any)-->any-->any) S)]
```

And then if `combinators` is appended to the top-level environment `builtins`, evaluating, say

```
interpret (read "(S K K 2, S K K \"two\")")
```

yields `UP (UI 2, US "two") : U` just as one would hope.

Values of type `U` which represent polymorphic functions cannot simply be projected down to ML values with polymorphic (generalisable) ML types, in the sense that there isn't a way of extending the definitions so that

```
  project something (embed (any-->any) I)
```

has polymorphic type `'a->'a`. However we *can* project the same `U` value at multiple monomorphic ML types:

```
let val eI = embed (any-->any) I
in (project (int-->int) eI 3,
    project (string-->string) eI "three")
end
```

and then we can explicitly simulate type abstraction and application with ML's value abstraction and application:

```
let val eK = embed (any-->any-->any) K
    val pK = fn a => fn b => project (a-->b-->a) eK
in  (pK int string 3 "three", pK string unit "four" ())
end
```

Furthermore, we are quite free to project *untypeable* expressions in our object language down to typed ML values. For example, here's a much more amusing way of calculating factorials:

```
- let val embY = interpretclosed (read
         "fn f=>(fn g=> f (fn a=> (g g) a))
                (fn g=> f (fn a=> (g g) a))")
      val polyY = fn a => fn b=> project
                (((a-->b)-->a-->b)-->a-->b) embY
      val sillyfact = polyY int int
                (fn f=>fn n=>if n=0 then 1 else n*(f (n-1)))
  in (sillyfact 5) end;
val it = 120 : int
```

Here we have written an untyped CBV fixpoint combinator in the interpreted language and then projected it down to a polymorphic ML type, where it can be applied to values in the interpreting language.

4.3 Embedding Datatypes

There is a fairly natural way to embed arbitrary metalanguage datatypes, such as lists and trees, into the object language. If we ignore the straightforward but messy plumbing involved in trying to add pattern-matching syntax too, we can just extend the type `U` with one more constructor for tagging sum types

```
datatype U = ... | UT of int*U
```

and then define combinators for sums and recursive types

```
  val wrap : ('a -> 'b) * ('b -> 'a) -> 'b EP -> 'a EP
  val sum  : 'a EP list -> 'a EP
  val mu   : ('a EP -> 'a EP) -> 'a EP
```

whose definitions are

```
fun wrap (decon,con) ep = ((embed ep) o decon, con o (project ep))
fun sum ss = let fun cases brs n x =
                 UT(n, embed (hd brs) x)
                 handle Match => cases (tl brs) (n+1) x
             in (fn x=> cases ss 0 x,
                  fn (UT(n,u)) => project (List.nth(ss,n)) u)
             end
fun mu f = (fn x => embed (f (mu f)) x, fn u => project (f (mu f)) u)
```

The idea is that `sum` is given a list of partial embedding/projection pairs, each of which is specific to one constructor of the datatype. When embedding a datatype value, each of the embeddings is tried in turn until the appropriate one (i.e. the one which does not raise `Match`) is found. This yields both the appropriate embedding function and an integer tag. When projecting, the integer tag is used to select the appropriate projection function from the list. The `wrap` combinator is used to strip off and replace the actual datatype constructors, whilst the `mu` combinator is used to construct fixpoints of embedding/projection pairs for recursive types. Thus given a datatype definition of the form

$$\texttt{datatype d = } C_1 \texttt{ of } \tau_1 \texttt{ | ... | } C_n \texttt{ of } \tau_n$$

the associated embedding/projection pair will be given by

$$\begin{array}{l} \texttt{val d = mu (fn z => sum [wrap (fn (}C_1\texttt{ x) => x, }C_1\texttt{) }\overline{\tau_1}\texttt{,} \\ \qquad\qquad\qquad\qquad\qquad\quad \ldots \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{wrap (fn (}C_n\texttt{ x) => x, }C_n\texttt{) }\overline{\tau_n}\texttt{])} \end{array}$$

where $\overline{\tau_i}$ is the embedding/projection pair associated with the type τ_i, with `z` used as the embedding/projection pair for recursive occurences of the datatype `d`. Nullary constructors are treated as if they had type `unit`. For example, we can use these combinators to embed ML's (polymorphic) lists:

```
- fun list elem = mu ( fn l => (sum
   [wrap (fn []=>(),fn()=>[]) unit,
    wrap (fn (x::xs)=>(x,xs), fn (x,xs)=>(x::xs)) (elem ** l)]));
val list : 'a EP -> 'a list EP
```

If appropriate bindings are added to `builtins`:

```
 [... ("cons", embed (any**(list any)-->(list any)) (op ::)),
      ("nil", embed (list any) []),
      ("null", embed ((list any)-->bool) null), ... ]
```

then we can embed and project lists and list manipulating functions:

```
- interpretclosed (read "let fun map f l = if null l then nil
                   else cons(f (hd l),map f (tl l)) in map");
val it = UF fn : U
- project ((int-->int)-->(list int)-->(list int)) it;
val it = fn : (int -> int) -> int list -> int list
```

```
- it (fn x=>x*x) [1,2,3];
val it = [1,4,9] : int list
```

Whilst this kind of embedding is semantically elegant, it is extremely inefficient. The problem is that there are multiple representations for datatype values, and each time they cross the boundary between the two languages, they are converted in their entirety from one representation to another. Each use of an embedded primitive operation involves at least two of these representation changes which, for example, makes the above version of `map` have quadractic, rather than linear, time complexity.[5]

An alternative approach is to keep values of recursive types in their metalanguage representation. This can be done by adding extra constructors to the universal datatype `U` (just as we did with `tactic` in the Hal interpreter), but in ML it is also possible to use a well-known trick to extend `U` with new types dynamically. Briefly, by using the extensibility of ML exceptions, one can implement a structure `Dynamic` which exports an abstract type `dyn` and a function `newdyn` of type `unit -> ('a->dyn)*(dyn->'a)` which generates a new pair of functions for embedding and projecting values of any type into the type `dyn`. If one then extends the type `U` with a constructor `UD of Dynamic.dyn` then one can write, for example

```
fun newtype () = let val (tod,fromd) = Dynamic.newdyn()
                 in (UD o tod, fromd o PD)
                 end
val intlist = newtype () : (int list) EP
```

and then embed and project values whose types involve `int list`. This gains efficiency at the considerable cost of losing the ability to deal with datatypes polymorphically. One also has to be careful not to apply `newtype` twice to the same ML type, since the resulting embeddings will be type incompatible.

5 Monads

At first sight, it seems unlikely that we could extend our embeddings and projections to the situation where the interpreter is parameterised by an arbitrary monad, unless the values which we embed all have very restricted ML types. The problem is that we seem to need an 'extensional' version of a monadic translation which is normally defined *intensionally*, by induction on types and terms.

5.1 Monadic Translations, Extensionally

Recall that the CBV monadic translation $(\cdot)^*$ on types (Moggi, 1991) for a monad `'a t` is defined as follows:

$$\texttt{int}^* \quad = \quad \texttt{int} \quad \text{(similarly for other base types)}$$

[5] And with a pretty shocking constant factor too. It's been suggested to me that lazy evaluation might help here, but it doesn't really: although an embedded version of `hd`, for example, could avoid converting the entire tail of the list too, the back-and-forth coercions still build up in such a way that the `map` function remains quadratic.

$$(A \to B)^* = A^* \to (B^* \ \mathtt{t})$$
$$(A \times B)^* = A^* \times B^*$$

and there is an associated translation on terms such that if

$$\Gamma \vdash M : A$$

then

$$\Gamma^* \vdash M^* : A^* \, \mathtt{t}$$

If we do not have access to the source code of the values which we wish to translate (because they are already compiled code in the metalanguage) then we might wonder if there is a family of functions $\mathtt{tran}_A$ of type $A \to A^*$ for each ML type A such that for all $M : A$, $\mathtt{tran}_A \ M = M^* : A^*$. But a moment's thought should convince the reader that such functions cannot be defined in Standard ML or Haskell.[6]

Amazingly, however, provided that we restrict ourselves to CBV and to a metalanguage which already supports first-class continuations, it *is* possible to define an extensional monadic translation and, moreover, to express its type-dependency using ML values. The key is to use Filinski's deeply ingenious *monadic reflection* operations (Filinski, 1996; Filinski, 1999). We will only sketch Filinski's technique here for reasons of space, but a full account, containing both proofs and all the SML/NJ code we reference here, may be found in his thesis (Filinski, 1996).

Semantically, an ML function of type $A \to B$ may be understood as a mathematical function of type $A \to T_{ML}(B)$ where T_{ML} is a monad capturing the implicit effects of ML computations. These effects include non-termination, exceptions, IO and the use of mutable storage and, in the case of SML/NJ and MLton, first-class control. Filinski showed firstly that control is a universal effect, which can simulate any other monadic effect.[7] Secondly, he showed how one may define reification and reflection operations allowing one to move between opaque (implicit) and transparent (explicit) notion of control. Thirdly, he showed that this could all actually be implemented within a typed CBV language with control effects and global store, such as SML/NJ. Putting all the pieces together, one can define a functor which takes an arbitrary monad structure `M` as input and returns a reflected monad structure `R`, which adds two new operations to those of the original monad:

```
val reflect : 'a M.t -> 'a
val reify : (unit -> 'a) -> 'a M.t
```

These allow one to move between opaque and transparent representations of arbitrary ML-definable monads.

We will define our extensional CBV monadic translation by combining Filinski's reflection and reification functions with our earlier technique of defining type-indexed families of functions by representing each type as a pair of functions. Previously, we interpreted each type as a pair of an embedding into, and a projection

[6] Consider a type such as $(\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$, the translation of which involves `t` in a negative position.

[7] Formally, there is a monad retraction from any definable monad into the continuations monad.

from, the universal type `U`. To define our translation, we will represent each type A by a pair of a translation function $t : A \to A^*$ and an untranslation function $n : A^* \to A$. The type of this pair is not parametric in A so we cannot express it as an abstract type with a single parameter as we did with `'a EP`. However, we can parameterize over two type variables as shown in the result signature for our translation functor:

```
signature TRANSLATION =
sig
  structure R : RMONAD (* Filinski's reflected monad sig *)

  type ('a,'astar) TR = ('a->'astar)*('astar->'a)

  type 'a BASE = ('a,'a) TR

  val int : int BASE
  val string : string BASE
  val unit : unit BASE
  val bool : bool BASE

  val ** : ('a,'astar) TR * ('b,'bstar) TR ->
                          ('a*'b, 'astar*'bstar) TR
  val --> : ('a,'astar) TR * ('b,'bstar) TR ->
                          ('a->'b, 'astar -> 'bstar R.M.t) TR

  val translate : ('a,'astar) TR -> 'a -> 'astar
  val untranslate : ('a,'astar) TR -> 'astar -> 'a
end
```

The matching functor looks like this (where we have elided some declarations which are unchanged from code appearing earlier):

```
functor Translation (R : RMONAD) : TRANSLATION =
struct
  structure R = R

  val base = (I,I)
  val int = base
  val string = base
  val unit = base
  val bool = base

  fun (t,n)**(t',n') = (cross(t,t'), cross(n,n'))

  fun (t,n)-->(t',n') =
    (fn f=> fn x=> R.reify (fn ()=> t' (f (n x))),
```

```
    fn g=> fn x=> n'( R.reflect (g (t x))))

  fun translate (t,n) = t
  fun untranslate (t,n) = n
end
```

If, for example, we apply the reflection and translation functors to a structure defining a monad for integer-valued state

```
  type 'a t = int -> int * 'a
```

with the obvious monad structure and extra operations including

```
  fun accum m n = (m+n,())  (* : int -> unit t *)
```

then we can see the extensional translation at work in the following transcript:

```
- fun apptwice f = (f 1; f 2; "done");
val apptwice : (int->unit)->string
- val tapptwice = translate ((int-->unit)-->string) apptwice;
val tapptwice : (int->unit t)->string t
- tapptwice accum 0;
val it = (3, "done") : int*string
```

`tapptwice` is indeed what one would get by applying the intensional CBV monadic translation to the syntax of `apptwice`, but has been obtained extensionally from the compiled code.

5.2 Embedding Monadic Interpreters

We can now combine embedding/projection pairs with the extensional translation to write interpreters which are parameterised by an arbitrary monad *and* support embedding and projection of ML values.

In fact, there are two approaches we can take. Firstly, we might explicitly parameterize the universal datatype and the interpreter code by the monad. ML values would then be lifted to the object language by first translating them and then embedding them. A neater alternative, however, is to keep the monad *implicit* in the code for the interpreter. This way, we can leave the signatures and structures for the interpreter and embedding/projection pairs completely unchanged. We then write a new functor `MEmbeddings` which pairs each embedding/projection pair with its corresponding monadic translation/untranslation pair, yielding type-indexed *quads*. The signature is

```
signature MEMBEDDINGS =
sig
  structure R : RMONAD
  datatype U = datatype Embeddings.U

  type ('a,'b) TR = ('a->'b)*('b->'a)
```

```
  type 'a EP = ('a,U) TR
  type ('a,'astar) QUAD = ('a EP) * ('a,'astar) TR

  type 'a BASE = ('a,'a) QUAD

  val int : int BASE
  val string : string BASE
  val unit : unit BASE
  val bool : bool BASE
  val any : U BASE

  val ** : ('a,'astar) QUAD * ('b,'bstar) QUAD ->
                ('a*'b, 'astar*'bstar) QUAD
  val --> : ('a,'astar) QUAD * ('b,'bstar) QUAD ->
                ('a->'b, 'astar -> 'bstar R.M.t) QUAD


  val embed : ('a,'astar) QUAD -> 'a -> U
  val project : ('a,'astar) QUAD -> ('b -> U) -> 'b -> 'astar R.M.t
  val membed : ('a,'astar) QUAD -> 'astar -> U
end
```

whilst the matching functor looks like this:

```
functor MEmbeddings (R : RMONAD) : MEMBEDDINGS   =
struct
  ... bits elided...

  structure Tr = Translation(R)

  val int = (Embeddings.int, Tr.int)
  val string  = (Embeddings.string, Tr.string)
  val unit = (Embeddings.unit, Tr.unit)
  val bool =(Embeddings.bool, Tr.bool)
  val any = (Embeddings.any,(I,I))

  fun (ep,tn) ** (ep',tn') = (Embeddings.**(ep,ep'), Tr.**(tn,tn'))
  fun (ep,tn) --> (ep',tn') = (Embeddings.-->(ep,ep'), Tr.-->(tn,tn'))

  fun embed ((e,p),(t,n)) = e       (* A -> U  ordinary embedding *)
  fun membed ((e,p),(t,n)) = e o n  (* A* -> U monadic embedding  *)

  fun project ((e,p),(t,n)) f x = R.reify (fn ()=> t (p (f x)))
end
```

To embed an ordinary ML value, we still use the first component of the quad, so `embed ((e,p),(t,n))` is just `e`. The interesting thing is the way in which we embed the monad-specific extra operations associated with whatever notion of computation we have added. These extra operations will be implemented by ML values which have types which are *already* in the image of the CBV translation: we embed them using a new embedding function `membed` which first untranslates its argument and then embeds the result.

When we project back to ML, it is most useful to get a transparent representation of a computation of a translated type. Thus our `project` function combines reification with translation and projection. We also have to delay the evaluation of the computation we are projecting to make it happen 'under' the reification operation, which is why `project` takes a function `f` and an argument `x` rather than just a value of type `U`.

To see how all this works in practice, we consider the case of the list monad, which gives a kind of finite non-determinism to our interpreter. The two special operators we add are one which makes a choice between two values and a failing computation (which returns no result). The ML definitions of these functions are

```
fun choose (x,y) = [x,y] (* choose : 'a*'a->'a M.t *)
fun fail () = []         (* fail : unit->'a M.t *)
```

Note that the types of these functions are the CBV translations of `'a*'a->'a` and `unit->'a` respectively, and it is those underlying types which they will appear to have from the point of view of the object language when we add them to `builtins`:

```
val builtins = [("choose", membed (any**any-->any) choose),
                ("fail", membed (unit-->any) fail),
                ("+", embed (int**int-->int) Int.+), ... ]
```

We can then interpret non-deterministic programs and project their results back down to ML:

```
- project int (interpret (read
   "let val n = (choose(3,4))+(choose(7,9))
    in if n>12 then fail() else 2*n",[])) [];
val it = [20,24,22] : int ListMonad.t
```

Such a monadic embedded interpreter could, for example, be used to embed or implement a query language into an application (for example, the XML query language due to Fernandez, Siméon and Wadler (2001)).

6 Processes

So far, we have only considered embedding interpreters for object languages which are essentially CBV λ-calculi. There is another tractable foundational language which is quite different from, yet admits an interpretation of, the λ-calculus: the π-calculus(Milner *et al.*, 1992; Milner, 1999; Sangiorgi & Walker, 2001). In this section we will show how our techniques allow one to connect ML and an interpreter for the π-calculus such that the embedding of an ML value is the process which would be obtained by applying a well-known CBV translation of the λ-calculus, and suitably well-behaved processes may be projected back down as ordinary ML values.

We will actually use the choice-free, asynchronous, polyadic π-calculus, since this is simple, yet expressive enough to interpret functional computation. The same fragment was used as the core of the Pict language (Pierce & Turner, 2000). Note that this is a first-order calculus (the only values which may be transmitted are names or of primitive types) and that this makes the translation of functions more interesting than would be the case for a higher-order calculus in which processes may be transmitted along channels.

Like the monadic translation, the interpretation of functions as processes is usually presented by induction on terms, and target contexts can make distinctions between λ-terms which are extensionally equivalent in a more classical theory. The presence of first-class control in the metalanguage allows those distinctions between ML values to be made and is also the key to our interpretation of processes.

6.1 An Interpreter for the π-calculus

Writing a naïve interpreter for the π-calculus in ML is easy, though the degree to which processes, configurations and behaviours are treated as explicit, first-class "functional" values can vary widely. Here we choose a very imperative, implicit style because that leads to the most concise code and makes embedding ML values rather straighforward. It is also possible to use our earlier monadic translation techniques to embed and project ML values into and from an interpreter written in a more explicit, denotational style.

The signature for our π-calculus interpreter is as follows:

```
signature PROCESS =
sig
  type Name
  datatype BaseValue = VI of int | VS of string | VB of bool |
                       VU | VN of Name
  type Value = BaseValue list

  val new : unit -> Name
  val fork : (unit->unit)->unit
  val send : Name * Value -> unit
  val receive : Name -> Value

  type Process
  val Nil : Process
  val Nu : (BaseValue->Process)->Process
  val Par : Process*Process -> Process
  val Send : BaseValue*Value -> Process
  val Receive : BaseValue*(Value->Process)->Process
  val BangReceive : BaseValue*(Value->Process)->Process
```

```
  (* external interface to processes *)
  val schedule : Process -> unit
  val sync : unit -> unit
  val newname : unit -> BaseValue
end
```

Since we are implementing a polyadic calculus, a transmissible `Value` is a list of `BaseValue`s, which can be `Name`s, integers, strings or booleans. The combinators `Nil`, `Nu`, `Par`, `Receive`, `BangReceive` and `Send` correspond to the nil process **0**, restriction $\nu n.P$, parallel composition $P \mid Q$, input $x(y).P$, replicated input $!x(y).P$ and asynchronous output $\bar{x}\langle y \rangle$ respectively.

There is a single, global queue of runnable tasks, to which `Process`es are added using `schedule`. The `sync` command is used from the ML top-level loop and transfers control to the process scheduler, returning when/if no process is runnable. Finally, we have chosen to implement name generation using a global, sideeffecting name supply which is accessed through `newname`.

One possible implementation of this signature is shown in Figure 2. The first part of this structure is a fairly standard implementation of coroutines and asynchronous channels using `callcc` and `throw` (Wand, 1980; Reppy, 1999). A name (channel) is represented as a pair of mutable queues, one of which holds unconsumed values sent on that channel and the other of which holds continuations for processes blocked on reading that channel (at most one of the queues can be non-empty). New names are generated with `new`. Runnable threads are represented by `unit`-accepting continuations held in the mutable queue `readyQ`. Values are sent asynchronously using `send`; this either adds the value to the message queue for that channel or, if there is a thread blocked on reading, it enqueues its own continuation and transfers control to the blocked thread, passing the sent value. Calls to `receive` either remove and return a value from the appropriate queue or, if no value is yet available, add their own continuation to the blocked queue and call the scheduler. New threads are created and scheduled using `fork`, which enqueues its own continuation, runs the child and returns to the scheduler. Control is transferred to the scheduler with `sync`, which busy-waits for the `readyQ` to become empty before returning. Our slightly more π-calculus-style combinators are then thin wrappers over the coroutine implementation.

Now we can construct and run processes using the combinators directly, but it is more convenient to write a parser and interpreter for a more palatable syntax, such as that of Pict:

```
- val pp = %`new ping new pong (ping?*[] = echo!"ping" | pong![]) |
                               (pong?*[] = echo!"pong" | ping![]) |
                               ping![]`;
val pp = - : Exp
- schedule (interpret (pp, Builtins.static) Builtins.dynamic);
val it = () : unit
- sync ();
```

```
structure Process :> PROCESS =
struct
  open Queue
  open SMLofNJ.Cont

  type 'a chan = ('a queue) * ('a cont queue)

  datatype BaseValue = VI of int | VS of string | VB of bool | VU | VN of Name
  and Name = Name of (BaseValue list) chan

  type Value = BaseValue list

  val readyQ = mkQueue() : unit cont queue

  fun new() = Name (mkQueue(),mkQueue())

  fun scheduler () = throw (dequeue readyQ) ()

  fun send (Name (sent,blocked),value) =
     if isEmpty blocked then
       enqueue (sent,value)
     else callcc (fn k => (enqueue(readyQ,k); throw (dequeue blocked) value))

  fun receive (Name (sent,blocked)) =
     if isEmpty sent then
       callcc (fn k => (enqueue (blocked,k); scheduler ()))
     else dequeue sent

  fun fork p = callcc (fn k=> (enqueue (readyQ,k); p(); scheduler()))

  fun sync () = if length readyQ = 0 then ()
                else (callcc (fn k=> (enqueue(readyQ, k); scheduler())); sync())

  type Process = unit -> unit

  fun newname () = VN (new())
  val schedule = fork

  fun Nil () = ()
  fun Send(VN ch,v) () = (send(ch,v);())
  fun Receive(VN ch,f) () = (f (receive ch) ())
  fun Par(p1,p2) () = (fork p1; p2())
  fun BangReceive(VN ch,f)  = Receive(VN ch,fn v=>Par(f v, BangReceive(VN ch,f)))
  fun Nu f () = f (newname()) ()
end
```

Fig. 2. Implementation of Continuation-Based Coroutines

```
pingpongpingpongpingpongpingpongpingpongpingpong...
```

The parser and interpreter themselves are unexciting, so we just present parts of their signatures below. Observe, however, that in order to be able to observe the process, we have already made use of some embedding: `pp` has a free identifier, `echo`, which is bound to a name by an environment in the structure `Builtins`. This also defines a process listening on that name and printing the strings it receives. We will give the definition of `Builtins` once we have explained the embedding of ML functions as processes.

```
signature INTERPRET =
sig
  type Exp
  val read : string -> Exp     (* simple parser *)
  val % : Exp frag list -> Exp (* quoted parser *)

  type staticenv = string list
  type dynamicenv = Process.BaseValue list
  val interpret : Exp*staticenv -> dynamicenv -> Process.Process
end

structure Interpret :> INTERPRET = ... elided ...

signature BUILTINS =
sig
  val static : Interpret.staticenv
  val dynamic : Interpret.dynamicenv
end

structure Builtins :> BUILTINS = ... see later ...
```

6.2 Functions as Processes and Back Again

Translations of λ-calculi into π-calculi have been extensively studied since the start of the 1990s (Milner, 1992; Sangiorgi, 1992). The CBV translation is typically presented like this:

$$\begin{array}{rcl} [\![\lambda x.M]\!]p & = & \bar{p}\langle c\rangle.!c(x,r).[\![M]\!]r \\ [\![x]\!]p & = & \bar{p}\langle x\rangle \\ [\![M\,N]\!]p & = & \nu q.([\![M]\!]q \mid q(v).\nu r.([\![N]\!]r \mid r(w).\bar{v}\langle w,p\rangle)) \end{array}$$

The basic idea is that the translation of a computation is a process, parameterized by a name p along which the the location of a process encoding a value should be sent. A λ-abstraction is translated as a process located at a particular name c. The process repeatedly receives pairs of values on c, the first of which is the location of an argument and the second of which is a name along which the location of the result should be sent. Application is translated as the parallel composition of three processes: the first evaluates the function and sends its location v along q, the second receives that location and evaluates the argument, sending its value along r. The third process wires the function and argument together by receiving the argument value location w along r and sending it, together with the name p where the final result should be sent, to the function at v.

As stressed by, for example, Boudol (1997) and Sangiorgi & Walker (2001), this encoding is essentially a CPS translation. Since our interpretation of processes is already based on continuations, it is perhaps not too surprising that we can implement the encoding using this interpretation.

```
signature EMBEDDINGS =
sig
  type 'a EP
  val int : int EP
  val string : string EP
  val bool : bool EP
  val unit : unit EP

  val **  : ('a EP)*('b EP) -> ('a*'b) EP
  val --> : ('a EP)*('b EP) -> ('a->'b) EP

  val embed : ('a EP) -> 'a -> Process.BaseValue
  val project : ('a EP) -> Process.BaseValue -> 'a
end
```

This signature is apparently simple – it is essentially the same as that we presented back in Section 3, with `BaseValue` instead of `U` – but some complexity is hidden in the fact that embeddings are now side-effecting. Embedding an ML function will create and schedule an appropriate process, returning a channel by which one may interact with it. Similarly, projecting a function only takes a name as argument, but implicitly refers to the global process pool. The implementation of the encoding is shown in Figure 3. The embeddings for base values are straightforward: they just return the the corresponding transmissible `BaseValue`. The embeddings and projections at function and product types are defined in terms of the coroutine primitives.

The embedding at a function type takes a value `f`, generates a new name `c` and then forks a thread which repeatedly receives an argument and a result channel along `c`. The argument is projected, `f` is applied and the result is embedded, yielding a `BaseValue`, `resc`, and possibly spawning a new thread. Then (the location of) the result is sent along the result channel `rc`. The projection at function types takes a channel `fc` for interacting with a functional process and yields a function which embeds its argument (spawning a new thread in the case that the argument is a function), generates a new name `rc` for the result of the application and sends that, along with the location of the argument to `fc`. It then blocks until it receives a reply along `rc` and returns the projection of that reply.

To embed a pair, we embed the two components and then return the address of

```
structure Embeddings :> EMBEDDINGS =
struct
  open Process

  type 'a EP = ('a->BaseValue)*(BaseValue->'a)

  val int = (VI, fn (VI n)=>n)
  val string = (VS, fn (VS s)=>s)
  val bool = (VB, fn (VB b)=>b)
  val unit = (fn ()=> VU, fn VU=>())

  infix **
  infixr -->

  fun (ea,pa)-->(eb,pb) =
    (fn f => let val c = new()
                 fun action () = let val [ac,VN rc] = receive c
                                     val _ = fork action
                                     val resc = eb (f (pa ac))
                                 in send(rc,[resc])
                                 end
             in (fork action; VN c)
             end,
     fn (VN fc) => fn arg => let val ac = ea arg
                                 val rc = new ()
                                 val _ = send(fc,[ac,VN rc])
                                 val [resloc] = receive(rc)
                             in pb resloc
                             end
    )

  fun (ea,pa)**(eb,pb) =
   ( fn (x,y) => let val pc = new()
                     val c1 = ea x
                     val c2 = eb y
                     fun action () = let val [VN r1,VN r2] = receive pc
                                     in (fork action;send(r1,[c1]);send(r2,[c2]))
                                     end
                 in (fork action; VN pc)
                 end,
     fn (VN pc) => let val r1 = new()
                       val r2 = new()
                       val _ = send(pc,[VN r1, VN r2])
                       val [c1] = receive r1
                       val [c2] = receive r2
                   in (pa c1, pb c2)
                   end
   )

  fun embed (ea,pa) v = ea v
  fun project (ea,pa) c = pa c
end
```

Fig. 3. Extensional π-calculus Translation

a new located process which repeatedly receives two channel names along which it sends the addresses of the embedded values. Projection creates two new names, sends them to the pair process, waits for the two replies and returns a pair of their projections.

Now the body of the `Builtins` structure can be defined something like this:

```
fun e name typ value = let val bv = embed typ value
                       in (name,bv)
                       end
val embs =
  [e "inc" (int-->int) (fn x=>x+1),
   e "add" (int-->int-->int) (fn x=>fn y=>x+y),
   e "print" (string-->unit) print,
   e "itos" (int-->string) Int.toString,
   e "twice" ((int-->int)-->(int-->int)) (fn f => fn x => f (f x)),
   ...]

val specials = [("devnull", "devnull?*[x]=nil"),
                ("echo", "echo?*[s] = print![s devnull]")]

val static = (map (#1) embs) @ (map (#1) specials)
val dynamic = (map (#2) embs) @ (map (fn _ => newname()) specials)
val _ = map (fn (_,s) =>
            schedule (interpret (read s,static) dynamic)) specials
```

Note that the definition of the `echo` process, which we used earlier, is not entirely trivial: it is an asynchronous (one-way) wrapper which sends a message to a process embedding the ML `print` function, but directs responses to be sent to a process which simply discards them.

Here is a simple example of a process using embedded functions:

```
- fun test s = let val p = Interpreter.interpret (Exp.read s,
                                  Builtins.static) Builtins.dynamic
               in (schedule p; sync())
               end;
val test = fn : string -> unit
- test "new r1 new r2 twice![inc r1] | r1?f = f![3 r2]
                                     | r2?n = itos![n echo]";
5
```

The process is essentially the translation of

```
print (Int.toString (twice inc 3))
```

and does indeed compute the right answer. The reader will notice that we have applied a tail-call optimization: `itos` is asked to send its result directly to `echo`. More interesting examples involve processes which interact with embedded ML functions in non-sequential, non-functional ways. For example, if we embed the function

```
fun appupto f n = if n < 0 then ()
                  else (appupto f (n-1); f n)
```

of type `(int->unit)->int->unit`, we can do

```
- test "new r1 new r2 new c appupto![println r1] |
        (r1?f = c?*[n r] = r![] | f![n devnull]) |
        appupto![c r2] | r2?g = g![10 devnull]";
00100110220130120123102342013430124541235523466345745656767878889910
```

We have applied `appupto` to `println` yielding a process located at `f` which will sequentially print the integers from 0 to n. We then define a functional process at `c` which accepts an integer and indicates completion immediately but spawns a call to the functional process located at `f`. Finally, we call `appupto` again with the process located at `c` and `10` as arguments. The outcome is that for each n from 0 to 10, we print all the integers from 0 to n, with the 11 streams being produced in parallel.

Here is an example of projection:

```
- fun ltest name s =  let val n = newname()
                          val p = Interpreter.interpret (Exp.read s,
                      name :: Builtins.static) (n :: Builtins.dynamic)
                      in (schedule p; n)
                      end;
val ltest = fn : string -> string -> BaseValue
- val ctr = project (unit-->int) (ltest
               "c" "new v v!0 | v?*n = c?[r]=r!n | inc![n v]");
val ctr = fn : unit -> int
- ctr();
val it = 0 : int
- ctr();
val it = 1 : int
- ctr();
val it = 2 : int
```

We have defined a π-calculus process implementing a counter and projected as a stateful ML function of type `unit->int`. An interesting variant is the following, in which two counter processes, both listening on the same channel, are composed in parallel:

```
- val dctr = project (unit --> int) (ltest "c"
              "(new v v!0 | v?*n = c?[x r]=r!n | inc![n v]) |
               (new v v!0 | v?*n = c?[x r]=r!n | inc![n v])");
val dctr = fn : unit -> int
- dctr();
val it = 0 : int
```

```
- dctr();
val it = 0 : int
- dctr();
val it = 1 : int
- dctr();
val it = 1 : int
```

Each call to dctr nondeterministically picks one of the counters to advance. And, of course, we can calculate factorials by projecting a π-calculus version of a fixpoint combinator to ML:

```
- val y = project (((int-->int)-->int-->int)-->int-->int)
          (procandchan "y" "y?*[f r] = new c new l r!c | f![c l] |
                                l?h = c?*[x r2]= h![x r2]");
val y = fn : ((int -> int) -> int -> int) -> int -> int
- y (fn f=>fn n=>if n=0 then 1 else n*(f (n-1))) 5;
val it = 120 : int
```

7 Discussion and Related Work

Type-directed constructions like those we use for embedding/projection pairs have arisen in several contexts recently, including normalisation by evaluation (NBE) and type-directed partial evaluation (TDPE) (Danvy, 1996; Hatcliff *et al.*, 1998; Danvy, 1998b; Yang, 1998), printf-like string formatting (Danvy, 1998a), pickler combinators (Kennedy, 2003) and generic programming.

The similarities with NBE are most striking. Indeed, were it not for the fact that NBE does not apply to all the types in which we are interested, one could imagine solving our original problem in a dual manner: using NBE to produce object-level expressions corresponding to already-compiled metalanguage values and then simply interpreting them along with the user program. Filinski's monadic reflection techniques are also used when extending TDPE and NBE to languages with sums and side-effects (Danvy, 1996). Our observation that monadic reflection can be used to define an extensional variant of Moggi's CBV translation is original, though Filinski (2001) has recently, and independently, published a paper on using the same technique to define an extensional CPS translation. Our treatment of recursive types does not seem to have any analogue in the NBE literature, though Kennedy's (2003) pickler combinators use essentially the same technique.

If we were working in Haskell, rather then SML, we could make good use of the type class mechanism, which is powerful enough to deal with the generic requirements of our interpreter.[8] One could simply define a type class EP specifying the types of the embedding and projection functions and then declare firstly, each of the base types to be instances of that class (giving the specific embeddings and projections), and secondly, that if 'a and 'b are in the class EP, so are 'a->'b and

[8] Of course, the monadic and π-calculus embeddings cannot be implemented in Haskell, since they rely on first-class control.

`'a*'b` (giving the appropriate constructions on embeddings and projections). Then one could just write `embed v` instead of `embed` τ `v`. Of course, polymorphic values would still need to be explicitly constrained to resolve the ambiguity. Rose (1998) has applied just this technique to an implementation of TDPE in Haskell.

Staged computation is also an active research area (Taha & Sheard, 2000; Leone & Lee, 1996). The metaprogramming facilities of our interpreter are not really comparable to those of compilers for proper staged languages, since we only have untyped object programs and no real compilation of object programs. Nevertheless, it is certainly interesting to see how far one can get with such a simple-minded approach, and there are opportunities to make it a little less simple-minded, for example by combining our techniques with phantom types (Rhiger, 2002) and exploiting the fact that the semantic (`%%`...``) version of quotation allows embedded metalanguage expressions access to the object-level environment.

The π-calculus embedding is novel and certainly has didactic value, but I believe it may also prove to be useful in practice. The translation of functions into processes is just what one would need to expose higher-order functions as web services (World Wide Web Consortium, 2002), for example. It should also be pointed out that although the implementation given here is cooperative, in SML/NJ it is not hard to use an asynchronous signal from an operating system timer to implement genuinely preemptive scheduling (Reppy, 1999).

Finally, of course, there is ample scope for formulating, and attempting to prove, some correctness results for the constructions presented here.

Thanks to Andrew Kennedy, Simon Peyton Jones and an ICFP referee for helpful comments on earlier drafts of this paper.

References

Benton, N., Kennedy, A., & Russell, G. 1998 (Sept.). Compiling Standard ML to Java bytecodes. *Proceedings of the 3rd ACM SIGPLAN conference on functional programming.*

Boudol, G. (1997). The pi-calculus in direct style. *Pages 228–241 of: ACM symposium on principles of programming languages.*

Danvy, O. (1996). Type-directed partial evaluation. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages.* ACM.

Danvy, O. (1998a). Functional unparsing. *Journal of functional programming*, **8**(6).

Danvy, O. (1998b). A simple solution to type specialization (extended abstract). *Pages 908–917 of:* Larsen, Skyum, & Winskel (eds), *Proceedings of the 25th international colloquium on automata, languages, and programming.* Lecture Notes in Computer Science, vol. 1443. Springer-Verlag.

Fernandez, M., Siméon, J., & Wadler, P. (2001). A semistructured monad for semistructured data. Van den Bussche, J., & Vianu, V. (eds), *Proceedings of the 8th international conference on database theory.* Lecture Notes in Computer Science, vol. 1973. Springer-Verlag.

Filinski, A. (1996). *Controlling effects.* Tech. rept. CMU-CS-96-119. Carnegie-Mellon University.

Filinski, A. (1999). Representing layered monads. *Pages 175–188 of: Proceedings of the*

26th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM.

Filinski, A. 2001 (Jan.). An extensional CPS transform (preliminary report). Sabry, A. (ed), *Proceedings of the 3rd ACM SIGPLAN workshop on continuations*. Technical Report 545, Computer Science Department, Indiana University.

Hatcliff, J., Mogensen, T., & Thiemann, P. (eds). (1998). *Proceedings of the DIKU 1998 international summerschool on partial evaluation*. Lecture Notes in Computer Science, vol. 1706. Springer-Verlag.

Jones, N. D., Gomard, C. K., & Sestoft, P. (1993). *Partial evaluation and automatic program generation*. Prentice Hall International.

Kennedy, A. (2003). *Pickler combinators*. to appear in Journal of Functional Programming.

Leone, M., & Lee, P. (1996). Optimising ML with run-time code generation. *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*. ACM.

Milner, R. (1992). Functions as processes. *Mathematical structures in computer science*, **2**(2), 119–146.

Milner, R. (1999). *Communicating and mobile systems: The pi-calculus*. Cambridge University Press.

Milner, R., Parrow, J., & Walker, D. (1992). A calculus of mobile processes I/II. *Information and computation*, **100**(1), 1–77.

Moggi, E. (1991). Notions of computation and monads. *Information and computation*, **93**, 55–92.

Paulson, L. C. (1991). *ML for the working programmer*. Cambridge University Press.

Pierce, B. C., & Turner, D. N. (2000). Pict: A programming language based on the pi-calculus. Plotkin, G., Stirling, C., & Tofte, M. (eds), *Proof, language and interaction: Essays in honour of Robin Milner*. MIT Press.

Reppy, J. H. (1999). *Concurrent programming in ML*. Cambridge University Press.

Reynolds, J. C. (1998). Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, **4**(11), 363–397.

Rhiger, M. 2002 (Aug.). *A foundation for embedded languages*. Tech. rept. RS-02-34. BRICS.

Rose, K. (1998). Type-directed partial evaluation in Haskell. Danvy, O., & Dybjer, P. (eds), *Preliminary proceedings of the 1998 APPSEM workshop on normalization by evaluation*. BRICS Notes, nos. NS–98–1.

Sangiorgi, D. (1992). The lazy lambda calculus in a concurrency scenario. *IEEE symposium on logic in computer science*. IEEE.

Sangiorgi, D., & Walker, D. (2001). *The pi-calculus: A theory of mobile processes*. Cambridge University Press.

Taha, W., & Sheard, T. (2000). MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, **248**(1–2), 211–242.

Wand, M. (1980). Continuation-based multiprocessing. *Proceedings of the 1980 ACM conference on LISP and functional programming*. 1980.

World Wide Web Consortium. (2002). *Web services activity*. `http://www.w3.org/2002/ws/`.

Yang, Z. 1998 (Sept.). Encoding types in ML-like languages. *Proceedings of the 3rd ACM SIGPLAN conference on functional programming*.