Embedded Interpreters

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

Abstract

This is a tutorial on using type-indexed embedding/projection pairs when writing interpreters in statically-typed functional languages. The method allows (higher-order) values in the interpreting language to be embedded in the interpreted language and values from the interpreted language may be projected back into the interpreting one. This is particularly useful when adding command-line interfaces or scripting languages to applications written in functional languages.

We first describe the basic idea and show how it may be 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 π-calculus via an 'extensional' variant of a standard translation from λ into π.

1 Introduction

1.1 Embedding Little Languages

Many applications need to incorporate one or more 'little languages' (Bentley, 1986), typically to provide a top-level interactive loop or a scripting interface. These languages may be domain-specific (van Deursen *et al.*, 2000) or general-purpose.

If the application is written in C, a common approach is to use one of several popular embeddable interpreters for general-purpose scripting languages, such as Tcl (Ousterhout, 1990) or Python (van Rossum, 2003). The code for the interpreter is linked with that of the application and calls made at runtime to initialize the environment of instances of the interpreter with bindings for (wrapped versions of) functions defined by the application.

If the application is written in a language like Java or C$^\sharp$, there are also numerous embeddable interpreters available, such as Jython (Hugunin, 1997), Rhino (Mozilla Organisation, n.d.) or JScript.NET (Clinick, 2000). In these cases, the glue between the two languages is provided by the underlying execution engine's support for dynamic types and reflection.

In the functional programming community, many language implementations have an interactive read-eval-print loop; it is then common for applications to rely on this host programming environment to provide a user interface and 'scripting' language. An application simply comprises a collection of top-level definitions that are used, scripted and extended by writing more code in the same language. This approach has much to recommend it; higher-order functions, and powerful macro facilities like those of Scheme, allow one to blur the distinction between application, API, domain-specific language and combinator library and the application gets a powerful, efficient, general purpose command language at very low cost. The advantages of these 'shallow embeddings' of domain specific languages in general purpose higher-order languages are argued in more detail by, for example, Shivers (1996) and Hudak (1998).

Powerful though the shallow embedding technique is, especially when writing programs for use by computer scientists, it does have its drawbacks:

1. Applications have to be shipped with, or rely upon, a complete implementation of the programming language in which they are implemented, including standard libraries. This may be impractical for reasons of code size or licensing.
2. One can only get an interactive user interface this way if the language implementation has an interactive top-level loop (or supports `eval`).
3. The syntax of the application-specific language is, even with the use of macros, restricted to be essentially the same as that of the host language.
4. The same is true of semantics. Embedding a DSL with dynamic binding is hard in a host language that supports only static binding, for example.
5. It is very hard indeed to produce application-specific error messages.
6. It is very hard to *restrict* what users can do.
7. The mechanics and plumbing of top-level loops are usually primitive and/or inflexible. Integrating textual commands within an application-specific GUI (rather than a language-specific IDE), for example, is likely to be tricky.

Thus, even when an application is written in a higher-order functional language, it is frequently necessary to augment it with a separate parser and interpreter. This appears to present few problems: writing parsers and interpreters in functional languages is straightforward and well-studied. However, 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 values, possibly of higher-order function types, 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 *wrap* (or *lift*) these values from the metalanguage in which the application is written into the domain-specific object language. It is also often useful to map values in the

opposite direction, *unwrapping* the results of object language computations so they may be passed back into the application. How to perform this kind of wrapping and unwrapping is the problem we will address here.

We should remark that the lifting problem also arises when embedding interpreters for scripting languages in C programs. In that context, the application programmer typically writes (or uses a separate tool to generate) an explicit wrapper function for each application function that is to be be exposed in the interpreter. These wrappers comprise tedious boilerplate for converting between C and Python (or whatever) representations of values, and typically also have to deal with complexities of interlanguage memory management. The essence of our technique is an elegant, concise and uniform way of generating this kind of boilerplate in a functional language.

The basic issue can be seen as one of interlanguage interoperability: we wish to make values defined in one language available for use in another. In this paper, we are mainly concerned with the special, asymmetric situation in which the second language is implemented by an interpreter written entirely in the first language. Ramsey (2003) has applied similar techniques to embed an interpreter for the scripting language Lua into a compiler written in OCaml. We will further assume that our metalanguage is statically typed (we will use Standard ML, but OCaml or Haskell would do just as well, at least for the first half of the paper). This makes the problem harder than would be the case in a dynamically-typed language, since to make the embedding operations statically type-correct we will have to do something different for different metalanguage types, and each of those types is distinct from the static type(s) of values manipulated at runtime by the interpreter. In an untyped language such as Scheme, by contrast, (even without `eval` or the reflective operations used in script interpreters for the JVM or CLR) it is easier to arrange that values manipulated by the interpreter are compatible at runtime with values defined in the application. (This still requires care, as the replacement of Exercise 4.8 in (Abelson *et al.*, 1985) with Exercise 4.14 in (Abelson *et al.*, 1996) indicates.)

Although we work with a statically typed metalanguage, our object languages will be untyped. This should not be taken as an endorsement of the view that scripting languages should be untyped (Ousterhout, 1998), but simply a reflection of the fact that if object language programs are parsed from strings at runtime then any typing restrictions can only be enforced by running an explicit type checker, which is independent of the static type system of the metalanguage. Typechecking object programs can thus be regarded as an orthogonal problem, solved by entirely standard methods. (In cases where object programs are less dynamic and the embedding is shallower, techniques such as phantom types (Rhiger, 2003) can leverage the metalanguage type system to reject ill-typed object programs at compile-time.)

1.2 A Motivating Example: The Hal Applet

The basic ideas presented here were originally developed to solve a 'real' problem. As one of the examples for MLj (Benton *et al.*, 1998), we compiled Paulson's (1991) tactical theorem prover Hal as an applet, which runs in any Java-enabled web

browser. 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.

Like many ML programs, Hal is intended to be used from within the top-level read-eval-print loop of an interactive ML system, but we wanted to compile it as a compact, stand-alone applet with its own user-interface. 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 had to deal with processing user input. This meant writing an interpreter for a simple functional language and embedding within it as primitives the thirty or so SML values, like `repeat`, `conjR` and `||`, that make up the application. Doing this in an ad-hoc manner would have been rather tedious, especially as many of the primitives have higher-order types.

1.3 Outline

The rest of the paper is organised as follows:

- Section 2 explains the basic technique for type-directed embeddings into a lambda-calculus interpreter. This is all we required in the case of the Hal applet.
- Section 3 discusses mapping values back from the object language to the metalanguage and shows how this yields a simple form of metaprogramming.
- Section 4 discusses polymorphism, untypable object programs and type-based dispatch.
- Section 5 shows how to extend the basic technique to treat recursive datatypes.
- Section 6 describes a continuation-based version of the embedding technique, which allows the embedded interpreter to be parameterized by an arbitrary monad.
- Section 7 uses continuations to embed functions into an interpreter for a concurrent language based on the π-calculus.

The reader who is primarily interested in implementing a simple replacement for a top-level loop in standalone functional programs may wish to read only Sections 2 and 5. The material in Sections 3 and 4 indicates how the embedding/projection technique relates to metaprogramming and dynamic types; this is not necessary for understanding any of the rest of the paper. Sections 2 to 5 assume only basic knowledge of functional programming and are mostly applicable in any statically-typed higher-order language. Sections 6 and 7 are semantically more advanced, assuming some knowledge of monads and process calculi, respectively, and the programming techniques rely on first-class control operators and are therefore not so widely applicable.

Most of the examples are artificial, in that they involve embedding and projecting simple and familiar 'theoretician's' functions, such as factorials and fixpoint combinators. As this might otherwise obscure the distinction between techniques that are likely to be useful in practice and those that are of more academic interest, I have indicated into which category I believe each technique falls. Your mileage may, of course, vary.

To improve readability of code, we use an *italic* typewriter font for object language programs. We also take mild liberties with formatting, such as splitting string constants across lines without including continuation characters.

2 The Basic Idea

Here is an SML datatype representing abstract syntax trees for a minimal functional language:

```
datatype Exp = EId of string                    (* identifier    *)
             | EI of int                        (* integer const *)
             | ES of string                     (* string const  *)
             | EApp of Exp*Exp                  (* application   *)
             | EP of Exp*Exp                    (* pairing       *)
             | ELet of string*Exp*Exp           (* let binding   *)
             | EIf of Exp*Exp*Exp               (* conditional   *)
             | ELam of string*Exp               (* abstraction   *)
             | ELetfun of string*string*Exp*Exp (* recursive fn  *)
```

We will further assume that we have implemented a parser

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

for an ML-like concrete syntax.

There are several ways of structuring an interpreter for such a language. The 'syntactic' approach – actually performing substitutions on elements of `Exp` (or some extension thereof with a case for closures) – does not support the constructions we will be making, as there is no sufficiently general way to map compiled code for metalanguage values back into syntax trees (though we briefly discuss Normalization By Evaluation in Section 8). Instead we take the 'semantic' approach, interpreting

expressions of type `Exp` in an ML datatype giving a model of an untyped CBV lambda calculus with constants and pairing:

```
datatype U = UF of U->U | UP of U*U | UU | UI of int | US of string
```

Writing an interpreter in this style amounts to giving a denotational semantics. We show an extract in Figure 1; this is fairly standard, though note the binding-time separation: rather than a single environment of type `(string*U) list`, the interpreter takes a static environment of type `string list` and produces a function consuming a matching dynamic environment of type `U list`. The case for identifiers calls `Builtins.lookup`, which we will define shortly.

The novel observation is that such a meta-circular interpreter (Reynolds, 1972), which uses metalanguage functions to interpret object language 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 that *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 **     : ('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)  *)
  fun PI (UI(n))=n  (* : U -> int    *)
  fun PS (US(s))=s  (* : U -> string *)
  fun PU (UU)=()    (* : U -> unit   *)
```

```
  infixr --> infix **
  val unit   = (UU,PU)
  val int    = (UI,PI)
  val string = (US,PS)
  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 an 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`. Of course, these exceptions *should* be caught and gracefully handled, but we will omit all error handling for reasons of space and clarity.

For any ML type A that 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. If one replaced the exception-throwing with divergence, then `embed`$_A$ `o project`$_A$ would be less than or equal to the identity in the conventional domain-theoretic sense.

We can now use our embeddings to lift ML values into the object language. The structure defining the built-in values looks like

```
structure Builtins :>
sig
  val lookup : string -> Embeddings.U
end =
struct
  (* An arbitrary example of a higher-order function *)
  fun iter m f n = if n=0 then m else f n (iter m f (n-1))

  val builtins =
        [("*", embed (int**int-->int) Int.*),
         ("true", embed bool true),
         (">", embed (int**int-->bool) (op >)),
         ("print", embed (string-->unit) print),
```

```
          ("toString", embed (int-->string) Int.toString),
          ("iter", embed (int-->(int-->int-->int)-->int-->int) iter),
          ...
         ]

  fun find x ((y,v)::rest) = if x=y then v else find x rest
  fun lookup s = find s builtins
end
```

and we can indeed access embedded values from the object language:

```
- interpretclosed (read "let val f = iter 1 (fn x => fn y => x*y)
                         in print (toString (f 5))");
120
val it = UU : U
```

Observe in particular how the higher-order ML function `iter` is called, passing an object language function, which is itself defined in terms of ML's `*`.

The return value `UU:U` above is the object language interpretation of the `unit` value returned by `print`. The code for the Hal theorem prover similarly updates its state and prints its results imperatively, so this kind of simple unidirectional embedding of ML functions into the command language was all that was needed to solve our original problem.

3 Projection and Quoting

Embedding/projection pairs allow one to do considerably more than just run end-user commands that 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
```

More interesting are the cases in which the object language expression is either constructed or read in at run-time and the result of its evaluation is then used in a non-trivial metalanguage context. 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))
```

```
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 => (fn dynamic => (UI n))
 | EId s => (let val n = indexof (static,s)
             in fn dynamic => List.nth (dynamic,n)
             end handle Match => let val lib = Builtins.lookup s
                                 in fn dynamic => 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)
                    in fn dynamic => let fun g v = s1 (v::UF(g)::dynamic)
                                     in s2 (UF(g)::dynamic)
                                     end
                    end
 | ... other clauses elided ...

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

Fig. 1. An Interpreter (extract)

```
   in filter pred records
   end
```

Query strings are interpreted and then projected down as ML functions (predicates) that can be passed to `filter`.

Projection can also be used to provide a simple form of metaprogramming or run-time code generation. Since ML code that 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 (Slind, 1991) and Moscow ML. This allows one to write a parser (which we call `%`) for the object language syntax into which ML values of type `Exp` may be spliced using the *antiquote* operator, '`^`'.

The standard example of metaprogramming is a version of the power function `pow x y` that 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*`$\cdots$`*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 that looks just like MetaML (Taha & Sheard, 2000) though, of course, the most interesting aspect of MetaML is the multistage type system, which in our case we have not got:

```
- 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 calls to partial applications of `pow` are significantly faster than calls to 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 that 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 `%%` that 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 extend the following cunning trick with a more sophisticated treatment of free variables:

```
- 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 that manipulate second, and higher, level object programs.

Utility Projecting the interpretation of object level expressions that are read in dynamically is certainly useful in integrating script evaluation into a broader application.

It is not so clear that one would really want to write ML programs that dynamically *construct* object level syntax in complex ways (rather than reading it from an input stream) before projecting its interpretation. The usual reason for doing runtime code generation is performance, but in the absence of genuine runtime compilation, careful staging of a program written in entirely in the metalanguage (as demonstrated in our interpreter itself) is probably the best approach to achieving that.

In cases where the object level syntax or semantics differ significantly from, or extend, that of the metalanguage, projection of even statically constant object level expressions can be convenient, however. There are many popular systems that allow source code in multiple languages to be intermingled (for example, embedded SQL or various (server- and client-side) frameworks for mixing HTML and code), and our techniques allow similar things to be done in ML.

4 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

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

yields `UP (UI 2, US "two") : U` just as one would hope. Mapping all type variables to a universal type is like the use of the 'top' type `Object` when writing polymorphic code in monomorphic object-oriented languages.

Values of type `U` that represent polymorphic functions cannot simply be projected down to ML values with polymorphic (generalisable) ML types. However we *can* project the same `U` value at multiple monomorphic ML types, and thus 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, it is possible to project object language expressions that would be untypeable were they written in ML down to well-typed ML values. For example, the following is 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. More interesting examples involve projection of untyped object language functions that switch more dynamically on the constructors of the universal datatype. If bindings such as

```
... ("ispair", embed (any-->bool) (fn UP _ => true | _ => false)),
    ("isint", embed (any-->bool) (fn UI _ => true | _ => false)), ...
```

are included in the environment, then one can implement some simple *generic* (or *polytypic* (Jeuring & Jansson, 1996)) functions, such as the following comparison function, which works for any type built from our base types and pairing:

```
- local val eleq = interpretclosed (read "
    let fun leq p = let val x = fst p in
                    let val y = snd p in
                     if isint x then leqint (x,y)
                     else if isstring x then leqstring (x,y)
                     else if ispair x then
```

```
                        and (leq (fst x,fst y),leq (snd x,snd y))
                  else if isbool x then implies x y
                  else if isunit x then true
                  else false
    in leq")
  in fun leq t p = project (t**t-->bool) eleq p
  end;
val leq = fn : 'a EP -> 'a * 'a -> bool
- leq int (3,4);
val it = true : bool
- leq string ("ho","hi");
val it = false : bool
- leq (int**string) ((3,"hi"),(4,"ho"));
val it = true : bool
```

With just a little more effort, we can switch on ML types (including function types), rather than the constructors of the universal type; this yields a 'poor man's' form of runtime type analysis. We extend the abstract type `'a EP` with a third component holding a syntax tree for the type `'a` in the obvious way so we can define an equality function

```
val tyeq : 'a EP * 'b EP -> bool
```

Then we can, for example, define

```
(* join : 'a EP * 'a -> 'b EP * 'b -> 'c EP -> 'c *)
fun join (t1,v1) (t2,v2) =
  fn t => if tyeq(t,t1) then project t (embed t1 v1)
          else if tyeq(t,t2) then project t (embed t2 v2)
          else raise Match
```

The `join` function allows one to simulate type-based dispatch:

```
- val plus = fn p => join (int**int-->int,Int.+)
                          (string**string-->string, String.^) p;
val plus = fn : 'a EP -> 'a
- plus (int**int-->int) (3,4);
val it = 7 : int
- plus (string**string-->string) ("one","two");
val it = "onetwo" : string
```

Utility Embedding and using polymorphic ML values is certainly useful.

It is hard to think of convincing uses of the ability to project untypable funtions as demonstrated in the fixpoint combinator example.

Dynamic types, runtime type analysis and generic functions are certainly useful in their own right, and their addition to statically typed languages such as ML has been (and continues to be) well-studied. Using a universal type and embedding/projection pairs is simple and neat, but limited and inefficient by comparison

with directly extending the language or using other techniques (e.g. ones using on type classes in Haskell, or the exception-based trick in ML that we will describe in the next section). Further discussion may be found in other work on dynamic typing, including (Abadi *et al.*, 1991; Henglein, 1992; Weirich, 2000; Weirich, 2001).

From the perspective of this paper, the main utility of type-passing using embedding/projection pairs is in impedance matching between ML and more dynamic object languages. Scripting languages often have late-binding, runtime types, or reflective operations, which one might expect to be difficult to project back to a statically-typed metalanguage. However, the combination of static polymorphism and dynamic type representations does allow many of these features to be interpreted in ML-like languages.

5 Embedding Datatypes

There is a natural, but inefficient, 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 `U` with one more constructor for tagging sum types[1]

```
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
```

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

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

the associated embedding/projection pair will be given by

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

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`.

Here are the actual definitions for `wrap`, `sum` and `mu`:

```
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)
```

[1] `UT` is not strictly necessary, as we could reuse the existing constructors for pairing and integers.

```
              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 that 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.

For example, this is how we 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
```

We could, of course, have written the embedding and projection functionals for `list` directly as recursive functions. The main advantage of using `wrap`, `sum` and `mu` is that they allow one to keep the universal type and the representation of embedding/projection pairs abstract.

Whilst our embedding of datatypes is semantically elegant, it is extremely inefficient. The problem is that there are multiple representations for datatype values, and each time one crosses the boundary between the two languages, it is converted in its entirety from one representation to the other. Each use of an embedded primitive operation involves at least two representation changes, which, for example, makes the above version of `map` have quadratic, instead of linear, time complexity.[2]

[2] 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.

An alternative approach is to keep values of recursive types in their metalanguage representation, by adding extra constructors to the universal datatype `U`. Adding cases can be done in an ad hoc manner, modifying the interpreter for the types relevant to a particular application, or in a slightly more modular way by making the initial definition of `U` be parametric, rather than immediately recursive, and then tying the recursive knot later; this technique is sometimes called *two-level types* and is described in more detail by Steele (1994), Sheard and Pasalic (2004) and Ramsey (2004). In ML it is also possible to use a well-known folklore[3] trick to extend `U` with new types more dynamically:

```
signature DYNAMIC =
sig
  type dyn
  val newdyn : unit -> ('a->dyn)*(dyn->'a)
end

structure Dynamic :> DYNAMIC =
struct
  exception Dynamic
  type dyn = exn
  fun newdyn () =
   let exception E of 'a
   in (fn a => (E a),fn (E a)=>a | _=> raise Dynamic)
   end
end
```

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 treat datatypes polymorphically: the embedded versions of different types of lists, and all of their operations, are unrelated to one another. Embedding at the `any` (that is, `U`) instantiation is no help, as the whole point was to keep datatypes values in their metalanguage representations. One also has to be careful not to apply `newtype` twice to the same ML type, since the resulting embeddings will be type incompatible.

Utility As we have said, the major disadvantage of our first solution to the problem of embedding datatypes is its poor performance. This is a shame, since it is

[3] I have been unable to discover who first observed that ML exceptions can be (ab)used in this way. The specfic code given here is adapted from Filinski's (1996) thesis.

simple, uniform, works in Haskell as well as ML and supports polymorphism. For prototyping, academic experimentation or in situations where the values of recursive datatypes are guaranteed to be small (lists of configuration options, perhaps) it is still viable, but in most real situations one would probably be forced to use one of the other techniques.

6 Monads

Since Moggi's (1991) work on using monads to structure denotational semantics, using monads (or monad transformers) to build interpreters in a modular way has become a popular functional programming technique. The basic idea is that by writing an interpreter parameterized by a monad one can then obtain interpreters for languages with a wide range of (combinations of) features ('notions of computation'), such as exceptions, state or control operators, simply by instantiating the monad appropriately and adding a small number of monad-specific operations.

For example, in a non-deterministic language, an expression of type `int` might be interpreted in $\mathbb{P}(\mathbb{Z})$ (i.e. as a set of integers) and one of type `int->int` as an element of $\mathbb{P}(\mathbb{Z} \to \mathbb{P}(\mathbb{Z}))$ (equivalently, a set of relations). In a language with exceptions, an expression of type `int` could be interpreted as an element of $E + \mathbb{Z}$ and one of type `int->int` as an element of $E + (\mathbb{Z} \to E + \mathbb{Z})$. Moggi's observation was that many of the operations on types accounting for the effectful behaviour of computations (such as $\mathbb{P}(-)$ and $E + (-)$) occur in the same places in the semantics, and have the structure of a monad (an operator on types equipped with two operations and satisfying some equations). More details concerning monadic interpreters may be found in, for example, (Wadler, 1992; Liang *et al.*, 1995; Liang & Hudak, 1996).

6.1 Monadic Translations, Extensionally

Given the usefulness of monadic interpreters, it is a natural question whether our embedding technique can be extended to interpreters for languages with arbitrary monadic notions of computation. At first sight, however, it seems unlikely that it can, unless the values that we embed are all first-order. 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.

Recall (Moggi, 1991; Benton *et al.*, 2002) that the monadic approach factors the semantics of a simply-typed lambda calculus as a syntactic translation into the computational metalanguage followed by the interpretation of the metalanguage in an appropriate category. There are actually three different translations (called *direct*, *lifted* and *call-by-value* by Benton and Wadler (1996)), we will only discuss the call-by-value (CBV) one here, as that is the only one we can embed with any generality in ML.

The CBV monadic translation $(\cdot)^*$ on simple types for a monad $(T, val(\cdot), let \cdot = \cdot \; in \; \cdot)$ is defined as follows:

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

$$\begin{aligned}(A \to B)^* &= A^* \to T(B^*)\\ (A \times B)^* &= A^* \times B^*\end{aligned}$$

with an associated translation on typing judgements

$$(\Gamma \vdash M : A)^* \;=\; \Gamma^* \vdash M^* : T(A^*)$$

where the translation on terms is defined by

$$\begin{aligned}&\text{Value translation } |\cdot|\text{:}\\ |x| &= x\\ |n| &= n\\ |\lambda x : A.M| &= \lambda x : A^*.M^*\\ |(V_1, V_2)| &= (|V_1|, |V_2|)\\ &\text{Expression translation } (\cdot)^*\text{:}\\ V^* &= \mathit{val}|V|\\ (M\ N)^* &= \mathit{let}\ f = M^*\ \mathit{in}\ (\mathit{let}\ x = N^*\ \mathit{in}\ f\ x)\\ (M,\ N)^* &= \mathit{let}\ x = M^*\ \mathit{in}\ (\mathit{let}\ y = N^*\ \mathit{in}\ \mathit{val}(x, y))\\ (\pi_i M)^* &= \mathit{let}\ p = M^*\ \mathit{in}\ \mathit{val}(\pi_i p)\end{aligned}$$

Factoring the interpretation of elements of our AST datatype `Exp` through the monadic translation is straightforward, but embedding ML values, which will have to have the monadic translation applied to them too, seems problematic. The translation is defined by induction over syntax, but we do not have access to the syntax of compiled metalanguage values; the only way of examining a function is to call it.

One then wonders if there is a family of functions tran_A of type $A \to A^*$ for each metalanguage type A such that for all $V : A$, $\mathit{tran}_A\ V = |V| : A^*$. But such functions cannot be defined in the 'pure' part of Standard ML (or in Haskell). To understand intuitively why, consider the instance at type $A = (\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$, which would produce results of type $(\mathtt{int} \to T(\mathtt{int})) \to T(\mathtt{int})$. There are two ways to argue:

- The semanticist's argument. There are λ-calculus values V_1, V_2 and instances of T such that $V_1 = V_2 : A$ but for which $|V_1| \neq |V_2| : A^*$. An example would be

  $$\begin{aligned}V_1 &= \lambda f : \mathtt{int} \to \mathtt{int}.f1\\ V_2 &= \lambda f : \mathtt{int} \to \mathtt{int}.(\lambda z : \mathtt{int}.f1)(f1)\end{aligned}$$

  with T being the state monad.
- The programmer's argument. Just try to write tran_A. One quickly realizes that there are no very interesting functions of this type that work for arbitrary T, essentially because T occurs in a negative position in A^*. We would have to have

  $$\mathit{tran}_A F \;=\; \lambda f : (\mathtt{int} \to T(\mathtt{int}))\ldots\ (F\ \mathit{something})\ \ldots$$

but there is no sufficiently uniform way to produce a suitable *something* of type $(\texttt{int} \to \texttt{int})$ from the $f : (\texttt{int} \to T(\texttt{int}))$ we have in our hand.

These arguments can be made quite precise in the context of the 'pure' CBV λ-calculus, in which the only side-effect is divergence. But they are not so convincing in the case of the real ML language, in which expressions can have a range of side-effects. In particular, the two functions V_1 and V_2 are *not* actually equal (observationally equivalent) in ML. Furthermore, for some particular instances of T we actually *can* write $tran_A$ by moving between explicit and implicit representations of side-effects. For example, in the simple case of integer-valued state (with the obvious monad structure):

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

fun tranA (F: (int->int)->int) =
  fn (f : int -> int t) => fn state =>
    let val r = ref state
        fun wrappedf n = let val (newstate,result) = f n (!r)
                         in (r := newstate; result)
                         end
        val result = F wrappedf
    in (!r, result)
    end
```

Can we do this for every monad definable in ML? If so, can we do it parametrically in the monad? If so, can we do it parametrically in the type A?

In SML with only the implicit side-effects required by the language definition (references, exceptions, IO), the answer to the first question is 'no', because control is not expressible in terms of the other effects. Amazingly, however, if the language also has first-class control amongst its implicit effects (as SML/NJ and MLton do), 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). Filinski showed firstly that the combination of control and state is a universal effect, which can simulate any other monadic effect. (There is a monad retraction from any definable monad into the continuations monad.) Secondly, he showed how one may define reification and reflection operations allowing one to move between implicit (opaque) and explicit (transparent) 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 that 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 implicit and explicit 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 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
  val translate : ('a,'astar) TR -> 'a -> 'astar
  val untranslate : ('a,'astar) TR -> '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
end
```

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

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

  type ('a,'astar) TR = ('a->'astar)*('astar->'a)
  fun translate (t,n) = t
  fun untranslate (t,n) = n
```

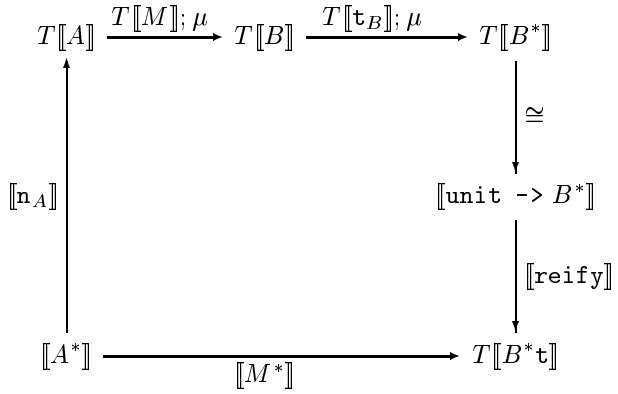
```
  val base = (I,I)
  val int = base
  val string = base
  val unit = base
  val bool = base

  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))))

end
```

Writing $[\![\cdot]\!]$ for the underlying monadic denotational semantics of ML, with monad (T, μ, η), and using $(\cdot)^*$ for the source-to-source CBV translation using the ML type constructor `'a t`, the following slightly informal diagram may help explain what is going on semantically in the translation case for an ML function `fn x=>M` of type A `->` B:



If, for example, we apply the reflection and translation functors to a structure defining our monad for integer-valued state (`'a t = int -> int * 'a`) with 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.

6.2 Embedding Monadic Interpreters

We can now combine embedding/projection pairs with the extensional translation to write interpreters that are parameterized 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. The universal type would then look like

```
datatype U = UF of U -> U M.t | UP of U*U | ...
```

and ML values would then be lifted to the object language by first translating them and then embedding them. However, as we have seen in the previous section, it is possible to move quite smoothly between implicit and explicit representations of the monad, allowing us to take a second, neater approach. We will keep the monad *implicit* in the code for the interpreter, which allows us to leave the signatures and structures for the interpreter and embedding/projection pairs completely unchanged. We then write a new functor `MEmbeddings` that pairs each embedding/projection pair with its corresponding monadic translation/untranslation pair, yielding type-indexed *quads*. The signature is

```
signature MEMBEDDINGS =
sig
  structure Tr : TRANSLATION
  (* SML'97 datatype replication:
     imports constructors (qua constructors, not just values) too *)
  datatype U = datatype Embeddings.U

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

  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 Tr.R.M.t) QUAD
end
```

whilst the matching functor looks like this:

```
functor MEmbeddings (R : RMONAD) : MEMBEDDINGS   =
struct
  structure Tr = Translation(R)
  structure E = Embeddings
  datatype U = datatype E.U
  type ('a,'astar) QUAD = ('a E.EP) * ('a,'astar) Tr.TR

  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)))

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

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

  fun (ep,tn) ** (ep',tn') = (E.**(ep,ep'), Tr.**(tn,tn'))
  fun (ep,tn) --> (ep',tn') = (E.-->(ep,ep'), Tr.-->(tn,tn'))
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. If these operations are given by embedding ML values as values in the object language, then the ML types of those values will *already* be in the image of the CBV translation: we embed them using a new embedding function `membed` that first untranslates its argument and then embeds the result. For example, in the case of the simple exception monad given by ML's `option` datatype, the extra operations might be

```
  (*  throw : unit -> 'a option *)
  fun throw () = NONE
  (*  try : (unit -> 'a option)*(unit -> 'a option) -> 'a option *)
  fun try (block, alt) = case block () of
                             NONE => alt ()
                           | SOME v => SOME v
```

Observe that the types of `throw` and `try` are are the translations of `unit->'a` and `(unit->'a)*(unit->'a)->'a`, respectively, so those are the types their untranslated and embedded versions will appear to have from the point of view of the object language. The alternative approach, which often leads to more palatable object language syntax, is to add the monad operations as new language constructs, rather than value bindings. This involves adding cases to the parser and interpreter, either in terms of `membed` or by manually calling `reify` and `reflect`.

The interpretation of an object language expression is a generally a computation; when we project such things back to ML we have to get an ML value of a translated type, i.e. with the effects represented explicitly. For example, when projecting a non-deterministic integer computation back as an ML value, we need to get a list of integers, rather than an integer (not just because we do not want to lose multiple results, but also because there is no integer to return in the case the list is empty). Thus our `project` function needs to use reification, projection and translation to return a value of type `'astar Tr.R.M.t`. The choice of argument types also requires explanation. The reader might have expected to see

```
val project : ('a,'astar) QUAD -> U -> 'astar Tr.R.M.t
```

but this would be no good: by the time we have a *value* of type `U`, the implicitly represented effects of the computation have already happened and cannot subsequently be reified by `project`. The obvious solution is to pass a thunk instead:

```
val project : ('a,'astar) QUAD -> (unit -> U) -> 'astar Tr.R.M.t
```

and this is essentially what we do, except that as most of the computations we wish to project are already applications of the curried function `interpret`, re-abstracting seems gratuitous. Instead we generalize, allowing an arbitrary `U`-returning function, `f`, and an argument, `x`, to be passed to `project`; the effectful function (e.g. a partial application of `interpret`) is applied to the argument (e.g. a dynamic environment) within the scope of the thunk passed to `reify`.

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 that 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 *)
```

Again, 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 that 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
```

Utility The extensional monadic translation is potentially useful in its own right. Common criticisms of the monadic approach to programming are that the monads 'infect' one's entire program and that one must write everything monadically from the start to be able to benefit from the ability to change the monad later. The fact that Filinski's techniques allow one to have one's monadic cake and eat it – switching at will between implicit and explicit representations of the monad – is somwhat underappreciated. Nevertheless, in the presence of higher-order functions, making appropriately nested calls to the raw reification and reflection functions is potentially error-prone. Our type-indexed translation and untranslation technique wraps up Filinski's operations in a much more convenient form.

The ability to parameterize embedded interpreters by any monad significantly increases the usefulness of our basic idea. Convenient domain-specific languages often have an implicit monadic effect, such as state (most command languages) or non-determinism (e.g. query languages, such as that of Fernandez, Siméon and Wadler (2001)). Using the embeddings presented here one can write the application in a straighforward explicit style and then push the monads under the hood in the interpreter, whilst still smoothly combining effectful operations with non-monadic ones lifted from the metalanguage. A related situation in which the monadic embeddings seem useful is in interfacing ML with external language implementations with their own notion of computation. A simple example is an interpreter with its own state, such as that used by Ramsey (2003); a more exotic one might be a logic programming language.

It is worth reiterating that for particular concrete monads one may well be able to define a monadic translation by hand, without using Filinski's reflection operations. The great advantage of the continuation-based technique is its genericity.

7 Processes

So far, we have only considered embedding interpreters for object languages that are essentially CBV λ-calculi. There is another tractable foundational language that 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 that 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 use the choice-free, asynchronous, polyadic π-calculus (Honda & Tokoro, 1991; Boudol, 1992), which is also the calculus on which the Pict language (Pierce & Turner, 2000) is built. (Choice or synchronous primitives are easily added if required, but would not materially affect our embeddings.) Note that this is a first-order calculus – the only values that may be transmitted are names or of primitive types – and that this makes the translation of functions more interesting than it might be in a higher-order calculus in which processes may be transmitted along channels.

Like the monadic translation, the interpretation of functions as processes is usu-

ally presented by induction on terms, and target contexts can make distinctions between λ-terms that are extensionally equivalent in a more standard 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.

7.1 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. It is possible, but not entirely trivial, to use our earlier monadic translation techniques to embed and project ML values into and out of an interpreter for the π-calculus that is structured as a denotational semantics. The denotational approach appears 'purer' and allows ML code to manipulate multiple process configurations explicitly as ordinary functional values. On the other hand, that manipulation is tricky and has to be done whenever one uses projected values, since they end up having complex ML types. Furthermore, the monad for processes ends up being a form of continuations, and it seems slightly perverse to use all the general machinery of continuation-based monadic reflection just to recover the continuation monad one started with. We will therefore take a more imperative approach to concurrency in this section, implementing a single, implicitly referenced, imperatively mutated, pool of processes directly in terms of references and control operations. Choosing imperative concurrency leads to a concise implementation and straightforward embeddings and projections.

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, booleans or the unit value. The combinators `Nil`, `Nu`, `Par`, `Receive`, `BangReceive` and `Send` correspond to the nil process $\mathbf{0}$, 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 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 performs a small bit of gymnastics to create a continuation for its argument, place it on `readyQ` and return to its caller. (A `fork` that immediately transfers control to its argument is simpler, but the version given is a little more friendly to use from the top level, as one can schedule several processes before calling `sync()`.) 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. We'll assume a structure `Interpret` which matches the following signature:

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

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

  type 'a chan = ('a Q.queue) * ('a C.cont Q.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 = Q.mkQueue() : unit C.cont Q.queue
  fun new() = Name (Q.mkQueue(),Q.mkQueue())

  fun scheduler () = C.throw (Q.dequeue readyQ) ()

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

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

  fun fork p =
    let val newThread =
      C.callcc (fn k1 => (C.callcc (fn k2 => (C.throw k1 k2));
                          (p ()) handle _ => (); scheduler ()))
    in Q.enqueue(readyQ,newThread)
    end

  fun sync () =
      if Q.length readyQ = 0 then ()
      else (C.callcc (fn k=> (Q.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(BangReceive(VN ch,f), f v))
  fun Nu f () = f (newname()) ()
end
```

Fig. 2. Implementation of Continuation-Based Coroutines

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

and which allows object programs to be written in a syntax like that of Pict:

```
- val pp = read "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 ();
pingpongpingpongpingpongpingpongpingpongpingpong...
```

The `pp` process comprises three parallel (`|`) subprocesses in the scope of two *new* channel names, *ping* and *pong*. The first process repeatedly waits for (*?**) an empty message (*[]*) on channel *ping*; when one arrives (`=`), it sends (*!*) the string "*ping*" along the channel *echo* and (`|`) sends a signal along *pong*. The second process does the same, but with the roles of *ping* and *pong* reversed, whilst the third process starts things off by sending an initial signal to *ping*. The result is that the two processes alternately signal one another endlessly, sending a stream of "*ping*" and "*pong*" messages along the channel *echo*. The identifier *echo* is bound to a name by an environment in the structure `Builtins`, which 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. We note in passing that, although there is no preemption, the round-robin behaviour of the scheduler and the implementation of replicated input make pure π-calculus processes rather fair. If, for example, one adds another parallel copy of the *pong* process, printing a different string, then the two copies will alternate strictly.

7.2 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:

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

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 fresh 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 an example, here is the (value) translation of the *apply* function $\lambda f.\lambda x.fx$ as a process located at a channel `c` given in terms of our Pict-like concrete syntax:

```
c?*[f r] = new d r!d |
           d?*[x s] = new q q!f | (q?v = new t t!x | t?w = v![w s])
```

After some internal communications (administrative reductions) this becomes

```
c?*[f r] = new d r!d | d?*[x s] = f![x s]
```

As stressed by, for example, Boudol (1997) and Sangiorgi & Walker (2001), this encoding is essentially a CPS translation. Our implementation of the encoding combines our usual embedding/projection pairs technique with the continuation-based coroutine operations. Here is the signature:

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

  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
end
```

This signature is apparently simple – it is essentially the same as that we presented back in Section 2, 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

```
structure Embeddings :> EMBEDDINGS =
struct
  structure P=Process
  datatype BaseValue = datatype P.BaseValue
  type 'a EP = ('a->BaseValue)*(BaseValue->'a)
  fun embed (ea,pa) v = ea v
  fun project (ea,pa) c = pa c

  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 = P.new()
                  fun action () = let val [ac,VN rc] = P.receive c
                                      val _ = P.fork action
                                      val resc = eb (f (pa ac))
                                  in P.send(rc,[resc])
                                  end
              in (P.fork action; VN c)
              end,
      fn (VN fc) => fn arg => let val ac = ea arg
                              val rc = P.new ()
                              val _ = P.send(fc,[ac,VN rc])
                              val [resloc] = P.receive(rc)
                          in pb resloc
                          end
    )
  fun (ea,pa)**(eb,pb) =
   ( fn (x,y) => let val pc = P.new()
                     val c1 = ea x
                     val c2 = eb y
                     fun action () = let val [VN r1,VN r2] = P.receive pc
                                     in (P.fork action;
                                         P.send(r1,[c1]); P.send(r2,[c2]))
                                     end
                 in (P.fork action; VN pc)
                 end,
     fn (VN pc) => let val r1 = P.new()
                   val r2 = P.new()
                   val _ = P.send(pc,[VN r1, VN r2])
                   val [c1] = P.receive r1
                   val [c2] = P.receive r2
              in (pa c1, pb c2)
              end
   )
end
```

Fig. 3. Extensional π-calculus Translation

projections at function and product types are defined in terms of the coroutine primitives.[4]

The embedding at a function type takes a value `f`, generates a new name `c` and then forks a thread that 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 that 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 a new located process that 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.

We can now give the definition of the `Builtins` structure:

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

structure Builtins :> BUILTINS =
struct
  open Embeddings
  infix **  infixr -->
  structure I=Interpret
  structure P=Process

  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)),
     ...]
```

[4] Using our π-calculus combinators might be more attractive, but I have not managed to make such a definition behave correctly from the top-level loop of SML/NJ. In any case, one has to break their abstraction to get names into and out of `Processes`.

```
  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 _ => P.newname()) specials)
  val _ = map (fn (_,s) =>
          schedule (I.interpret (I.read s,static) dynamic)) specials
end
```

The evaluation of `embs` has a top-level side-effect, returning new channels for interaction with the processes corresponding to the ML values. The string names for those channels comprise the pervasive static environment, whilst the channel values themselves make up the dynamic environment. There are some further top-level effects associated with interpreting, naming and scheduling the processes in `specials`, which are programmed explicitly in the object language rather than being embedded from the metalanguage. Note that the definition of the `echo` process, which we used earlier, is not entirely trivial: it is an asynchronous (one-way) wrapper that sends a message to the process embedding the ML `print` function, but directs responses to be sent to a process that 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 print the expected value. 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 that 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`, and `printn` which is `print o itos`, then we can do

```
- test "new r1 new r2 new c appupto![printn r1] |
        (r1?f = c?*[n r] = r![] | f![n devnull]) |
```

```
        appupto![c r2] | r2?g = g![10 devnull]";
00100110220130120123102342013430124541235523466345745656767878899 10
```

We have applied `appupto` to `printn` yielding a process located at `f` that will sequentially print the integers from 0 to n. We then define a functional process at `c` that 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
```

Utility The embedded π-calculus interpreter demonstrates that a theoretical construction has a more direct implementation in a real programming language than one might have expected. In the form presented here, however, its main value seems to be didactic. One apparent problem is that the concurrency we have is non-preemptive: embedded ML functions have to run to completion before any other process will be scheduled. However, it is fairly easy to modify the code given here to be preemptive under SML/NJ by using an operating system timer and asynchronous signal to force a context-switch at regular intervals. This is the way in which CML is implemented; more details are given by Reppy (1999). Even so, it is not clear that the 'first order' style of embedding given here is the most natural one for a concurrent command language, or that factoring an application so the concurrency is encapsulated in the command language (rather than running throughout the rest of the application) is useful. Simply applying the basic ideas of Section 2 in a more pervasively concurrent language, such as CML, seems more practical.

Some version of our π-calculus translation does appear potentially useful in building distributed systems. Most traditional remote procedure call (RPC) systems, as well as more recent variants such as web services (World Wide Web Consortium, 2002), restrict transmissible values to be first order. Ohori and Kato (1993) described a uniform type-directed method for generating stub code for *higher-order* RPC in a distributed CBV language, dML. Their translation is based on passing functions as dynamically generated remote handles and has been used as a case study in work on intensional type analysis by Harper and Morrisett (Harper & Morrisett, 1995) and by Trifonov *et al.* (2000). The original dML language is single-threaded with synchronous (blocking) remote calls and one can easily implement something similar in any ML system using only our basic technique. The continuation-based variant presented here would allow one to perform higher-order RPC in a more realistic concurrent and asynchronous setting.

8 Discussion and Related Work

We have shown how the simple idea of representing types by pairs of functions, one mapping out of the type and one mapping back into it, can be applied, focussing on the way in which it allows one to move between the object and metalanguages when writing interpreters. The idea is surprisingly robust, extending from the metacircular case (in which the two languages are (essentially) the same) to situations in which the languages appear quite different.

Modelling types as retracts or projections of a universal domain is an old idea in denotational semantics, going back to the early work of Scott (1976) and Mc-

Cracken (1979). It is only comparatively recently that related techniques have been recognised as useful in functional programming. Danvy (1998b) credits Filinski and Yang with having used our basic construction in 1995 and 1996, respectively, to implement normalization by evaluation (NBE), whilst Kennedy and I discovered it in 1997 in the contexts of writing picklers (also known as serializers, or marshallers) (Kennedy, 2004) and interpreters, respectively. Similar type-directed constructions have also been used in implementing `printf`-like string formatting (Danvy, 1998a) and in generic programming. We have already mentioned Ramsey's (2003; 2004) independent use of the same basic technique to embed the LuaML interpreter.

The applications described here are most similar to work on NBE and type-directed partial evaluation (TDPE) (Danvy, 1996; Hatcliff *et al.*, 1998; Danvy, 1998b; Yang, 1998). NBE produces syntactic normal forms from compiled code via type-indexed reification and reflection functions between metalanguage types and a syntactic 'universal' type. Indeed, were it not for the fact that NBE does not apply to all the types and values in which we are interested (e.g. infinite coproducts like `int` and recursive functions), 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 (2004) pickler combinators use essentially the same technique.

Amongst the many other contexts in which similar type-indexed pairs of functions arise are Leroy's (1992) work on compiling polymorphism using specialized representations, Findler and Felleisen's (2002) work on run-time monitoring of contracts for higher-order functions and the Cousots' (1977) abstract interpretation framework for static analyses. At a rather abstract level, the same ideas appear repeatedly because we are dealing with a situation which is ubiquitous in computer science: relating multiple interpretations of typed programming languages. If the language is fixed then the interpretations may be at different levels of abstraction (as in abstract interpretation), choose different data representations or relate either to different locations (as in distributed computation) or to different times (as in staged computation). The details of the relations between interpretations may vary, but the same basic semantic techniques of adjunctions, logical relations, embedding/projection pairs and modal logic are applicable in all cases.

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 simple interpreters. (Of course, the monadic and π-calculus embeddings cannot be implemented in Haskell, since they rely on first-class control.) One can 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 `'a*'b` (giving the appropriate constructions on embeddings and projections). Then one can just write `embed v` instead of `embed` τ `v`, though polymorphic values still need to be explicitly constrained to resolve the ambiguity. Rose (1998) has applied just this technique to an implementation of TDPE in Haskell.

There is, of course, ample scope for formulating, and attempting to prove, some correctness results for the constructions presented here. Reynolds's (2000) careful analysis of the relationship between the 'intrinsic' (meanings are assigned to typing judgements) and 'extrinsic' (meanings are assigned to untyped terms and types are interpreted as properties of those meanings) approaches to semantics should provide a starting point.

I should like to thank Olivier Danvy, Andrew Kennedy, Simon Peyton Jones, Norman Ramsey, Philip Wadler and the anonymous referees for helpful discussions and comments on earlier drafts of this paper.

References

Abadi, M., Cardelli, L., Pierce, B., & Plotkin, G. (1991). Dynamic typing in a statically-typed language. *ACM transactions on programming languages and systems (TOPLAS)*, **13**(2), 237–268.

Abelson, H., Sussman, G. J., & Sussman, J. (1985). *Structure and interpretation of computer programs (first edition).* MIT Press.

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs (second edition).* MIT Press.

Bentley, J. (1986). Programming pearls: Little languages. *Communications of the ACM*, **29**(8), 711–721.

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

Benton, N., Hughes, J., & Moggi, E. (2002). Monads and effects. Barthe, G., Dybjer, P., Pinto, L., & Saraiva, J. (eds), *Applied semantics, advanced lectures.* Lecture Notes in Computer Science, vol. 2395. Springer-Verlag.

Benton, P. N., & Wadler, P. 1996 (July). Linear logic, monads and the lambda calculus. *Proceedings of the 11th IEEE symposium on logic in computer science (LICS).*

Boudol, G. 1992 (May). *Asynchrony and the π-calculus.* Tech. rept. Research Report 1702. INRIA.

Boudol, G. (1997). The pi-calculus in direct style. *Pages 228–241 of: Conference record of the 24th ACM symposium on principles of programming languages (POPL).*

Clinick, A. (2000). *Introducing JScript.NET.* MSDN Library: `http://msdn.microsoft.com/library/`.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference record of the 4th ACM symposium on principles of programming languages (POPL).* ACM.

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

Findler, R., & Felleisen, M. (2002). Contracts for higher-order functions. *Proceedings of the international conference on functional programming (ICFP).*

Harper, R., & Morrisett, G. (1995). Compiling polymorphism using intensional type analysis. *Conference record of the 22nd ACM symposium on principles of programming languages (POPL).*

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.

Henglein, F. (1992). Dynamic typing. Krieg-Brückner, B. (ed), *Proceedings of the 4th european symposium on programming (ESOP).* Lecture Notes in Computer Science, no. 582. Springer-Verlag.

Honda, Kohei, & Tokoro, Mario. (1991). An object calculus for asynchronous communication. *Pages 133–147 of:* America, P. (ed), *Proceedings of the European conference on object-oriented programming (ECOOP).* Lecture Notes in Computer Science, vol. 512. Springer-Verlag.

Hudak, Paul. 1998 (June). Modular domain specific languages and tools. *Pages 134–142 of: Proceedings of 5th international conference on software reuse.*

Hugunin, J. (1997). Python and Java: The best of both worlds. *Proceedings of the 6th international Python conference.*

Jeuring, J., & Jansson, P. (1996). Polytypic programming. *Pages 68–114 of:* Launchbury, J., Meijer, E., & Sheard, T. (eds), *Advanced Functional Programming, Second International School.* Springer-Verlag. LNCS 1129.

Kennedy, A. (2004). Functional pearl: Pickler combinators. *Journal of functional programming.* to appear.

Leroy, X. (1992). Unboxed objects and polymorphic typing. *Pages 177–188 of: Conference record of the 19th ACM symposium on principles of programming languages (POPL).* ACM.

Liang, S., & Hudak, P. 1996 (Apr.). Modular denotational semantics for compiler construction. *Proceedings of European symposium on programming (ESOP).*

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL).*

McCracken, N. 1979 (June). *An investigation of a programming language with a polymorphic type structure.* Ph.D. thesis, Syracuse University.

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.

Mozilla Organisation, The. *Rhino: Javascript for Java.* `http://www.mozilla.org/rhino/`.

Ohori, A., & Kato, K. (1993). Semantics for communication primitives in a polymorphic language. *Pages 99–112 of: Conference record of the 20th ACM symposium on principles of programming languages (POPL).*

Ousterhout, J. K. 1990 (Jan.). Tcl: An embeddable scripting language. *Pages 133–146 of: Proceedings of the winter USENIX conference.*

Ousterhout, J. K. (1998). Scripting: Higher-level programming for the 21st century. *IEEE computer*, **31**(3), 23–30.

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.

Ramsey, N. (2003). Embedding an interpreted language using higher-order functions and types. *ACM SIGPLAN 2003 workshop on interpreters, virtual machines and emulators.*

Ramsey, N. (2004). *ML module mania: A type-safe, separately compiled, extensible interpreter.* `http://www.eecs.harvard.edu/~nr/`.

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

Reynolds, J. C. 1972 (Aug.). Definitional interpreters for higher-order programming languages. *Pages 717–740 of: Proceedings of the ACM annual conference, Boston.* Reprinted in (Reynolds, 1998).

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

Reynolds, J. C. 2000 (Dec.). *The meaning of types – from intrinsic to extrinsic semantics.* Tech. rept. BRICS RS-00-32. BRICS, University of Aarhus.

Rhiger, M. (2003). A foundation for embedded languages. *ACM transactions on programming languages and systems (TOPLAS)*, **25**(3).

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 (LICS).* IEEE.

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

Scott, D. (1976). Data types as lattices. *SIAM journal of computing*, **4**.

Sheard, T., & Pasalic, E. (2004). Functional pearl: Two-level types and parameterized modules. *Journal of functional programming.* To appear.

Shivers, O. (1996). A universal scripting framework, or lambda: The ultimate "little language". *Pages 254–265 of:* Jaffar, J., & Yap, R. H. C. (eds), *Concurrency and*

parallelism, programming, networking, and security. Lecture Notes in Computer Science, vol. 1179. Springer-Verlag.

Slind, K. 1991 (Nov.). Object language embedding in Standard ML of New Jersey. *Proceedings of the 2nd ML workshop.* Revised February 1996.

Steele, G. L. Jr. (1994). Building interpreters by composing monads. *Conference record of the 21st ACM symposium on principles of programming languages (POPL).*

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

Trifonov, V., Saha, B., & Shao, Z. (2000). Fully reflexive intensional type analysis. *International conference on functional programming (ICFP).* ACM.

van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, **35**(6), 26–36.

van Rossum, G. 2003 (July). *Extending and embedding the Python interpreter.* Release 2.3 `http://www.python.org/doc/current/ext/ext.html`.

Wadler, P. (1992). The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL).*

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

Weirich, S. 2000 (Sept.). Type-safe cast: Functional pearl. *Pages 58–67 of: Proceedings of the 5th ACM SIGPLAN international conference on functional programming (ICFP).*

Weirich, S. (2001). Encoding intensional type analysis. *Pages 92–106 of:* Sands, D. (ed), *10th European symposium on programming (ESOP).* Lecture Notes in Computer Science, vol. 2028. Springer.

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 international conference on functional programming (ICFP).*