Undoing Dynamic Typing (Declarative Pearl)

Nick Benton

Microsoft Research
nick@microsoft.com

Abstract. We propose undoable versions of the projection operations used when programs written in higher-order statically-typed languages interoperate with dynamically typed ones, localizing potential runtime errors to the point at which a dynamic value is projected to a static type. The idea is demonstrated by using control operators to implement backtracking projections from an untyped Scheme-like language to ML.

1 Introduction

When working in a statically typed language one often has to deal with data whose types cannot be fully determined, or at least, fully checked, at compile time. Data read at run time from the console or persistent storage and calls to dynamically-linked (local or remote) programs or services must be subject to runtime checks if type safety is to be maintained.

In a typed language, dynamic data is usually given some rather uninformative 'catch all' static type; at a low level this might be `string` or `byte[]`, whilst higher level examples include `Object`, `Dynamic` and `IUnknown`. The interface between the statically checked and unchecked worlds is provided by a collection of *projection*, *(down)cast*, *coercion*, *retraction* or *unmarshalling* operations mapping values of the single dynamic type to various particular static types, and complementary *embedding*, *(up)cast*, *section* or *marshalling* operations going the other way. Projection operations in an ML-like language might have signatures along the lines of:

```
val toInt : Dynamic -> int
val toBool : Dynamic -> bool
val toIntToInt : Dynamic -> (int->int)
```

but these operations are naturally partial. `toInt`, for example, will typically raise an exception in the case that its argument turns out to be a `Dynamic` value representing a string or a function. It is good practice to make this possibility more explicit and instead type the projection with an `option`:

```
val toInt : Dynamic -> int option
```

The programmer then has to explicitly match on the returned value before using it, making it harder to forget to deal with the case of failure.[1]

There are actually two rather different classes of projection. In the case that values of type `Dynamic` carry an explicit representation of their types (usually called a *tag*, or *runtime type information*) *and* that representation can be trusted, a projection operation can essentially just check the tag and, assuming it matches, proceed to use the underlying value with no further checks. This is the way many proposals for adding dynamic typing to statically typed languages work [1] and is common when one program unmarshalls values that were originally marshalled to persistent storage by a (trusted) program written in the same typed language.

The second case is that in which dynamic values are untagged, incompletely tagged, or the tags are potentially unreliable. Safely projecting values of functional types (or very large values of simple datatypes) cannot then be done all in one go. This is generally the situation when making foreign calls to functions written in an untyped (or differently-typed) language, or when calling a remote function in another address space or over a network. Consider, for example, linking to a remote service that is supposed to compute some function on integers. The service may well have some attached metadata (e.g. WSDL) that we can check at runtime to see that it conforms to the programmer's expectation that there is an operation there that accepts and returns integers. One would expect a projection

```
val toIntToInt : url -> (int->int) option
```

that connects to the service to retrieve and check the metadata, returning `NONE` if it fails to match and `SOME f` it matches, where `f` is an ML wrapper function that sends the service a marshalled version of its argument and returns the unmarshalling of the service's response. But we cannot necessarily trust the metadata we went to all the trouble of checking, so the wrapper usually also incorporates a check that *every* returned value *is* an integer, and raises an exception if that ever fails. Now the programmer has also to deal with the possibility of failure each time he applies the function wrapping the service; we should really have made that possibility explicit by typing the projection as

```
val toIntToInt : url -> (int -> int option) option
```

where the pattern for general higher-order types is that we have to add an `option` in every positive position. A useful point of view is that projection functions wrap untyped values with code that dynamically monitors their adherence to a *contract* associated with the type, in the sense of 'Design by Contract' [14, 8].

Higher-order programming in the presence of all these potential runtime errors is, however, painful. The situation is especially bad if one tries to deal with

[1] One might choose instead to map inappropriate elements of `Dynamic` either to divergence or to some more defined default value of the target type. Although the first alternative is well-behaved from a denotational perspective, neither has good software engineering properties.

more than one potentially-misbehaving untyped function at the same time (e.g. passing one as an argument to another), in which case impedance matching is a problem and the correct assignment of blame, and hence what error handling is appropriate, can be tricky to ascertain.

What we would really like is to turn the second case into the first, at least from the point of view of the programmer. The initial projection of a dynamic value to a given static type may or may not succeed, but if it does then the programmer should have a typed value in his hand that he can use without further fear of failure. The message of this paper is that we can achieve this goal by making projections undoable: the projection of a dynamic value to a static type may provisionally succeed but subsequently be rolled back to fail retrospectively should runtime type errors (contract violations) occur. We will use control operators to give an implementation of undoable projections from an untyped interpreted language to ML.

2 Background: Embedded Interpreters

In this section, based on an earlier paper [2], we briefly recall how embedding-projection pairs may be used to translate higher type values between typed (ML-like) and untyped (Scheme-like) languages, focussing, for concreteness, on the situation in which the untyped language is the object language of an interpreter written in the typed metalanguage. The underlying semantic idea here is just that of interpreting types as retracts[2] of a suitable universal domain, which goes back to work of Scott [17] in the 1970s, though the realization that this is both implementable and useful in functional programming seems only to have dawned in the mid 1990s [20].

Our starting point is an ML datatype modelling an untyped call-by-value lambda calculus with constants:

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

An interpreter for an untyped object language, mapping abstract syntax trees to elements of `U` is then essentially just a denotational semantics. We assume the existence of a parser for a readable object language (typeset in *`italic`*) and let `pi : string -> U` be the composition of the parser with the intepretation function.

The idea of embedded interpreters 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:

[2] Recall that a *section-retraction* pair comprises two morphisms $s : X \to Y$ and $r : Y \to X$ such that $s; r = id_X$. We say X is a *retract* of Y. *Embedding-projection* pairs are a special case: if X and Y are posets, s and r are monotone and additionally $r; s \sqsubseteq id_Y$ then s is an *embedding* and r is a *projection*.

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

  val unit   : unit EP
  val bool   : bool 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
```

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.

Embeddings and projections let one smoothly move values in both directions between the typed and untyped worlds, as demonstrated in the following, rather frivolous, example in which we project an untyped (and untypeable) fixpoint combinator to an ML type and apply it to a function in ML:

```
- let val embY = pi "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 factorial = polyY int int
         (fn f=>fn n=>if n=0 then 1 else n*(f (n-1)))
  in factorial 5
  end;
val it = 120 : int
```

The above is simple, neat and all works very nicely in the case that untyped values play by the rules and are used correctly. But the code is something of a minefield, being littered with deeply buried non-exhaustive `Match` and `Bind` exceptions. Our earlier paper said

> ... these exceptions *should* be caught and gracefully handled, but we will omit all error handling in the interests of space and clarity.

but, in fact, anything other than letting the exceptions propagate up to the top is remarkably tedious and difficult to achieve by hand. Here we will show

how much of that error handling can be built into the embedding infrastructure instead. The SML code that follows relies on `call/cc`, which is supported by both SML/NJ and MLton (though MLton's implementation takes time linear in the current depth of the control stack). There is also a (linear time) `call/cc` library for the OCaml bytecode compiler.

3 Retractable Retractions

There are various ways in which the simple embedded interpreter of our previous work can go wrong. The first is that object programs can contain runtime errors all by themselves, without any attempt being made to cast them to ML types. So, whilst this is OK:

```
- pi "let val x = 4 in x";
val it = UI 4 : U
```

this is not:

```
- pi "let val x = 4 in x 3";
uncaught exception Bind
  [nonexhaustive binding failure]
  raised at: Interpret.sml:37.45-37.63
```

As we are going to be playing fancy games with control flow shortly, it is a good idea to replace these exceptions with something simpler and more explicit. To this end, we add an explicit error constructor `UErr` to our universal type, as is commonly done in denotational semantics [18, p.144][1]. The definition is now

```
datatype U = UF of U->U | UP of U*U | UI of int | US of string
           | UUnit | UB of bool | UErr
```

and we modify the interpreter to yield `UErr` when it would previously have raised an exception, which includes making all the language constructs strict in (i.e. preserve) `UErr`.[3] An extract of the interpreter code is shown in Figure 1; this is entirely standard, though note that we have separated the binding times of variable names and values in environments. We omit the definition of `Builtins`, which uses embedding to add a few pervasives, including arithmetic and comparisons, to the environment.

We now turn to revising the embedding-projection pairs. As one might expect from our initial discussion, we change the signature to reflect the fact that projection will now be partial:

[3] One could make the code slightly shorter and more efficient by sticking with implicit exceptions in place of `UErr`, but the choice we have made makes what is going on slightly clearer. In particular, we do not have to worry about interactions between handlers and continuations, as it is now obvious that there are no potentially uncaught exceptions lurking anywhere.

```
datatype Exp = EI of int                          (* integer constant   *)
             | EId of string                      (* identifier         *)
             | EApp of Exp*Exp                    (* application        *)
             | EP of Exp*Exp                      (* pair               *)
             | ELam of string*Exp                 (* lambda abstraction *)
             | EIf of Exp*Exp*Exp                 (* conditional *)
             | ... other clauses elided ...

(* interpret : Exp * (string list) -> U list -> U *)
fun interpret (e,static) =
case e of
   EI n => (fn dynamic => (UI n))
 | EId s => (case indexof (static,s) of
               SOME n => fn dynamic => List.nth (dynamic,n)
             | NONE => let val lib = Builtins.lookup s
                       in fn dynamic => lib
                       end)
         (* if s not in static env, lookup in pervasives instead *)
 | EP (e1,e2) => let val s1 = interpret (e1,static)
                     val s2 = interpret (e2,static)
                 in fn dynamic => case s1 dynamic of
                                    UErr => UErr
                                  | v1 => (case s2 dynamic of
                                             UErr => UErr
                                             | v2 => UP(v1, v2))
                 end
 | EApp (e1,e2) => let val s1 = interpret (e1,static)
                       val s2 = interpret (e2,static)
                   in fn dynamic => case s1 dynamic of
                                      UF(f) => f (s2 dynamic)
                                    |  _    => UErr
                   end
 | ELam (x,e) => let val s = interpret (e, x::static)
                 in fn dynamic => UF(fn v=> case v of UErr => UErr
                                                   | _ => s (v::dynamic))
                 end
 | EIf (e1,e2,e3) => let val s1 = interpret (e1,static)
                         val s2 = interpret (e2,static)
                         val s3 = interpret (e3,static)
                     in fn dynamic => case s1 dynamic of
                                        UB(true) => s2 dynamic
                                      | UB(false) => s3 dynamic
                                      | _ => UErr
                     end
 ... other clauses elided ...

fun pi s = interpret (read s, []) []
```

Fig. 1. Revised Interpreter (extract)

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

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

The matching structure makes unsurprising definitions of embedding-projection pairs:

```
  type 'a EP = ('a->U)*(U->'a option)
  fun embed ((e,p) : 'a EP) = e
  fun project ((e,p) : 'a EP) = p
```

and the instances at base types are also straightforward:

```
  val int  = (UI, fn (UI n) => SOME n | _ => NONE)
  val string  = (US, fn (US s) => SOME s | _ => NONE)
  val unit = (fn ()=>UUnit, fn UUnit => SOME () | _ => NONE)
  val bool = (UB, fn (UB b) => SOME b | _ => NONE)
```

The embedding-projection for products is only a little more complex:

```
  infix **

  fun (e,p)**(e',p') =
   (fn (v,v') => UP(e v, e' v'),
    fn uu => case uu
             of UP (u,u') => (case (p u, p' u')
                               of (SOME v, SOME v') => SOME (v,v')
                               | _ => NONE
                             )
              | _ => NONE)
```

To embed a pair of ML values, we simply embed each component and wrap the resulting pair of untyped values in the `UP` constructor. To project a value of type `U` to a pair type, we first check that it is indeed a `UP` and then that we can project each of the components in a pointwise fashion; if so, we return `SOME` of the paired results, and otherwise we return `NONE`.

We have now reached the important and tricky part of the paper: dealing with function types. Recall that our intuition is that whenever we have an ML

value of type A in our hand, then we can assume it really will behave itself as an element of type A. In fact, we might have obtained the value by projecting some ill-behaved untyped code, but we will arrange things so that the projected value is 'self-policing' – should it ever violate the contract associated with A then it will backtrack to the point of projection. Hence the violation will never be observable at the actual point of use.

With that idea in mind, the *embedding* component for function types can be fairly straightforward. Just as we made the interpreter 'total', in the sense that dynamic errors are explicitly reified as the `UErr` value, the embedded version of `f` $:A$`->`B should be a 'total' function from `U` to `U` that attempts to project its argument to type A, returning `UErr` if this fails, and returning the embedding at type B of `f` applied to the projected value otherwise. In code, assuming `p` is the projection for A and `e'` is the embedding for B, the embedding for A`->`B is

```
fn f => UF (fn u => case p u of SOME a => e' (f a)
                              | NONE => UErr)
```

of type (A`->`B)`->U`. This is simple because nobody is at 'fault' here yet: we can assume the ML function we're embedding will be well-behaved on the given domain and we merely extend it to return `UErr` on the rest of `U`.

We now need to define *projection* for a function type A`->`B, which will be of type `U->`((A`->`B)`option`). In the case that the argument value is not even a `UF`, it seems natural to return `NONE` immediately, though we *could* have chosen to delay even this check until we try to apply the projected function. Otherwise we have a function `f` of type `U->U`, which can clearly be turned into one, call it `f'`, of type A`->(`B `option)` by precomposition with the (total) embedding for A, `e`, and postcomposing with the (partial) projection for B, `p'`. So

$$f' = \texttt{p'} \circ f \circ \texttt{e}$$

However, we want something of type (A`->`B) `option`, for which we need a control operator. We grab the continuation k that is expecting a value of type (A`->`B) `option` and provisionally return `SOME`$\,g$ where g : `A->B` is a wrapper around `f'` that returns b when `f'` returns `SOME`$\,b$ and throws `NONE` (of type (A`->`B) `option`) to the captured continuation k should f' ever return `NONE` (of type B `option`). Putting this together with the embedding component, we arrive at the definition of the `-->` combinator that is shown in Figure 2.

The code shown in the figure looks rather simple, but the consequences of the way the uses of control are intertwined with the induction on types are perhaps not obvious, so we now present a series of examples to try to understand what we just did.

4 Examples

Our first set of examples are not intended to be representative of actual uses, but merely a set of test cases to demonstrate and check the behaviour of our earlier definitions. Since we want to be able to see *which* coercions fail, we make our test functions return values of the following type:

```
(* Recall:
  type 'a EP = ('a->U)*(U->'a option)
  val --> : ('a EP)*('b EP) -> ('a->'b) EP
*)

infixr -->

fun (e,p)-->(e',p') =
 (fn f => UF (fn u => case p u of SOME a => e' (f a)
                                | NONE => UErr),
  fn u => case u
          of UF f => callcc (fn k =>
                             SOME (fn a =>
                                     case p' (f (e a))
                                     of SOME b => b
                                      | NONE => throw k NONE))
           | _ => NONE)
```

Fig. 2. Embedding and projection for functions

```
datatype 'a TestResult = Fail of string | Result of 'a
```

We start with an untyped function that behaves like successor on small integers but returns a unit value on larger ones:

```
- val badsucc = pi "fn n => if n < 4 then n+1 else ()";
val badsucc = UF fn : U
```

and define a test function that attempts to project an untyped value down to the ML type `int -> int` and then maps the result over a list of integers:

```
- fun testIntToInt v l = case (project (int-->int) v)
                         of SOME f => Result (map f l)
                          | NONE => Fail "projection to int->int";
val testIntToInt = fn : U -> int list -> int list TestResult
```

Now, we try our test out on a list of small integers:

```
- val test1 = testIntToInt badsucc [1,2,3];
val test1 = Result [2,3,4] : int list TestResult
```

All the integers in the test list are small and `badsucc` behaves like a function of type `int -> int` on all of them, so we don't see any violation. Let's extend the list a little:

```
- val test2 = testIntToInt badsucc [1,2,3,4,5,6,7];
val test2 = Fail "projection to int->int" : int list TestResult
```

This time, `badsucc` behaves itself for the first three calls then violates its contract on the fourth, at which point we backtrack to the original point of projection and make that fail retrospectively. Alternatively, of course, we can project at `int -> unit`, in which case the sets of arguments exhibiting success and failure are swapped:

```
- fun testIntToUnit v l = case (project (int --> unit) v)
                          of SOME f => Result (map f l)
                           | NONE => Fail "projection to int->unit";
val testIntToUnit = fn : U -> int list -> unit list TestResult

- val test3 = testIntToUnit badsucc [7,6,5,4];
val test3 = Result [(),(),(),()] : unit list TestResult

- val test4 = testIntToUnit badsucc [7,6,5,4,3,2,1];
val test4 = Fail "projection to int->unit" : unit list TestResult
```

Note that each call to `project`, even on the same value, yields a new point to which we can backtrack. Let's check that we've stacked things up in the right order to maintain the property that embedding followed by projection is the identity, even on ill-behaved values:

```
- fun testreembed a1 a2 =
   case project (int --> int) badsucc
   of NONE => Fail "first projection"
    | SOME first =>
        let val r1 = first a1
            val reembed = embed (int --> int) first
        in case project (int --> int) reembed
           of NONE => Fail "second projection"
            | SOME second => let val r2 = second a2
                             in Result (r1,r2)
                             end
        end;
val testreembed = fn : int -> int -> (int * int) TestResult

- val test5 = testreembed 2 3;
val test5 = Result (3,4) : (int * int) TestResult

- val test6 = testreembed 2 4;
val test6 = Fail "first projection" : (int * int) TestResult
```

Although the violation is only triggered by the application of the reprojected value `second` to 4, we have correctly unwound all the way to the initial projection.

Now let's try some higher-order examples:

```
- fun testho A x y =
    case project (A-->int) x
```

```
    of NONE => Fail "function projection"
     | SOME f => (case project A y
                   of NONE => Fail "argument projection"
                    | SOME g => Result (f g));
val testho = fn : 'a EP -> U -> U -> int TestResult

- val test7 = testho (int-->int) (pi "fn f => f 1 + f 2")
                                 (pi "fn n => n + 1");
val test7 = Result 5 : int TestResult

- val test8 = testho (int-->int) (pi "fn f => f 1 + f 2")
                            (pi "fn n => if n = 1 then 7 else ()");
val test8 = Fail "argument projection" : int TestResult

- val test9 = testho (int-->int) (pi "fn f => f 1 + f true")
                                 (pi "fn n => n + 1");
val test9 = Fail "function projection" : int TestResult
```

In these tests, we try to project the first untyped program to `(int->int)->int`, the second to `int->int` and then apply one to the other. We can see that in `test7`, both are well-behaved, in `test8` it's the argument that goes wrong whilst in `test9` the higher order function is at fault.

What should happen if we try to combine more than one faulty value? We are doing dynamic checking of the contracts: projections are only rolled back in the case that the particular use that is made of them exposes a violation. The checking is eager, in that the context $C[\cdot]$ that tests a projected value v should not have previously violated *its* contract at the point when the v does something wrong; otherwise we should already have rolled back some other projected value in $C[\cdot]$. So when we combine more than one projected value, we roll back the first one which detectably goes wrong in the execution trace:

```
- val test10 = testho (int-->int) (pi "fn f => f 1 + f true")
                                  (pi "fn b => if b then 3 else 4");
val test10 = Fail "argument projection" : int TestResult

- val test11 = testho (int -->int) (pi "fn f => f true + f 1")
                                   (pi "fn b => if b then 3 else 4");
val test11 = Fail "function projection" : int TestResult
```

In `test10`, the error is signalled in the argument. This is because the first call made by the higher-order function passes an integer, as per the contract, and then the argument tries to use that in a conditional, violating its contract. In `test11`, we have swapped the order of evaluation in the addition so that the first dynamically occurring violation is the attempt by the higher order function to pass `true` in place of an integer; in this case the error is detected in the higher-order function. Here's a more complex example:

```
- val test12 = testho ((int-->int)-->(int-->int))
                (pi "fn f => f (f (fn n => n+1)) 5")
                (pi "fn g => if g 2 = 3 then fn n => true
                                        else fn n => n");
val test12 = Fail "argument projection" : int TestResult
```

In this case, the outer call to the function bound to `f` passes in a function that is not of type `int->int`, but that function was itself obtained by the inner call to `f`, which was with a well-behaved argument. Hence blame is correctly assigned to the second of the original terms.

Finally, we present a toy version of a marginally more realistic example. Consider making queries on an external database, modelled as a function that takes a query predicate on strings (of type `string -> bool`) and returns a function of type `int -> string` that enumerates the results. Here are some definitions in the untyped language that construct three different purported databases, using LISP-style lists internally (represented as nested pairs with the unit value for `nil`):

```
- fun mkdb ds = pi ("let fun query l f n =
                       if isnil l then \"\"
                       else if f (car l)
                            then if n=0 then (car l)
                                 else query (cdr l) f (n-1)
                            else query (cdr l) f n
                     in query " ^ ds)
val mkdb = fn : string -> U

- val db1 = mkdb "(\"um\",(\"dois\",(\"tres\",(true,()))))"
- val db2 = mkdb "(\"un\",(2,(\"trois\",(\"quatre\",()))))"
- val db3 = mkdb "(\"one\",(\"two\",(\"three\",(\"four\",()))))"
```

Note that the first two contain some non-string values. The following function takes a list of untyped values and returns the first one that projects correctly to our ML type for databases:

```
- fun selectdb [] = (fn f => fn n => "")
    | selectdb (x::xs) =
        case project ((string --> bool) --> (int --> string)) x
        of NONE => selectdb xs
         | SOME db => db
val selectdb = fn : U list -> (string -> bool) -> int -> string
```

Now we can try some queries:

```
- val test15 = let val thedb = selectdb [db1,db2,db3]
                   val results = thedb (fn s => String.size s > 3)
               in [results 0, results 1]
               end
```

```
val test15 = ["dois","tres"] : string list

- val test16 = let val thedb = selectdb [db1,db2,db3]
                   val results = thedb (fn s => String.size s > 3)
               in [results 0, results 1, results 2]
               end
val test16 = ["three","four",""] : string list
```

The first database in the list produces two results without violating the contract, so we get the answers from that database. When we ask for more results, however, the first database tries to apply the filter to a boolean, so gets rolled back; we then try the second database, which also fails because it contains an integer, and finally end up getting all our results from the third one.

5 Discussion

We have shown how continuation-based backtracking combines smoothly with type indexed embedding-projection pairs to yield a convenient form of dynamic contract checking for interoperability between typed and untyped higher-order languages, localizing runtime errors to a single point of failure.

Extensions of statically-typed languages with various forms of dynamic type have been well-studied (see, for example, [1, 10]), but undoable projections have not, as far as I'm aware, been proposed before.

The use of embedding-projection pairs to define type-indexed functions in ML-like languages is attributed by Danvy [6] to Filinski and to Yang [20], both of whom used it to implement type-directed partial evaluation [4], which involves type-indexed functions that appear at first sight to call for dependent types. Rose [16] describes an implementation of TDPE in Haskell that uses type classes to pass the pairs representing types implicitly. Kennedy and I have previously used it for writing picklers [12] and interpreters [2], respectively. Similar type-directed constructions have also been used in implementing `printf`-like string formatting [5] and in generic programming. Ramsey has also applied the technique for embedding an external interpreter for a scripting language (Lua) into OCaml programs [15].

Control operators have, of course, been used to implement various other forms of backtracking before, including that of logic programming languages. Nevertheless, getting the apparently simple code here correct is not entirely trivial (my first couple of attempts were more complex and subtly wrong).

It remains to be seen whether or not the technique presented here is actually useful in practical situations. Even before one worries about the specific technicalities, many reasonable people believe that experience with RPC, distributed objects, persistent programming, and so on, all indicates that trying to hide the differences between operations with widely varying runtime costs, failure models or lifetimes is fundamentally a bad idea – the distinctions should be reflected in the language because programmers need to be aware of them. Holding onto continuations costs space, whilst the possibility of backtracking over expensive

computations certainly doesn't make reasoning about time or space behaviour any easier.

There is also the issue of what can be undone. We have been implicitly assuming that the untyped programs that we project, and the typed contexts into which we project them, do not themselves involve side-effects other than potential divergence, as these will not be undone by throwing to the captured continuation. One could certainly extend our technique to effects that are internal to the language, such as uses of state or other uses of control, if one were prepared to modify the compiler, runtime system or bits of the basis library (as in previous work on transactions in ML [9]). But the most exciting examples, namely those that involve external I/O, unfortunately concern side-effects that are rather hard to roll back automatically and generically. If I've sent you some messages and then you start to misbehave, the best general thing I can do is break off further communication with you; I certainly can't unsend the messages. That suffices in the case of pure computations, but in the stateful case a general solution seems to require at least wrapping the underlying messages in a more complex fault-tolerant protocol, and probably introducing explicit transactional commitment points, beyond which rollbacks would no longer be possible. Indeed, explicitly delimiting the the extent of possible rollbacks may be advantageous even in the case of purely internal effects.

The semantics we have chosen to implement here tracks type errors rather strictly along the control flow: any violation will cause the guilty projection to be undone, even if the value that is eventually produced is well-behaved. A small change in the interpreter code to remove strictness in `UErr` yields a laxer, more data-dependent, semantics, in which 'benign' errors are ignored; this might be useful in some circumstances, but seems harder to reason about and a less natural fit with call-by-value languages.

It would be good to formulate and prove correctness of the code we have presented. The problem here seems not to be one of proof technique, but in coming up with a statement of correctness that covers the correct assignment of blame in the case of multiple ill-behaved untyped programs *and* which is intuitively significantly clearer than the code itself. One starting point might be the operational semantics for interoperability between ML-like and Scheme-like languages recently described by Matthews and Findler [13].

A second, and perhaps more interesting, line of further work is to extend the idea from dynamic checking of the interface between typed and untyped languages to recovery in more general higher-order contract monitoring. Runtime checking of contracts, and the associated issue of blame assignment, have been extensively studied in recent years (see, for example, [8, 7, 11, 3, 19]) and the kind of 'transactional' recovery mechanism described here seems eminently applicable in that setting.

Thanks to Josh Berdine, Olivier Danvy, Andrzej Filinski, Norman Ramsey and the referees for many useful comments on earlier drafts of this paper.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2), 1991.
2. N. Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4), 2005.
3. M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4/5), 2006.
4. O. Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1996.
5. O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6), 1998.
6. O. Danvy. A simple solution to type specialization (extended abstract). In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
7. R. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
8. R. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2002.
9. N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6), 1994.
10. F. Henglein. Dynamic typing. In *Proceedings of the 4th European Symposium on Programming (ESOP)*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
11. J. Jeuring, R. Hinze, and A. Loeh. Typed contracts for functional programming. In *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
12. A. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6), 2004.
13. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)*, 2007.
14. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
15. N. Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 2008. To appear.
16. K. Rose. Type-directed partial evaluation in Haskell. In *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation*, number NS-98-1 in BRICS Notes, 1998.
17. D. Scott. Data types as lattices. *SIAM Journal of Computing*, 4, 1976.
18. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
19. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ACM Workshop on Scheme and Functional Programming*, 2007.
20. Z. Yang. Encoding types in ML-like languages. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 1998.