Many Holes in Hindley-Milner

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

Abstract

We implement statically-typed multi-holed contexts in OCaml using an underlying algebraic datatype augmented with phantom types. Existing approaches require dynamic checks or more complex type systems. In order to support concatenation we use two type parameters to represent the number of holes in a context as the difference between two Peano numbers. In order to support plugging a context with contexts of different arity we introduce a datatype of lists of contexts of length n with a total of m holes. Further, we extend our representation to allow holes to be marked with additional type information. As an example, we use these marks to implement statically-typed multi-holed XHTML contexts. We take advantage of Garrigue's relaxed value restriction.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (functional) programming

General Terms Design, Languages

Keywords multi-holed context, phantom type, dependent type, indexed type, value restriction

1. Introduction

It is well-known how to define a statically typed encoding of the type of one-hole contexts of an algebraic datatype in a ML-style Hindley-Milner type system (Huet 1997; McBride 2001).

A more challenging problem is to define a datatype of multi-holed contexts. We require that the datatype supports operations for constructing multi-holed contexts, including an operation to concatenate two multi-holed contexts, as well as an operation for plugging all the holes of a multi-holed contexts with other multi-holed contexts.

Two obvious encodings come to mind: an algebraic datatype consisting of the raw algebraic datatype augmented with a constructor for holes, or a curried function where each argument represents a hole. The first encoding is deficient because plugging requires a dynamic check. The second encoding is deficient because it is too general in that it fails to capture the property that each hole occurs exactly once in a context, and it is too restrictive in that each hole can only be plugged with multi-holed contexts containing a fixed number of holes.

It is well-known how to solve this class of problem by adding features to the type system such as: type classes (McBride 2002), indexed types (Zenger 1997; Xi and Pfenning 1999), GADTs (Cheney and Hinze 2003; Xi et al. 2003; Jones et al. 2006), or full-on dependent types (Altenkirch et al. 2005; Fogarty et al. 2007). This paper gives an implementation of statically-typed multi-holed contexts in a standard Hindley-Milner type system. The only extension we rely on is abstract types. We also take advantage of Garrigue's relaxed value restriction (Garrigue 2004).

It is not entirely obvious how one might implement a concatenation operation on multi-holed contexts because it requires an encoding of type level addition (the type of `concat` should capture the property that the number of holes in the output context is the sum of the number of holes in each of the input contexts). The same difficulty arises in the slightly simpler task of defining an append function on lists of length n.

Folklore holds that it is not possible in ML to give the append function on lists a type that captures the property that the length of the output list is the sum of the lengths of the input lists. For instance, Xi writes (Xi 2007):

> A correct implementation of the append function on lists should return a list of length $m + n$ when given two lists of length m and n, respectively. This property, however, cannot be captured by the type system of ML, and the inadequacy can be remedied if we introduce a restricted form of dependent types.

We show how to capture this property in the type system of ML using phantom types. Our main innovation is to encode naturals at the type level as pairs of Peano numbers $\langle m, n\rangle$ representing the difference between n and m. This allows us to implement addition as composition: $(m - l) + (n - m) = (n - l)$. Once we have shown how to implement the append function, we apply and extend the technique to implement statically-typed multi-holed contexts with concatenation and plugging. We then demonstrate how to combine these multi-holed contexts with additional static type information, using Elsman and Larsen's MiniXHTML fragment of XHTML (Elsman and Larsen 2004) as an example.

2. Multi-holed contexts

The ideas of this paper are applicable to multi-holed contexts over any regular algebraic datatype. As a running example we use an algebraic datatype for representing XML contexts. The underlying datatype represents multi-holed XML contexts where the number of holes does not appear in the type.

```
type xml =
  | Empty
  | Text of string
  | Tag of string * xml
  | Concat of xml * xml
  | Hole
```

The constructors are interpreted as follows: `Empty` constructs an empty XML context, `Text s` constructs a text node, `Tag (name,`

x) wraps a tag whose name is name around the XML context x, Concat (x, y) concatenates the XML context x with the XML context y and Hole constructs a hole. To simplify the presentation we ignore attributes. Note that the first four constructors are sufficient for constructing XML. The Hole constructor allows us to promote the XML datatype to an XML context datatype. In general we can convert any regular algebraic datatype to a datatype of contexts over the original datatype by adding an extra Hole constructor.

The constructors can be used to build up an arbitrary XML context. For XML contexts to be useful we also need a means for deconstructing them. We define an operation to plug the holes of a *primary context* with a list of *sub-contexts*.

```
(* dynamic_plug : xml * xml list -> xml *)
let dynamic_plug (k, xs) =
  let rec plug (k, xs) =
    match k with
      | Empty -> Empty, xs
      | Text s -> Text s, xs
      | Tag (s, k) ->
          let (k, xs) = plug (k, xs) in
            Tag (s, k), xs
      | Concat (k, k') ->
          let (k, xs) = plug (k, xs) in
          let (k', xs) = plug (k', xs) in
            Concat (k, k'), xs
      | Hole ->
          begin match xs with
            | [] ->
              failwith "ran out of xml to plug in"
            | x::xs -> x, xs
          end in
  let k, xs = plug (k, xs) in
    if (xs <> []) then
      failwith "failed to plug in all the xml"
    else
      k
```

The dynamic_plug operation is defined in terms of an auxiliary plug function that recursively plugs the holes of the primary context with the sub-contexts, returning a pair of the plugged primary context (the *output context*) and any remaining sub-contexts. It is *dynamic* in the sense that it checks for failure at run-time. Plugging can fail in two places corresponding to too few or too many elements in the list of sub-contexts.

It is not difficult to verify that dynamic_plug(k, xs) will fail iff the number of holes in k differs from the length of xs, that is, dynamic_plug(k, xs) fails iff holes k $\neq$ length xs where length and holes are defined as follows:

```
let rec holes =
  function
    | Empty -> 0
    | Text s -> 0
    | Tag (_, k) -> holes k
    | Concat (k, k') -> holes k + holes k'
    | Hole -> 1

let rec length =
  function
    | [] -> 0
    | _::xs -> 1 + length xs
```

In the rest of this paper we will show how to define multi-holed contexts in such a way that plugging cannot fail at run-time. We do this by defining an XML context datatype that is annotated with its number of holes, a list of XML contexts datatype that is annotated with its length and the total number of holes in the list, and a plug function that takes an annotated primary context and an annotated list of sub-contexts and returns an annotated output context. The type of the plug function captures the property that the number of holes in the primary context matches the length of the list of sub-contexts, and furthermore that the number of holes in the output context is the same as the total number of holes in the list of sub-contexts.

3. Difference types

One of the most basic tools we need for counting statically is a type-level encoding of naturals. Type-level Peano numbers are easily encoded in OCaml.

```
type z
type 'a s
```

The type z represents zero and given any type-level natural n, the type n s represents the successor of n. (The syntax of OCaml forces the Peano numbers to appear backwards, for instance, (z s) s instead of s (s z)), but this is no great burden.) Note that these definitions define uninhabited types, which is what we want as their sole purpose is static checking. Also note that the type variable can be instantiated at types that do not encode naturals, for instance, there is nothing to stop us using the type int s. However, the use of abstract types at least ensures that such “nonsense types” can only be introduced through explicit type annotations, and we ensure that such annotations do not allow programmers to do anything unsafe.

Now we have a type-level encoding of naturals, it is not difficult to implement basic operations for constructing lists of length n.

```
module SimpleNList :
sig
  type ('length, 'elem_type) t

  val nil : (z, 'a) t
  val cons : 'a * ('n, 'a) t -> ('n s, 'a) t
end
  =
struct
  type ('n, 'a) t = 'a list

  let nil = []
  let cons (x, xs) = x :: xs
end
```

The first parameter of SimpleNList.t is a phantom type parameter that encodes the length of a list. The actual implementation of the SimpleNList operations simply calls the corresponding operation on standard lists. All of the interesting part of this code is in the types. The types encode the number of elements in the list. For instance, the following:

```
# open SimpleNList;;
# cons (1, (cons (2, nil)));;
- : (z s s, int) SimpleNList.t = <abstr>
```

produces a list of integers of length two (Peano number z s s).

We could use the same idea to define statically-typed multi-holed XML contexts. This works for all the constructors except Concat. The problem is that if we concatenate two contexts then we need to add the number of holes together, and our type-level encoding of Peano numbers cannot support addition. We need an alternative encoding that does support addition.

The key idea is to represent the number of holes as the difference between two Peano numbers rather than just a single number.

Let concat be the concatenation operator. Suppose the number of holes x is represented by the difference $n - m$ and the number of holes of y is represented by the difference $m - l$, then the number of holes in concat (x, y) is $(n - m) + (m - l) = (n - l)$: addition of differences does not require any addition at all!

Of course, we actually want to leverage the Hindley-Milner type system to perform addition for us, and we want to be able to write functions that are polymorphic in our encoding of naturals. Concretely, we encode a natural number as a pair of type parameters 'm*'n. The intention is that 'm and 'n be only instantiated as type-level Peano numbers, and that the pair 'm*'n represents the difference 'n − 'm.

In fact, the operations we define will ensure that the types exposed to users are always in a more restricted form. The first parameter is always a type variable 'm, and the second one is always of the form 'm $\texttt{s}^i$, writing $\texttt{s}^i$ for a sequence of s constructors of length i. We refer to types of this form as *difference types*. Note that difference types make use of the successor constructor s, but do not require the zero constructor z (zero is encoded as 'm*'m). A difference type encodes a type function that given any 'm returns 'm $+ i$. To perform addition we compose two such type functions together: given 'n*'n $\texttt{s}^i$ and 'm*'m $\texttt{s}^j$ we simply unify 'm $\texttt{s}^i$ with 'n to obtain 'm*'m $\texttt{s}^{i+j}$.

Before implementing statically-typed XML contexts we illustrate difference types by adapting our simple implementation of lists of length n to use difference types, and augmenting it with an append operation.

```
module NList :
sig
  type (+'length, +'elem_type) t

  val nil : ('m*'m, 'a) t
  val cons : 'a * ('m*'n, 'a) t ->
             ('m*'n s, 'a) t
  val append : ('m*'n, 'a) t * ('l*'m, 'a) t ->
               ('l*'n, 'a) t

  val to_list : ('i, 'a) t -> 'a list
end
  =
struct
  type ('i, 'a) t = 'a list

  let nil = []
  let cons (x, xs) = x :: xs
  let append (xs, ys) = xs @ ys

  let to_list xs = xs
end
```

As with SimpleNList.t the first parameter of NList.t is a phantom type parameter that encodes the length of the list. The types of nil and cons are adjusted accordingly and the type of the append operation adds the list lengths of its two inputs together by composing the difference types.

3.1 The relaxed value restriction

The variance annotations (+) on the type variables of NList.t indicate that the type variables are only used in covariant positions, enabling Garrigue's relaxed value restriction (Garrigue 2004). (Of course, the phantom type variable does not occur at all in the type, so we could equally well give it a contravariant annotation if we wanted.) With the variance annotations any list we construct will always be as polymorphic as possible. For instance:

```
# NList.cons (1, NList.nil);;
- : ('a * 'a s, int) NList.t = <abstr>
```

Normally the value restriction (Wright 1995) would prevent this term from being generalised, and hence it would not be polymorphic. However, Garrigue's relaxed value restriction (Garrigue 2004) allows it to be generalised. The relaxed value restriction allows any free type variables which only occur in covariant positions outside of reference types to be generalised even for terms which are not syntactic values. Without the variance annotations, OCaml would not be able to determine that the difference type parameter to NList.t only occurs covariantly and we would get:

```
- : ('_a * '_a s, int) NList.t = <abstr>
```

(The weak type variable '_a, once instantiated, must always be instantiated to the same type in the future.) Of course, we still do not get all the polymorphism we might hope for. For instance:

```
# let curry f = fun x y -> f(x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b ->
            'c = <fun>
# (curry NList.cons) 1;;
- : ('_a * '_b, int) NList.t ->
    ('_a * '_b s, int) NList.t = <fun>
```

The problem here is that the type variables occur in contravariant positions, so the relaxed value restriction does not apply. We have deliberately chosen to give uncurried types to constructors in order to make it harder to accidentally lose polymorphism. Manual currying does give us a value and hence allows type inference to generalise:

```
# fun xs -> NList.cons (1, xs);;
- : ('a * 'b, int) NList.t ->
    ('a * 'b s, int) NList.t = <fun>
```

The to_list operation performs a safe cast from the type ('i,'a) NList.t to the type 'a list (a list of length n can always be safely treated as a list of arbitrary length). Casting allows us to spread our implementation across several modules.

4. Counting holes

The implementation of statically-typed multi-holed contexts uses the same ideas as NList.

```
module NContext
  :
sig
  (* context*)
  type +'holes t

  (* context constructors *)
  val empty : ('m*'m) t
  val text : string -> ('m*'m) t
  val tag : string * 'i t -> 'i t
  val concat : ('m*'n) t * ('l*'m) t -> ('l*'n) t
  val hole : ('n*'n s) t

  (* upcast to xml *)
  val to_xml : 'i t -> xml
end
  =
struct
  type 'i t = xml

  let empty = Empty
  let text s = Text s
  let tag (s, x) = Tag (s, x)
  let concat (x, y) = Concat (x, y)
  let hole = Hole

  let to_xml k = k
end
```

The operations `empty`, `text`, `tag`, `concat` and `hole` simply invoke the corresponding constructors of the xml type. As with `append`, the `concat` operation adds the number of holes together by composing difference encodings.

Example 1

```
  open NContext;;
  # let k =
      concat
       (tag ("p", hole),
        tag ("table",
          concat
           (tag ("tr", hole),
            tag ("tr", hole))));;
  val k : ('a * 'a s s s) NContext.t = <abstr>
  # to_xml k;;
  - : xml =
  Concat
    (Tag ("p", Hole),
     Tag ("table",
       Concat
        (Tag ("tr", Hole),
         Tag ("tr", Hole))))
```

The context `k` has three holes, and is hence assigned the type `('a * 'a s s s) NContext.t`.

Now we can statically type the construction of multi-holed contexts, but we would also like a means for statically typing the destruction of multi-holed contexts: a statically-typed plugging operation. At first glance, it may seem unlikely that we would be able to define a statically typed plugging operation. In the general case we want to be able to plug an n-holed context with a heterogeneous list of length n of multi-holed contexts $[C_1, \ldots, C_n]$ to give an $(m_1 + \cdots + m_n)$-holed context where each C_i is an m_i-holed context. But how can we implement a heterogeneous list?

The key observation is that the implementation does not actually need to use a heterogeneous list. The plugging operation takes an n-holed context and a list of multi-holed contexts, but it is not necessary to track the number of holes in each of the individual contexts; we just need to know the sum of the total number of holes in the list of contexts. Thus we define a datatype of lists of multi-holed contexts of length n with m holes, along with an associated plugging operation.

```
module NContext
  :
sig
  ...

  (* context list *)
  type (+'holes, +'length) ts

  (* context list constructors *)
  val nil : ('p*'p, 'm*'m) ts
  val cons :
    ('p*'q) t * ('o*'p, 'm*'n) ts ->
    ('o*'q, 'm*'n s) ts
  val append :
    ('p*'q, 'm*'n) ts * ('o*'p, 'l*'m) ts ->
    ('o*'q, 'l*'n) ts

  (* plugging *)
  val plug : 'j t * ('i, 'j) ts -> 'i t
end
  =
struct
  ...

  type ('i, 'j) ts = ('j, xml) NList.t

  let nil = NList.nil
  let cons (x, xs) = NList.cons (to_xml x, xs)
  let append = NList.append

  let plug (k, xs) =
    dynamic_plug (k, NList.to_list xs)
end
```

The underlying implementation is a homogeneous list of length n of unannotated XML contexts. The list constructor operations forward to the corresponding operations on lists of length n. The casts to unannotated contexts allow us to get away with using a homogeneous list in the implementation.

Example 2

```
  # let xs =
      cons (tag ("em", text "plugging"),
      cons (tag ("td", hole),
      cons (tag ("td", text "holes"), nil)));;
  val xs :
   ('a * 'a s, 'b * 'b s s s) NContext.ts =
     <abstr>
  # plug (k, xs);;
  - : ('a * 'a s) NContext.t = <abstr>
  # to_xml (plug (k, xs));;
  - : xml =
  Concat (Tag ("p", Tag ("em", Text "plugging")),
   Tag ("table",
    Concat (Tag ("tr", Tag ("td", Hole)),
     Tag ("tr", Tag ("td", Text "holes")))))
```

The contexts in the list `xs` have a total of one hole and there are three contexts in the list, hence it is assigned the type `('a * 'a s, 'b * 'b s s s) NContext.ts`. As `k` has three holes, it can be plugged with the elements of `xs`, yielding a one-holed context of type `('a * 'a s) NContext.t`.

Example 3

```
  # let ys =
      cons (tag ("td", hole),
      cons (tag ("td", text "holes"), nil));;
  val ys :
    ('a * 'a s, 'b * 'b s s) NContext.ts =
      <abstr>
  # plug (k, ys);;
  Characters 5-12:
    plug (k, ys);;
         ^^^^^^^
  This expression has type
    ('a * 'a s s s) NContext.t *
      ('b * 'b s, 'a * 'a s s) NContext.ts
  but is here used with type
    ('a * 'a s s s) NContext.t *
      ('b * 'b s, 'a * 'a s s s) NContext.ts
```

The contexts in the list ys have a total of one hole and there are two contexts in the list, hence it is assigned the type `('a * 'a s, 'b`

`* 'b s s) NContext.ts`. Attempting to plug `k` with `ys` fails as there are only two elements in `ys` but `k` has three holes.

An important criticism of the style of “type-hackery” that we are engaging in is that it can lead to hard to understand error messages. The above error message simply says that the plugging operation has been supplied with a pair of a context with three holes and a list of two elements, but expects a pair of a context with three holes and a list of three elements. It would certainly be nicer if the difference types could be rendered using Arabic numerals, but apart from that it seems quite readable, at least to the author. Admittedly, this view becomes rather less tenable as the number of holes gets bigger — the unary Peano representation of naturals is exponentially longer than the denary Arabic representation. One might imagine post-processing error messages to turn difference types into numbers.

5. Marking holes

Using a number of tricks we have managed to implement statically-typed multi-holed contexts in OCaml. The type system keeps track of the number of holes in contexts and statically ensures that we cannot plug the wrong number of sub-contexts into a primary context. The solution presented thus far is somewhat restrictive, though, in that it does not allow further type information to be attached to contexts or holes. For instance, we might want to statically ensure that our XML matches some XML schema. This would require a way of attaching additional type information to both contexts and holes.

In this section, we demonstrate how to add this extra type information. A number of different means for statically enforcing XML validity appear in the literature (Brabrand et al. 2001; Thiemann 2002; Hosoya and Pierce 2003; Elsman and Larsen 2004; Møller and Schwartzbach 2005). As a proof of concept, we illustrate how to combine our statically-typed multi-holed contexts with Elsman and Larsen’s MiniXHTML (Elsman and Larsen 2004). We believe it should be possible to integrate other XML typing schemes with our multi-holed XML contexts as the two features appear to be orthogonal. Elsman and Larsen’s is a natural fit for our setting as it uses phantom types to classify the different kinds of XHTML tags. MiniXHTML is a tiny fragment of XHTML which only includes the tags `p`, `em`, `pre`, `big`, `table`, `tr` and `td`.

The DTD for MiniXHTML is:

```
<!ENTITY %block "p|table|pre">
<!ENTITY %inline "%inpre|big">
<!ENTITY %flow "%block|%inline">
<!ENTITY %inpre "#PCDATA|em">
<!ENTITY %td "td">
<!ENTITY %tr "tr">

<!ELEMENT p (%inline)*>
<!ELEMENT em (%inline)*>
<!ELEMENT big (%inline)*>
<!ELEMENT pre (%inpre)*>
<!ELEMENT td (%flow)*>
<!ELEMENT tr (%td)+>
<!ELEMENT table (%tr)+>
```

The adaptation of contexts to accomodate extra type information is relatively straightforward. As well as the phantom type parameter `+'i` for the number of holes, the type `t` is also given a further type parameter `+'h`, a *mark* which encodes validity constraints on the XML. Furthermore, a mark is also added to each hole in the successor constructor, encoding validity constraints on the XML that is allowed to be plugged in the hole. In effect, we are moving from a difference encoding of naturals to a difference encoding of type lists. The type constructor `s` can now be read as *cons*.

```
module MX
  :
sig
  (* entities *)
  type (+'blk, +'inl) flw and tr and td
  type blk and inl and no and inpre
  type preclosed

  (* contexts *)
  type (+'holes, +'mark) t

  (* context constructors *)
  val empty : ('m*'m, 'h) t
  val text : string -> ('m*'m, 'h) t
  val p : ('i, (no,inl)flw*'c) t ->
          ('i, (blk,'b)flw*'c) t
  val em : ('i, (no,inl)flw*'c) t ->
           ('i, ('b,inl)flw*'c) t
  val pre : ('i, (no,inl)flw*inpre) t ->
            ('i, (blk,'b)flw*'c) t
  val big : ('i, (no,inl)flw*'c) t ->
            ('i, ('b,inl)flw*preclosed) t
  val table : ('i, tr*'c) t ->
              ('i, (blk,'b)flw*'c) t
  val tr : ('i, td*'c) t ->
           ('i, tr*'c) t
  val td : ('i, (blk,inl)flw*'c) t ->
           ('i, td*'c) t
  val concat : ('m*'n, 'h) t * ('l*'m, 'h) t ->
               ('l*'n, 'h) t
  val hole : ('m*('m*'h) s, 'h) t

  (* cast a context to xml *)
  val to_xml : ('i, 'h) t -> xml

  (* context list *)
  type (+'holes, +'length) ts

  (* context list constructors *)
  val nil : ('m*'m, 'n*'n) ts
  val cons :
    ('p*'q, 'h) t * ('o*'p, 'm*'n) ts ->
    ('o*'q, 'm*(('n*'h) s)) ts
  val append :
    ('p*'q, 'm*'n) ts * ('o*'p, 'l*'m) ts ->
    ('o*'q, 'l*'n) ts

  val plug :
    ('j, 'h) t * ('i, 'j) ts ->
    ('i, 'h) t
end
  =
struct
  type (+'blk, +'inl) flw and tr and td
  type blk and inl and no and inpre
  type preclosed

  type ('i, 'h) t = xml

  let empty = Empty
  let text s = Text s
  let p x = Tag ("p", x)
  let em x = Tag ("em", x)
  let pre x = Tag ("pre", x)
  let big x = Tag ("big", x)
  let table x = Tag ("table", x)
  let tr x = Tag ("tr", x)
  let td x = Tag ("td", x)
  let concat (x, y) = Concat (x, y)
  let hole = Hole

  let to_xml k = k

  type ('i, 'j) ts = xml list

  let nil = []
  let append (xs, ys) = xs @ ys
  let cons (x, xs) = (to_xml x) :: xs

  let plug (k, xs) = dynamic_plug (k, xs)
end
```

The marks have two components. As explained in (Elsman and Larsen 2004) the first component is used for specifying entity types and the second one is used for implementing the element prohibition of XHTML 1.0 that disallows `big` elements from appearing anywhere inside `pre` elements. The names of the phantom types used in marks are as in (Elsman and Larsen 2004) except for `no` which they call `NOT` (unlike SML, OCaml requires type names to begin with a lowercase letter).

Note that the `tag` constructor has been replaced by specific constructors for each tag that introduce the validity constraints. The `empty` and `text` constructors leave the XML context unconstrained. The `hole` constructor ensures that the annotation on the hole is the same as the annotation on the context consisting of the hole.

Example 4 The context `p hole` which represents a paragraph element with a hole in it is given the type:

```
('a * ('a * ((MX.no, MX.inl) MX.flw * 'b)) s,
 (MX.blk, 'c) MX.flw * 'b)
```

This type indicates that the context has one hole which can contain `inline` entities and the context itself is a `flow` entity that contains `block` entities.

Example 5 The extension of `NContext.ts` to lists of MiniX-HTML simply threads the marks through. The singleton context list `cons (p hole, nil)` is given the type:

```
('a * ('a * ((MX.no, MX.inl) MX.flw * 'b)) s,
 'c * ('c * ((MX.blk, 'd) MX.flw * 'b)) s)
MX.ts
```

This type indicates that the list of contexts has one hole which can contain `inline` entities and contains one context which is a `flow` entity that contains `block` entities.

Example 6

```
# open MX;;
# let k =
    concat
     (p hole,
      table (concat (tr hole, tr hole)));;
val k :
  ('a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
                  ((MX.no, MX.inl) MX.flw * 'b))
   s, (MX.blk, 'c) MX.flw * 'b)
  MX.t = <abstr>
# to_xml k;;
- : xml =
Concat
 (Tag ("p", Hole),
  Tag ("table",
    Concat
      (Tag ("tr", Hole),
       Tag ("tr", Hole))))
```

The type of `k` is the same as in Example 1, but now each hole is annotated with extra typing information for constraining what entities are allowed to be plugged into it, and the context itself is similarly annotated with extra typing information constraining what of entity it can be. In this case the first hole must be plugged with an `inline` entity, and the other two holes with `td` entities. The context itself is a flow entity that contains `block` entitites.

Example 7

```
# let xs =
    cons (em (text "plugging"),
    cons (td hole,
    cons (td (text "holes"), nil)));;
val xs :
  ('a * ('a * ((MX.blk, MX.inl) MX.flw * 'b)) s,
   'c *
   ((('c * (MX.td * 'd)) s * (MX.td * 'b)) s *
               (('e, MX.inl) MX.flw * 'f)) s)
  MX.ts = <abstr>
# plug (k, xs);;
- : ('a * ('a * ((MX.blk, MX.inl)
                      MX.flw * 'b)) s,
   (MX.blk, 'c) MX.flw * 'b)
  MX.t
```

If the number of holes matches the number of elements in the list and the XHTML constraints on the sub-contexts match those of the holes, then plugging succeeds. The element `em` is an `inline` entity and the element `td` is a `td` entity, so plugging succeeds. As in Example 2 we obtain a one-holed context. As the hole is inside a `td` element it must be plugged with a `flow` entity. Plugging the holes does not change the type of entity ascribed to the context itself: it is still a flow entity that contains `block` entities.

Example 8

```
# let ys =
    cons (td hole,
    cons (td (text "holes"), nil));;
val ys :
('a * ('a * ((MX.blk, MX.inl) MX.flw * 'b)) s,
 'c * (('c * (MX.td * 'd)) s * (MX.td * 'b)) s)
MX.ts = <abstr>

# plug (k, ys);;
Characters 5-12:
  plug (k, ys);;
       ^^^^^^^

This expression has type
  ('a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
                ((MX.no, MX.inl) MX.flw * 'b))
   s, (MX.blk, 'c) MX.flw * 'b)
  MX.t *
  ('d * ('d * ((MX.blk, MX.inl) MX.flw * 'e)) s,
   'a * (('a * (MX.td * 'f)) s * (MX.td * 'e)) s)
  MX.ts
but is here used with type
  ('a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
                  ((MX.no, MX.inl) MX.flw * 'b))
   s, (MX.blk, 'c) MX.flw * 'b)
  MX.t *
  ('d * ('d * ((MX.blk, MX.inl) MX.flw * 'e)) s,
   'a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
                ((MX.no, MX.inl) MX.flw * 'b)) s)
  MX.ts
```

As in Example 3, we get a type error if we try to plug the wrong number of sub-contexts into a primary context. Though the type error may look rather intimidating, the important part is quite simple. The only part of the two types that differs is the second component of `MX.ts`. In the first type this has two successors, whereas in the second type it has three.

Example 9

```
# let zs =
    cons (em (text "plugging"),
    cons (tr hole,
    cons (td (text "holes"), nil)));;
val zs :
  ('a * ('a * (MX.td * 'b)) s,
   'c *
   ((('c * (MX.td * 'd)) s * (MX.tr * 'b)) s *
    (('e, MX.inl) MX.flw * 'f))
   s)
  MX.ts = <abstr>
# plug (k, zs);;
Characters 5-12:
  plug (k, zs);;
      ^^^^^^^
This expression has type
  ('a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
    ((MX.no, MX.inl) MX.flw * 'b))
   s, (MX.blk, 'c) MX.flw * 'b)
  MX.t *
  ('d * ('d * (MX.td * 'e)) s,
   'a *
   ((('a * (MX.td * 'b)) s * (MX.tr * 'e)) s *
    (('f, MX.inl) MX.flw * 'g))
   s)
  MX.ts
but is here used with type
  ('a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
    ((MX.no, MX.inl) MX.flw * 'b))
   s, (MX.blk, 'c) MX.flw * 'b)
  MX.t *
  ('d * ('d * (MX.td * 'e)) s,
   'a *
   ((('a * (MX.td * 'b)) s * (MX.td * 'b)) s *
    ((MX.no, MX.inl) MX.flw * 'b))
   s)
  MX.ts
```

If the number of holes match up, but the XHTML constraints do not, then we also get a static type error. Again the type error may look rather intimidating, but the only difference between the two types is a `MX.tr` in the first type that becomes `MX.td` in the second type. This difference exactly captures the bug: `k` is expecting a `td` entity in its second hole, but has been supplied with a `tr` entity.

As illustrated above, although the error messages are long, the parts that are relevant form only a small part of them, and once the relevant parts have been identified it is quite easy to understand what the problem is. With reference to their implementation Elsman and Larsen (Elsman and Larsen 2004) write:

> It is also our experience that type errors caused by erroneous use of XHTML combinators are understandable and pinpoint problems directly.

This author agrees, but feels that the type errors are too verbose (things are made slightly worse by the use of multi-holed contexts, but the main problem is due to the complexity of the types needed to type plain XHTML). Other systems have similar problems (Thiemann 2002). It would be interesting to follow up Peter Thiemann's suggestion (Thiemann 2002) of "filtering and translating error messages to make them more informative to casual users".

6. Limitations of the difference encoding

The difference encoding of natural numbers has allowed us to implement a list append function and plugging operations for multi-holed contexts. It should be emphasised however, that our approach is fairly limited compared to indexed types, as implemented in DML (Xi and Pfenning 1999), type classes, as implemented in GHC (Hall et al. 1994), and GADTs, as implemented in GHC (Jones et al. 2006).

As already mentioned, nonsense types can be introduced. This is not really a problem in practice though. The other aesthetic issue that has already been mentioned is the verbosity of types, and in particular type error messages. This could be more of a problem in practice, particularly when trying to scale to large examples.

A much more severe limitation is that it is difficult to write non-trivial destructors. We can easily implement safe versions of functions for computing the head and tail of a list, but a general fold operation seems hopeless, and even an operation for filtering the elements of a list matching a predicate seems tricky. The filter operation is one of the standard examples that can be implemented in programming languages that support indexed types.

One problem is that the length of the output cannot be computed statically as it depends on the dynamic predicate. So it is not clear how we could even give a type to filter. We can at least sidestep this problem by using an existential type and instead define a `partition` function that returns a pair of lists: one containing the elements for which the predicate is true and the other containing the elements for which the predicate is false. The existential type we want to define is:

```
type split_list ('l, 'n, 'a) =
  exists 'm.
    ('l*'m, 'a) NList.t * ('m*'n, 'a) NList.t
```

which allows us to assign `partition` the following type:

```
val partition :
  ('a -> bool) -> ('l*'n, 'a) NList.t ->
  ('l, 'n, 'a) split_list
```

OCaml does not directly support existential types but they can be encoded using higher-rank polymorphism via records or recursive modules. For instance, the type `split_list` can be encoded as as follows:

```
type (-'l, -'n, -'a, -'r) cont =
{k: 'm.
  ('l*'m, 'a) NList.t * ('m*'n, 'a) NList.t -> 'r}
type (+'l, +'n, +'a) split_list =
  {l: 'r.('l, 'n, 'a, 'r) cont -> 'r}
```

Although existentials allow us to specify a type for `partition`, it is still not clear how to implement the body of the function. We could attempt to use the trick we used to plug a multi-holed context, where we first perform an upcast, then perform an unsafe version of the operation, and then perform a downcast. Unfortunately that trick does not work in this case because we do not know the length of the two lists in advance. Another alternative is to attempt to define `partition` in terms of more primitive operations on `split_lists`. The problem then is that we would need different branches for empty and non-empty lists, and the non-empty list branch would have to be able to perform operations such as taking the head and tail of a list which are not well-defined on empty lists. Indexed types (Zenger 1997; Xi and Pfenning 1999), type-classes (McBride 2002) and GADTs (Cheney and Hinze 2003; Xi et al. 2003; Jones et al. 2006) each provide different solutions to this problem, at the expense of adding more complexity to the type system.

7. Related work

Encoding types The idea of encoding expressive types in Hindley-Milner type systems is not new.

Zhe Yang (Yang 1998, 2004) introduced a general scheme for encoding type-indexed families of functions in ML. Each type constructor is encoded as an function that combines the functions associated with the arguments to the type constructor to build a composite function. In effect, the encoding is the implementation of the function at a particular type. A canonical example of Yang's technique is the implementation of Type-Directed Partial Evaluation (TDPE) (Danvy 1996) in ML. Each type constructor is encoded as a pair of *reify* and *reflect* functions for converting between ML values and abstract syntax. This allows arbitrary pure ML values to be reified as abstract syntax using a type-indexed program written in plain ML. Danvy's functional unparsing (Danvy 1998), which he uses to implement a statically typed variant of the C function `printf` in ML is another widely-used example of Yang's technique.

Daniel Fridlender and Mia Indrika (Fridlender and Indrika 2000) explore a particular instance of Yang's technique (though they do not make the connection with his technique, they do cite Danvy's work on functional unparsing as inspiration). Their work is related to this article in that the types they encode are naturals: they encode functions whose type depends on natural numbers. Whereas Fridlender and Indirka encode naturals as terms, we encode them as type-level differences between Peano numbers. The two encodings are really orthogonal. The term encoding is good for defining families of functions (such as an n-ary versions of `zipWith`), whereas the type encoding is good for enforcing static properties that are otherwise difficult to express (such as the property that appending a list of length m with a list of length n gives a list of length $m + n$).

Conor McBride (McBride 2002) "fakes" dependent types using the Haskell type class system. As well as being able to implement the class of functions supported by Yang's technique, McBride's technique directly supports operations such as addition on type-level natural numbers using type classes.

Difference types Our idea of using a difference to encode addable naturals at the type-level was inspired by Didier Rémy's technique for implementing polymorphic record concatenation "for free" (Rémy 1992). He defines record concatenation on top of his implementation of polymorphic extensible records using row types (Rémy 1989). A catenable record is encoded as a function that takes a single argument representing an extension of the record and returns a row consisting of the existing fields along with the extension. Concatenation is implemented as simple composition. To project a field of a record we can simply pass in an empty extension and then project from the resulting row.

Another way of viewing Rémy's encoding is as a difference between two records: the difference between the record consisting of the fields with the extension and the record consisting of just the extension. Conversely, as described in Section 3, another way of viewing our encoding of addable naturals is as type functions taking a single argument representing an offset and returning a natural consisting of the sum of the encoded natural plus the offset. Addition is then simple composition.

A common idiom both in functional programming and in logic programming is to encode a list as a function which takes a single argument representing an extension of the list and returns a list consisting of the extension appended to the existing lists. Concatenation is implemented as simple composition. The primary motivation here is to allow concatenation to be implemented in constant time (Hughes 1986) (concatenation takes linear time for the usual linked-list representation of lists employed by typical declarative programming languages). Of course, like Rémy's representation of catenable records, our representation of addable naturals, and our representation of catenable lists, we can view the functional representation of lists as a difference: the difference between the list consisting of the extension appended to the existing list and the list consisting of just the extension. Indeed, in logic programming the term used for this idiom is *difference list* (Sterling and Shapiro 1994, Chapter 15).

Formlets Our motivation for investigating a statically typed multi-holed plugging operation was to try to improve the implementation of *formlets* (Cooper et al. 2008a), an abstraction for building web forms. The original version of formlets builds up the HTML presentation of a formlet using single-holed plugging operations. This forces the semantic part of formlets to be intermingled with raw HTML and requires nested formlets to be rebound. A clearer and more efficient approach is to use a multi-holed plugging operation. Using the ideas of this paper we have implemented a version of formlets in OCaml with statically-typed multi-holed contexts (Cooper et al. 2008b).

Acknowledgments

Thanks to Jeremy Yallop for suggesting the use of the relaxed value restriction and the anonymous reviewers for helpful feedback. This work was supported by EPSRC grant number EP/D046769/1.

References

Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Unpublished manuscript, April 2005. `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *PASTE*, pages 38–45, 2001.

James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, July 2003. `http://ecommons.library.cornell.edu/handle/1813/5614`.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical Report EDI-INF-RR-1263, School of Informatics, University of Edinburgh, 2008a.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction, 2008b. `http://groups.inf.ed.ac.uk/links/formlets`.

Olivier Danvy. Type-directed partial evaluation. In *POPL*, pages 242–257, 1996.

Olivier Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, 1998.

Martin Elsman and Ken Friis Larsen. Typing XHTML web applications in ML. In *PADL*, pages 224–238, 2004.

Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *PEPM*, pages 112–121, 2007.

Daniel Fridlender and Mia Indrika. Do we need dependent types? *J. Funct. Program.*, 10(4):409–415, 2000.

Jacques Garrigue. Relaxing the value restriction. In *FLOPS*, pages 196–213, 2004.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. In *ESOP*, pages 241–256, 1994.

Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

John Hughes. A novel representation of lists and its application to the function “reverse”. *Inf. Process. Lett.*, 22(3):141–144, 1986.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP*, pages 50–61, 2006.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001. `http://strictlypositive.org/diff.pdf`.

Conor McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(4&5):375–392, 2002.

Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *ICDT ’05*, January 2005.

Didier Rémy. Typechecking records and variants in a natural extension of ML. In *POPL*, pages 77–88, 1989.

Didier Rémy. Typing record concatenation for free. In *POPL*, pages 166–176, 1992.

Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-19338-8.

Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *J. Funct. Program.*, 12(4&5):435–468, 2002.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Hongwei Xi. Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL*, pages 224–235, 2003.

Zhe Yang. Encoding types in ML-like languages. In *ICFP*, pages 289–300, 1998.

Zhe Yang. Encoding types in ML-like languages. *Theor. Comput. Sci.*, 315 (1):151–190, 2004.

Christoph Zenger. Indexed types. *Theor. Comput. Sci.*, 187(1-2):147–165, 1997.