Many Holes in Hindley-Milner

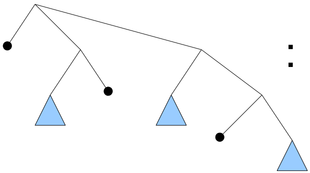
Sam Lindley

Laboratory for Foundations of Computer Science,
University of Edinburgh

Sam.Lindley@ed.ac.uk

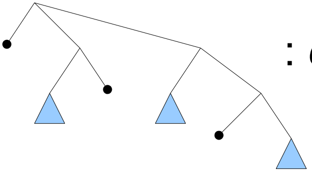
September 21, 2008

Plugging many-holed contexts

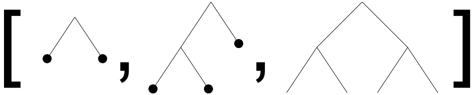


: *context*(3)

Plugging many-holed contexts

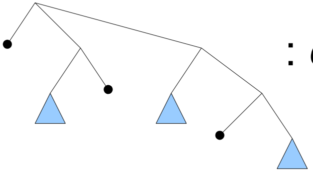


: *context*(3)

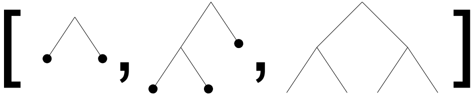


: *contextList*(2,3,0)

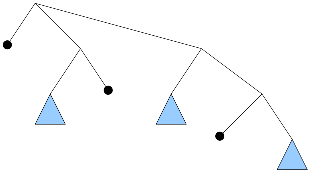
Plugging many-holed contexts



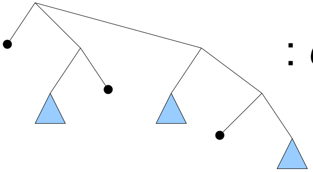
: *context*(3)



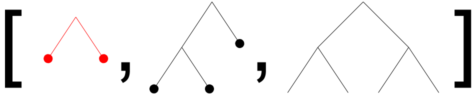
: *contextList*(2,3,0)



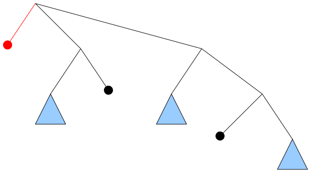
Plugging many-holed contexts



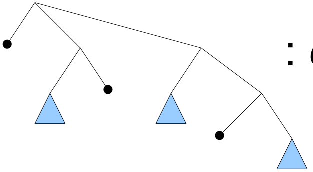
: *context*(3)



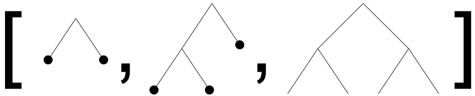
: *contextList*(2,3,0)



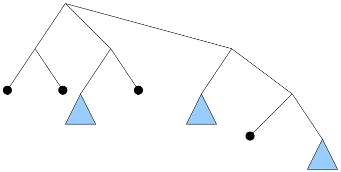
Plugging many-holed contexts



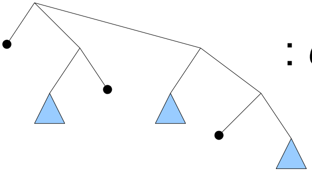
: *context*(3)



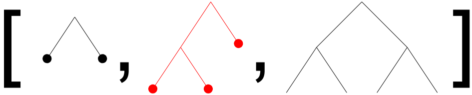
: *contextList*(2,3,0)



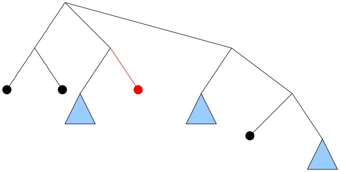
Plugging many-holed contexts



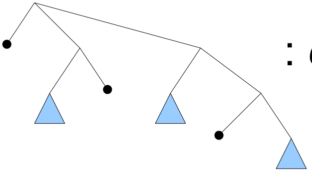
: *context*(3)



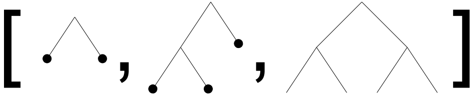
: *contextList*(2,3,0)



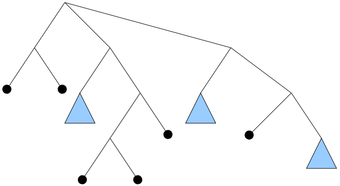
Plugging many-holed contexts



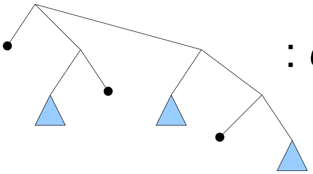
: *context*(3)



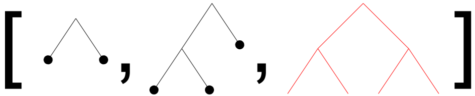
: *contextList*(2,3,0)



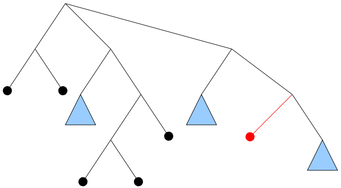
Plugging many-holed contexts



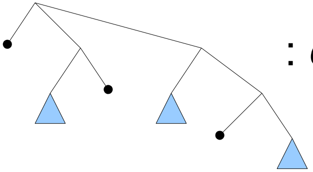
: *context*(3)



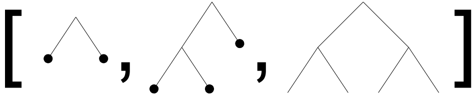
: *contextList*(2,3,0)



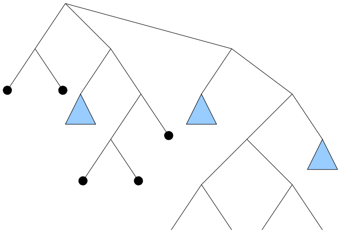
Plugging many-holed contexts



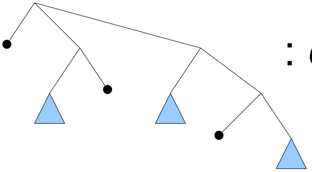
: *context*(3)



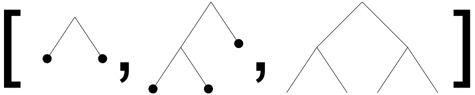
: *contextList*(2,3,0)



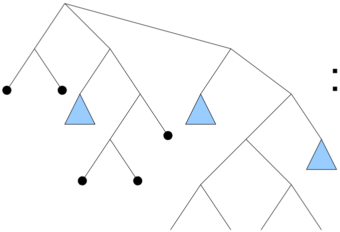
Plugging many-holed contexts



: *context*(3)



: *contextList*(2,3,0)



: *context*(5)

Lists of length n

Represent the naturals as type-level Peano numbers.

Lists of length *n*

Represent the naturals as type-level Peano numbers.

```
type z       (* zero *)
type 'n s    (* successor *)
```

Lists of length *n*

Represent the naturals as type-level Peano numbers.

```
type z       (* zero *)
type 'n s    (* successor *)
```

Represent the length of a list as a phantom type parameter.

Lists of length n

Represent the naturals as type-level Peano numbers.

```
type z       (* zero *)
type 'n s    (* successor *)
```

Represent the length of a list as a phantom type parameter.

```
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
```

Can we type list append?

From (Xi, JFP 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.*

Difference types: type-level addition in ML

Idea: represent naturals as pairs denoting the difference between two Peano numbers, then addition becomes composition: $(m, n) + (l, m) = (l, n)$.

Difference types: type-level addition in ML

Idea: represent naturals as pairs denoting the difference between two Peano numbers, then addition becomes composition: $(m, n) + (l, m) = (l, n)$.

```
module Nat : sig                                      struct
  type 'i t                                             type 'i t = int
  val zero : ('m×'m) t                                  let zero = 0
  val succ : ('m×'n) t → ('m×'n s) t                    let succ n = n+1
  val add : ('m×'n) t × ('l×'m) t → ('l×'n) t           let add (n,m) = n+m
  val to_int : 'i t → int                               let to_int n = n
end =                                                 end
```

Difference types: type-level addition in ML

Idea: represent naturals as pairs denoting the difference between two Peano numbers, then addition becomes composition: $(m, n) + (l, m) = (l, n)$.

```
module Nat : sig
  type 'i t
  val zero : ('m×'m) t
  val succ : ('m×'n) t → ('m×'n s) t
  val add : ('m×'n) t × ('l×'m) t → ('l×'n) t
  val to_int : 'i t → int
end =
```

```
struct
  type 'i t = int
  let zero = 0
  let succ n = n+1
  let add (n,m) = n+m
  let to_int n = n
end
```

Problem: syntactic non-values cannot be polymorphic.

succ zero : ('_n×'_n s) *Nat.t*

add (*succ* (*succ zero*), *succ* (*succ zero*)) : ('_n×'_n s s s s) *Nat.t*

The relaxed value restriction

Any free type variables which occur only in *covariant positions* outside of reference types can be safely generalised, even for terms that are not syntactic values (Garrigue, FLOPS 2004).

The relaxed value restriction

Any free type variables which occur only in *covariant positions* outside of reference types can be safely generalised, even for terms that are not syntactic values (Garrigue, FLOPS 2004).

```
type z        (* zero *)
type +'n s    (* successor *)
```

```
module Nat : sig                                        struct
  type +'i t                                              type 'i t = int
  val zero : ('m×'m) t                                    let zero = 0
  val succ : ('m×'n) t → ('m×'n s) t                      let succ n = n+1
  val add : ('m×'n) t × ('l×'m) t → ('l×'n) t             let add (n,m) = n+m
  val to_int : 'i t → int                                 let to_int n = n
end =                                                   end
```

The relaxed value restriction

Any free type variables which occur only in *covariant positions* outside of reference types can be safely generalised, even for terms that are not syntactic values (Garrigue, FLOPS 2004).

```
type z        (* zero *)
type +'n s    (* successor *)
```

```
module Nat : sig                                          struct
  type +'i t                                                type 'i t = int
  val zero : ('m×'m) t                                      let zero = 0
  val succ : ('m×'n) t → ('m×'n s) t                        let succ n = n+1
  val add : ('m×'n) t × ('l×'m) t → ('l×'n) t               let add (n,m) = n+m
  val to_int : 'i t → int                                   let to_int n = n
end =                                                     end
```

The covariance annotations enable generalisation!

succ zero : ('n×'n *s*) *Nat.t*

add (*succ* (*succ zero*), *succ* (*succ zero*)) : ('n×'n *s s s s*) *Nat.t*

Appending lists of length n

```
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
```

An XML context datatype

$$
\begin{array}{l}
\text{type } \mathit{xml} = \\
\quad \mid \mathit{Empty} \\
\quad \mid \mathit{Text} \text{ of } \mathit{string} \\
\quad \mid \mathit{Tag} \text{ of } \mathit{string} \times \mathit{xml} \\
\quad \mid \mathit{Concat} \text{ of } \mathit{xml} \times \mathit{xml} \\
\quad \mid \mathit{Hole}
\end{array}
$$

An XML context datatype

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

Dynamically checked plugging operation

$$dynamic\_plug : xml \times xml\ list \rightarrow xml$$

Counting holes

```
module NContext : sig
  type +'holes t

  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

  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
```

Lists of contexts

We want to be able to plug a heterogeneous list of many-holed contexts (each context in the list may have a different number of holes) into a many-holed context.

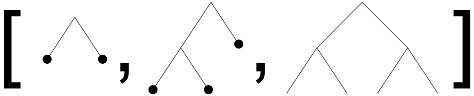
Lists of contexts

We want to be able to plug a heterogeneous list of many-holed contexts (each context in the list may have a different number of holes) into a many-holed context.

How can we represent a heterogeneous list of many-holed contexts?

We want to be able to plug a heterogeneous list of many-holed contexts (each context in the list may have a different number of holes) into a many-holed context.

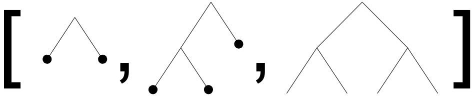
How can we represent a heterogeneous list of many-holed contexts?



: *contextList*(2,3,0)

We want to be able to plug a heterogeneous list of many-holed contexts (each context in the list may have a different number of holes) into a many-holed context.

How can we represent a heterogeneous list of many-holed contexts?



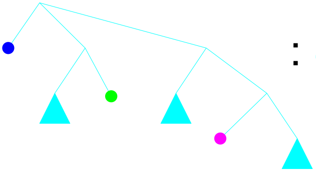
: *contextList*(5,3)

We don't need to! We only need to know the total number of holes and the total length of the list.

Plugging

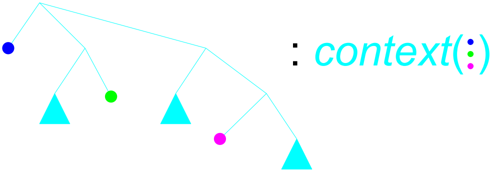
```
module NContext : sig
  ...
  type (+'holes, +'length) ts

  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

  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
```

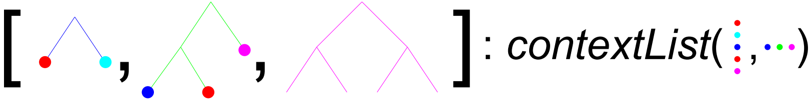
Marking holes and contexts



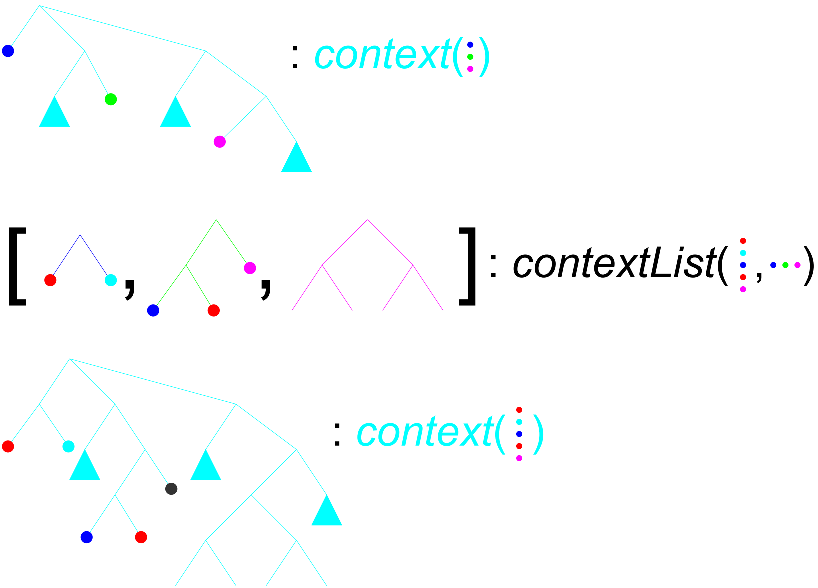
: *context*(⋮)

Marking holes and contexts





Marking holes and contexts



MiniXHTML

A small subset of XHTML introduced by Elsman and Larsen (PADL 2004) to demonstrate static typing of XHTML using phantom types.

```
<!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)+>
```

Statically typed MiniXHTML contexts

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

  type (+'holes, +'mark) t

  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

  val to_xml : ('i, 'h) t → xml
end = ...
```

Statically typed MiniXHTML contexts (continued)

```
module MX : sig
  ...
  type (+'holes, +'length) ts

  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 = ...
```

Related work

Encoding types

- Type-indexed families of functions in ML (Danvy, JFP 1998; Yang, ICFP 1998; Fridlender and Indrika, JFP 2000; ...)
- "Faking" dependent types using Haskell type classes (McBride, JFP 2002)

Difference types

- Polymorphic record concatenation for free (Rémy, POPL 1992)
- Hughes lists (Hughes, Information processing letters 1986)
- Prolog difference lists

Original motivation: *formlets*

- The essence of form abstraction (Cooper et al, APLAS 2008).
- Support syntactic sugar that allows form bindings to be embedded inside HTML.
- Basic desugaring uses a *tag* operation to build up the HTML presentation around a formlet *a single element at a time*.

  $$\mathit{tag} : \mathit{string} \rightarrow \mathit{attrs} \rightarrow \alpha\ \mathit{formlet} \rightarrow \alpha\ \mathit{formlet}$$

  - Easy to type, but the HTML and the part of the formlet abstraction that handles form input are intermingled.
- Alternative desugaring: replace *tag* with a many-holed *plug* operation.
  - Provides a clean separation between the HTML and the handling code. Relies on this work to statically type the *plug* operation.

Conclusions

- With difference types:
  - we *can* type list append in ML;
  - we *can* type many-holed plugging in ML;
  - we *can* mark holes and contexts with extra type information;
  - the relaxed value restriction helps a lot;
  - error messages are verbose.
- Type classes (GHC), indexed types (DML) and GADTs (GHC) are all more expressive than difference types.
  - In particular, they all provide better support for non-trivial destructors.