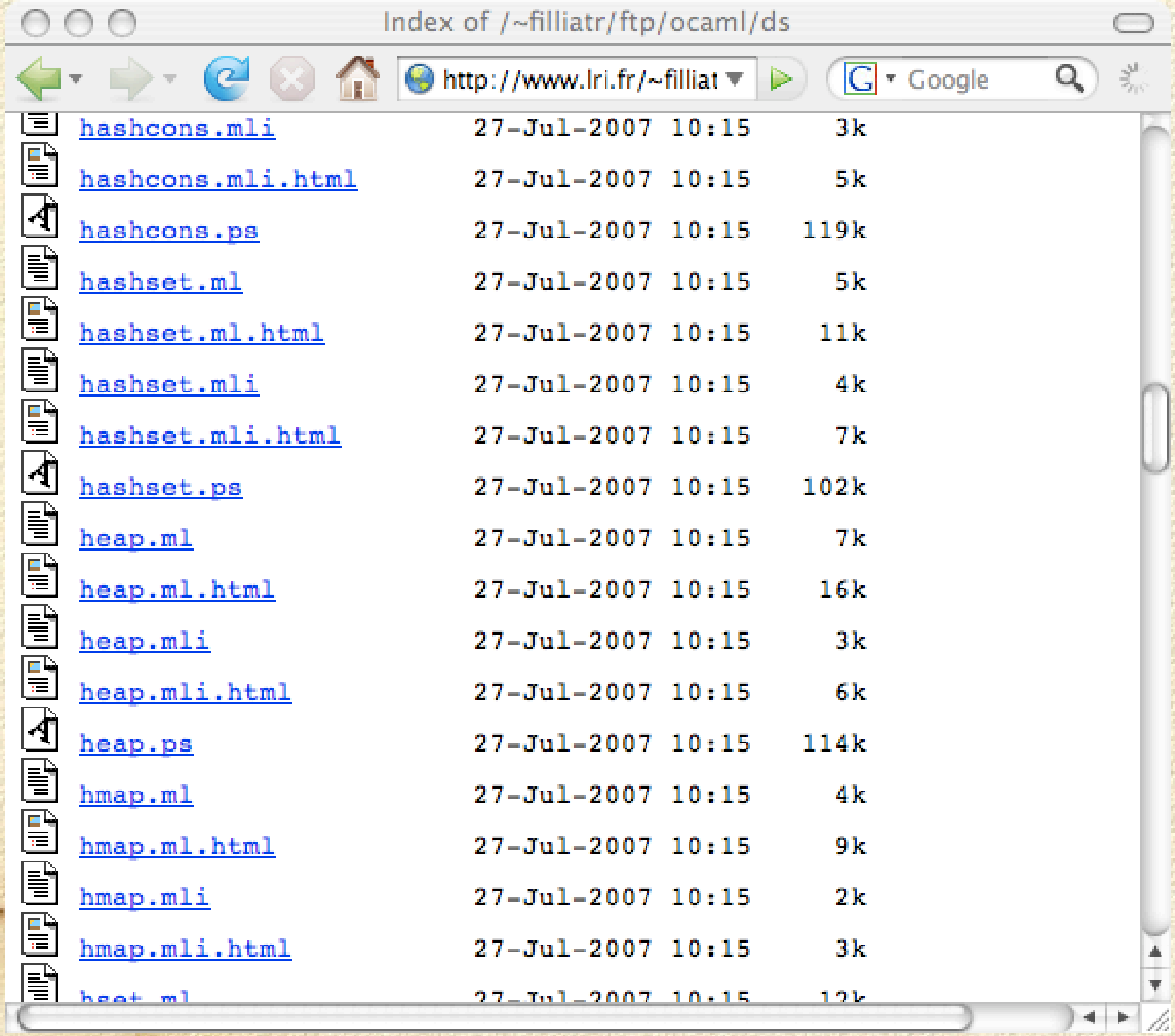
O’Caml Reins

Mike Furr

But...

- What’s wrong with the OCaml standard library?
  - Small collection of data structures
  - Closed to feature enhancements
  - Only updated with O’Caml releases
- Aren’t there existing implementations of other data structures?

Existing Implementations:



```
hashcons.mli            27-Jul-2007 10:15     3k
hashcons.mli.html       27-Jul-2007 10:15     5k
hashcons.ps             27-Jul-2007 10:15   119k
hashset.ml              27-Jul-2007 10:15     5k
hashset.ml.html         27-Jul-2007 10:15    11k
hashset.mli             27-Jul-2007 10:15     4k
hashset.mli.html        27-Jul-2007 10:15     7k
hashset.ps              27-Jul-2007 10:15   102k
heap.ml                 27-Jul-2007 10:15     7k
heap.ml.html            27-Jul-2007 10:15    16k
heap.mli                27-Jul-2007 10:15     3k
heap.mli.html           27-Jul-2007 10:15     6k
heap.ps                 27-Jul-2007 10:15   114k
hmap.ml                 27-Jul-2007 10:15     4k
hmap.ml.html            27-Jul-2007 10:15     9k
hmap.mli                27-Jul-2007 10:15     2k
hmap.mli.html           27-Jul-2007 10:15     3k
hset.ml                 27-Jul-2007 10:15    12k
```

Implemented structures

- 5 Lists (O(1) cat, random access, etc...)
- 4 Sets / 4 Maps (AVL, R/B, Patricia, Splay)
- 2 Heaps (Binomial, Skew Binomial)

O'Caml-Reins is so much more than this!

Unified Signatures

- type t = (int * float) option list
- Need a collection of t's
- Hmm... should I use a List or a Set?
- What if I change my mind later?
    - They have almost the same signature, right...?

Signature comparison

List	**Set**
No compare needed for t	Must write compare for t
Doesn't provide efficient compare for List.t	Efficient compare for Set.t
Choice of fold_left or fold_right	1 choice: fold
fold_left takes acc as first argument	fold takes acc as second argument

Before

```
type t = (int * float) option list
module T = struct
  type t = t
  (* Ugh, writing a custom compare function
     would be annoying! *)
  let compare = Pervasives.compare
end
module Collection = AVL.Set(T)
Collection.fold (fun acc x -> ...) acc0 t0

module Collection = List
Collection.fold_left(fun acc x -> ...) acc0 t0
```

With Reins

```
type t = (int * float) option list
module T = MonoList(MonoOption(MonoPair(Int)(Float)))
(* Look Ma, no boilerplate! *)
module Collection = AVL.Set(T)
Collection.fold (fun acc x -> ...) acc0 t0

type ‘a t2 = ‘a option list
module Collection = PolyList(PolyOption(PolyBase))
Collection.fold(fun acc x -> ...) acc0 t0
```

Changing data structures is as easy as changing the module definition!

Iterators

- Implemented on top of zipper-style cursors

```
type ordering =
   | PreOrder
   | InOrder
   | PostOrder
```

```
type 'a traversal =
  | Traverse_All
  | Traverse_If of ('a -> bool)
  | Traverse_While of ('a -> bool)
```

```
type direction =
  | Ascending of ordering
  | Descending of ordering
```

```
val create : direction -> 'a elt traversal -> 'a collection -> 'a t
```

Iterator Ops

Allows persistent, bi-directional C++ style navigation:

```
val at_end : 'a t -> bool
val next : 'a t -> 'a t

val at_beg : 'a t -> bool
val prev : 'a t -> 'a t
```

As well as higher order forms:

```
val iter : ('a elt -> unit) -> 'a t -> unit
val fold : ('a -> 'b elt -> 'a) -> 'a -> 'b t -> 'a
```

Benchmarks

- Unbiased benchmarks are notoriously hard to write
- Want to measure actual usage scenarios, not just 100,000 inserts
- Build on the work by Moss et al, on automatic benchmark generation

Automatic Benchmarking

- Use a *profile* which concisely summarizes a usage scenario for an ADT
  - distribution of ADT operations
  - What % of nodes are mutated/observed
  - etc...
- Basic operations on Profiles:
  - Extract a profile from an existing application
  - Construct a random program for a given profile

Benchmark Generation

- Collect a sampling of profiles (by hand or from existing applications)
- Generate a collection of random programs that fit these profiles
- Benchmark this collection
- Build a decision tree to choose to the fastest implementation based on a given profile

Oracle

- Run your application with an Oracle data structure
  - module TSet = OracleSet(T)
- Extract the profile (at_exit)
- Use the decision tree to choose best implementation
- Optimization for free!

Questions?

- Thanks to Stephen Weeks and Jane Street Capital for a putting together OSP!

Quickcheck

```
let rs = Random.State.make_self_init () in
let rand_int = Int.gen rs in
let rand_list = List(Int).gen rs ~size:100 in
  ...


  (let module T = RandCheck(struct
      module Arg = GenPair(Set)(Set)
      let desc = "[Set] Union is commutative"
      let law (t1,t2) =
         let t = Set.union t1 t2 in
         let t' = Set.union t2 t1 in
            Set.compare t t' = 0
    end) in T.test);
```