# Hashconsing in an Incrementally Garbage-Collected System

Pascal Cuoq (CEA LIST) *Damien Doligez* (INRIA)

September 21, 2008

# Plan

# Pointers

```
let x = a :: t in
let y = x in
...
```

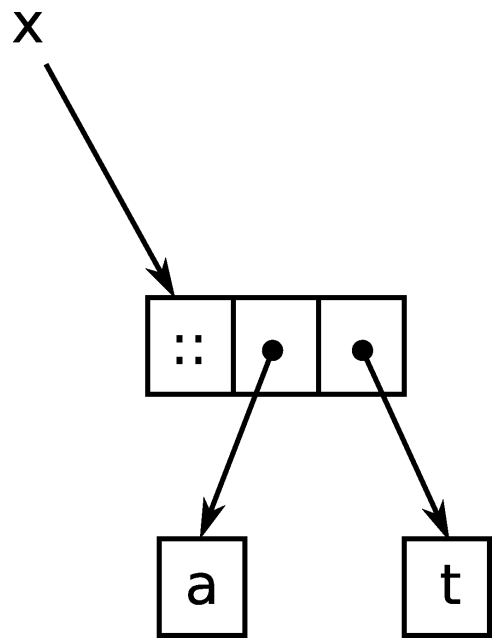
```
let x = a :: t in
let y = a :: t in
...
```

# Pointers

```
let x = a :: t in                    let x = a :: t in
let y = x in                         let y = a :: t in
...                                  ...
if x == y then printf "Hello world"
```

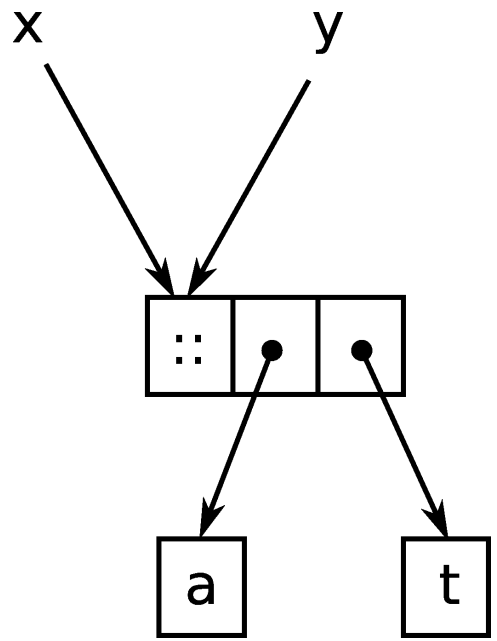
# Weak Pointers

```
let x = a :: t in
```



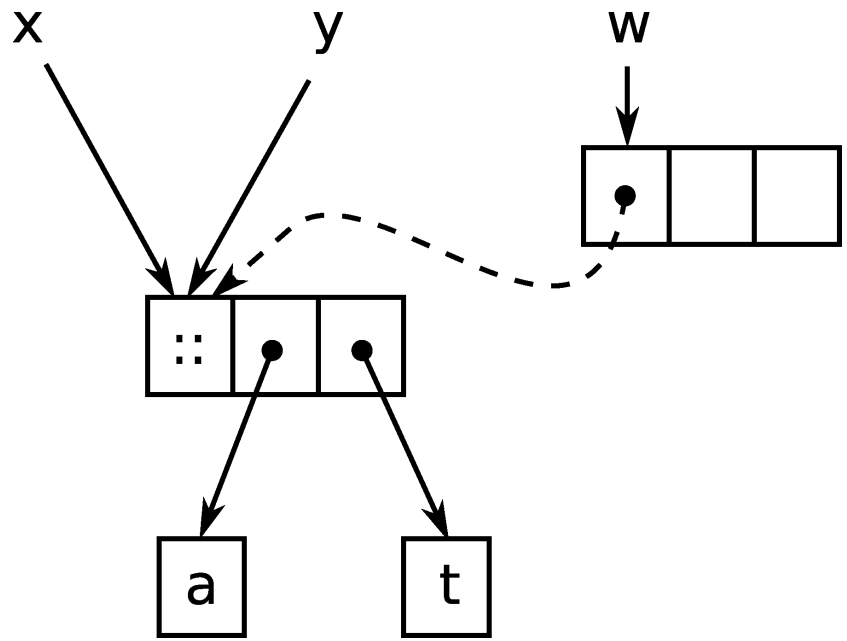
# Weak Pointers

```
let x = a :: t in
let y = x in
```



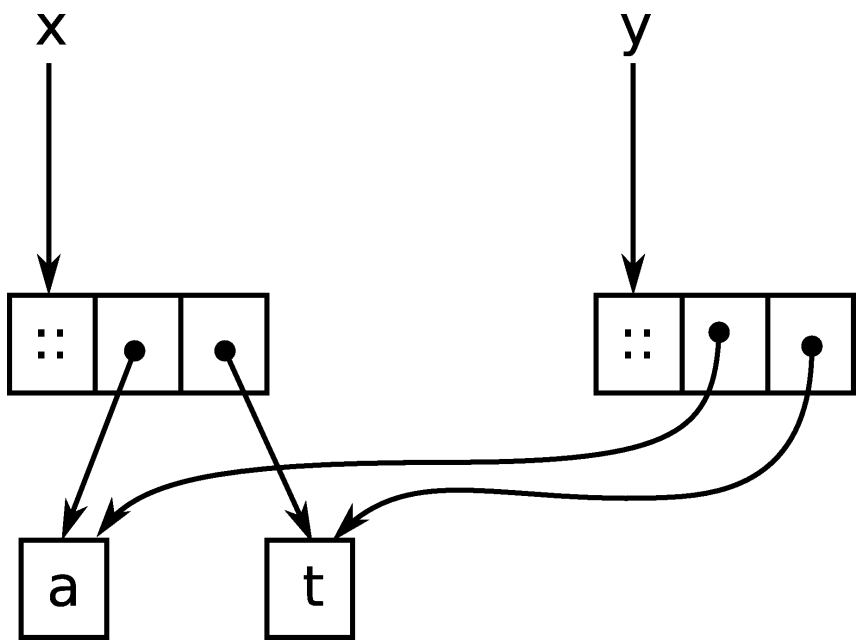
# Weak Pointers

```
let x = a :: t in
let y = x in
Weak.set w 1 x
```



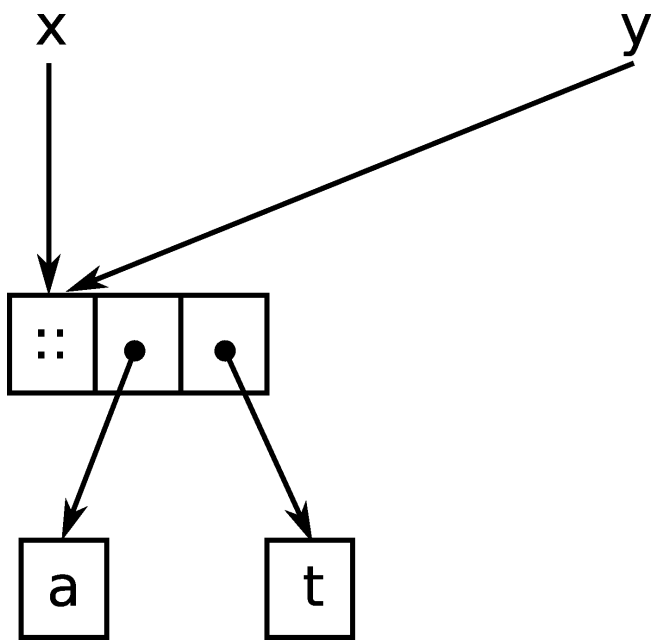
# Hashconsing 1

```
let x = a :: t in

let y = a :: t in
...
```



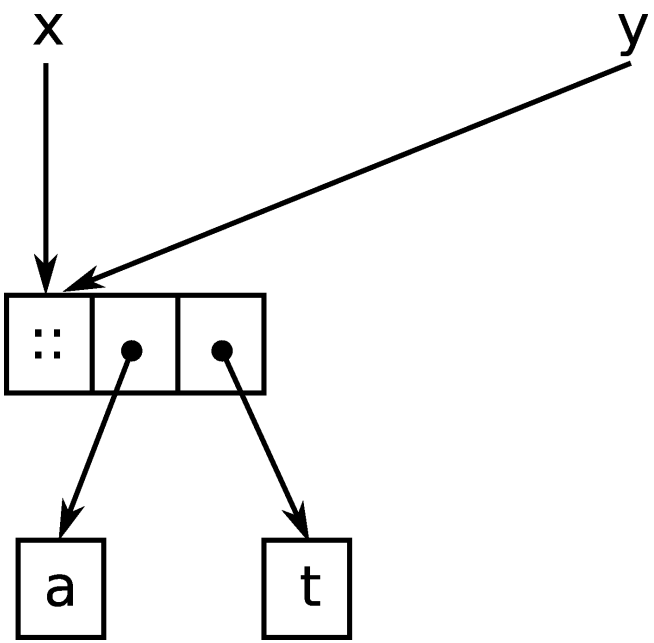
# Hashconsing 1

```
let x = a :: t in

let y = a :: t in
...
```



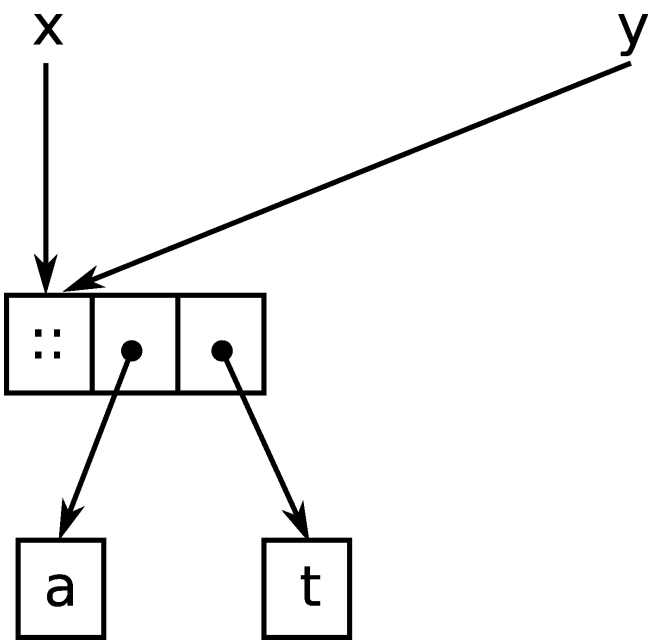
# Hashconsing 1

```
let x = a :: t in
(* arbitrary code *)
let y = a :: t in
...
```



# Hashconsing 1

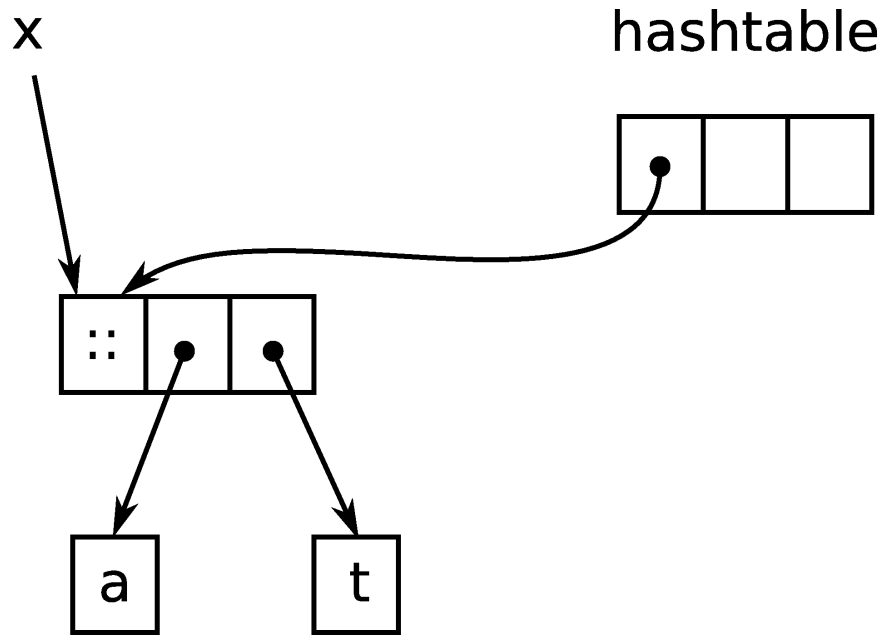
```
let x = a :: t in
(* arbitrary code *)
let y = a :: t in
...
```




Beware of mutable values !

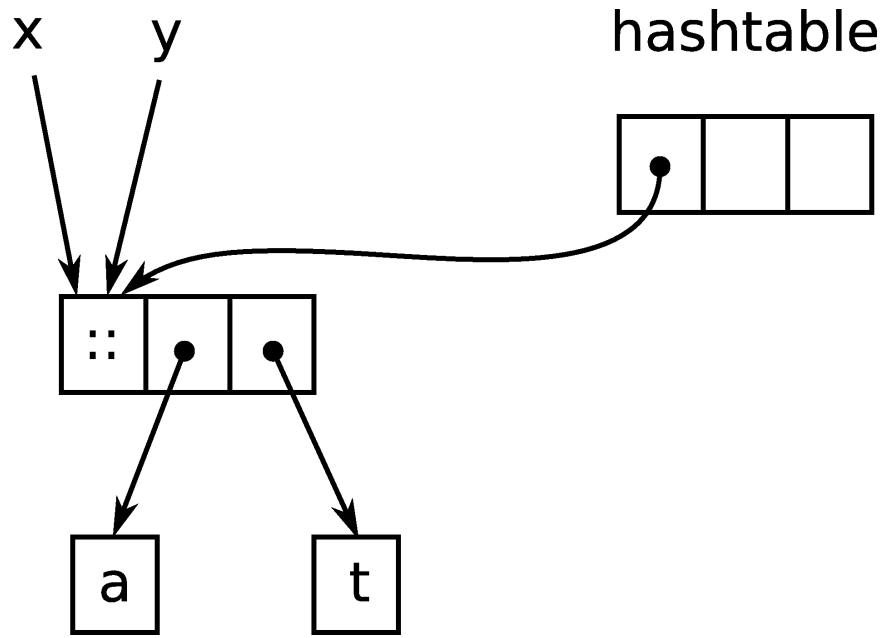
# Hashconsing 2

```
let x = cons a t in
```



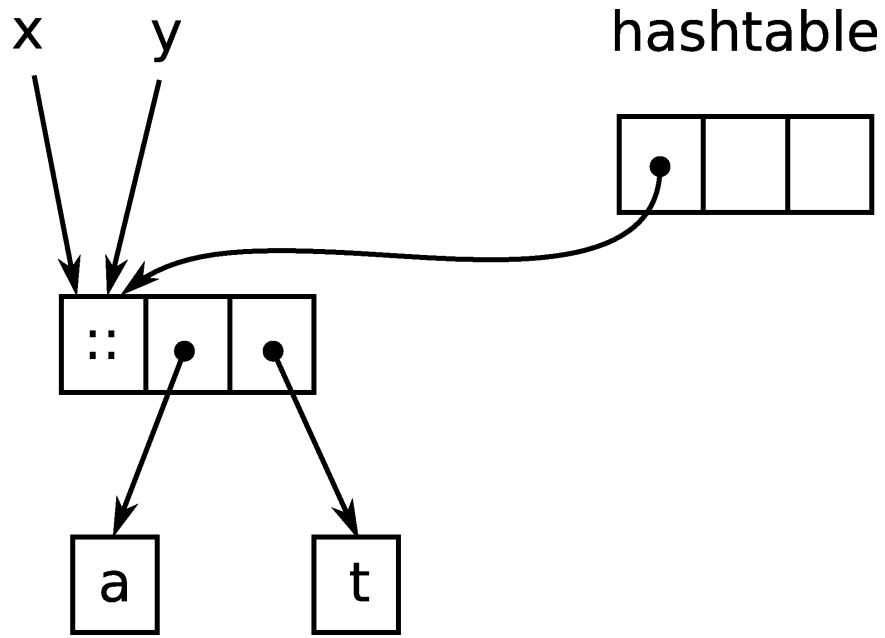
# Hashconsing 2

```
let x = cons a t in
(* arbitrary code *)
let y = cons a t in
...
```



# Hashconsing 2

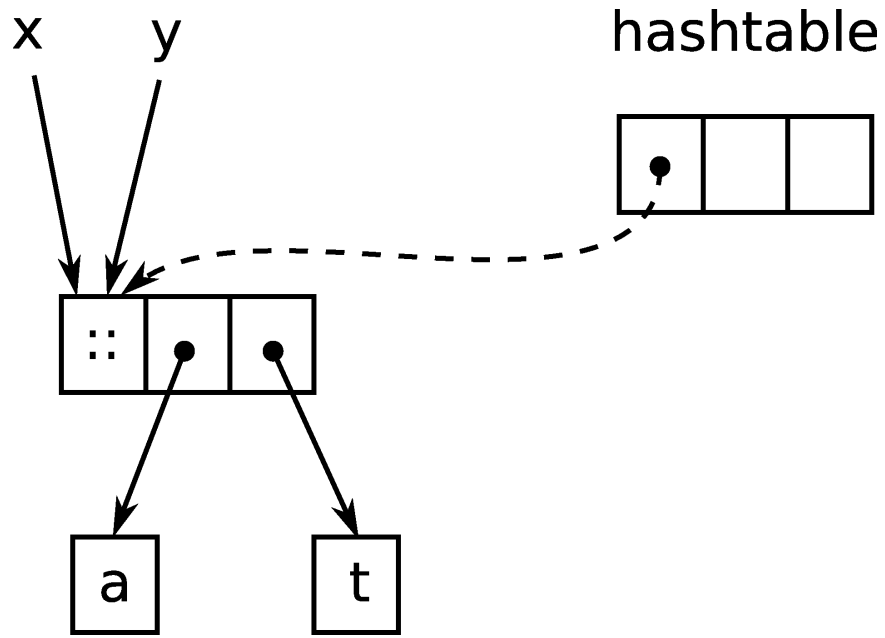
```
let x = cons a t in
(* arbitrary code *)
let y = cons a t in
...
```




Problem : we don't want to retain everything through the hashtable

# Hashconsing 2

```
let x = cons a t in
(* arbitrary code *)
let y = cons a t in
...
```




Problem : we don't want to retain everything through the hashtable
Solution : use weak pointers !

# Plan

# Implementation of Weak Pointers

Implemented in C, inside the runtime system.
Garbage Collector :

1. collect normally
2. erase weak pointers as needed

Incremental garbage collection :

- same thing, but erase weak pointers incrementally

Generational garbage collection :

- requires a little care

Must be able to enumerate all weak pointers.

- Overhead : 1 additional pointer in each weak array

# Naive Implementation of Hashconsing

Implemented in ML, based on the Weak Pointers interface. `type 'a t`

```
create : int -> 'a t
set : 'a t -> int -> 'a option -> unit
get : 'a t -> int -> 'a option
check : 'a t -> int -> bool
```

... Implements hashed *sets*, where the key is the data. This allows us to

use a user-defined comparison function to retrieve values modulo some arbitrary equivalence.
A hash table where each bucket is a weak array.

# Plan

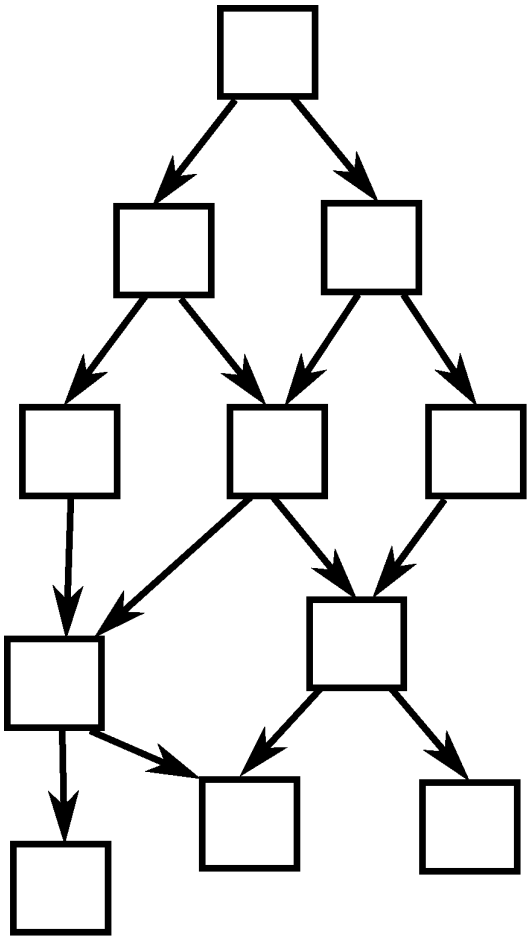
# Problems with Naive Hashconsing 1

- Each time a weak pointer is read into a normal pointer, the data must be marked reachable.
- When we call the user-defined equality function, we must give it a normal pointer.
- Because of the incremental GC, things marked reachable are not deallocated before the end of the GC cycle.

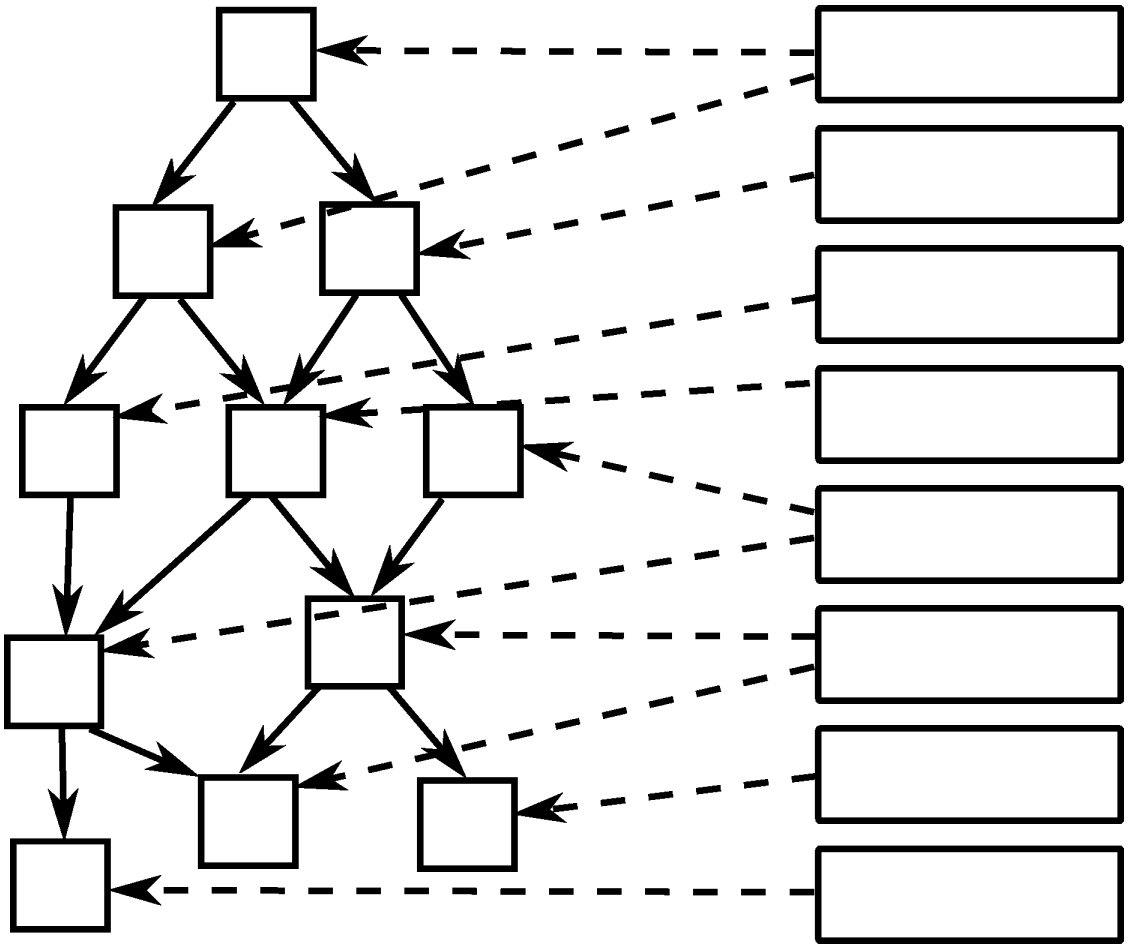
In the case of recursive datatypes (lists, terms, ...), the whole subtree under a value is reachable as soon as that value is reachable.

When searching a bucket, we compare each element, thus "refreshing" it. This leads to very slow deallocation, and the table retains values (almost) as if it was a strong hash table.

# Problems with Naive Hashconsing 2



# Problems with Naive Hashconsing 2



# Problems with Naive Hashconsing 3

We use dynamic resizing of both the table and the buckets.

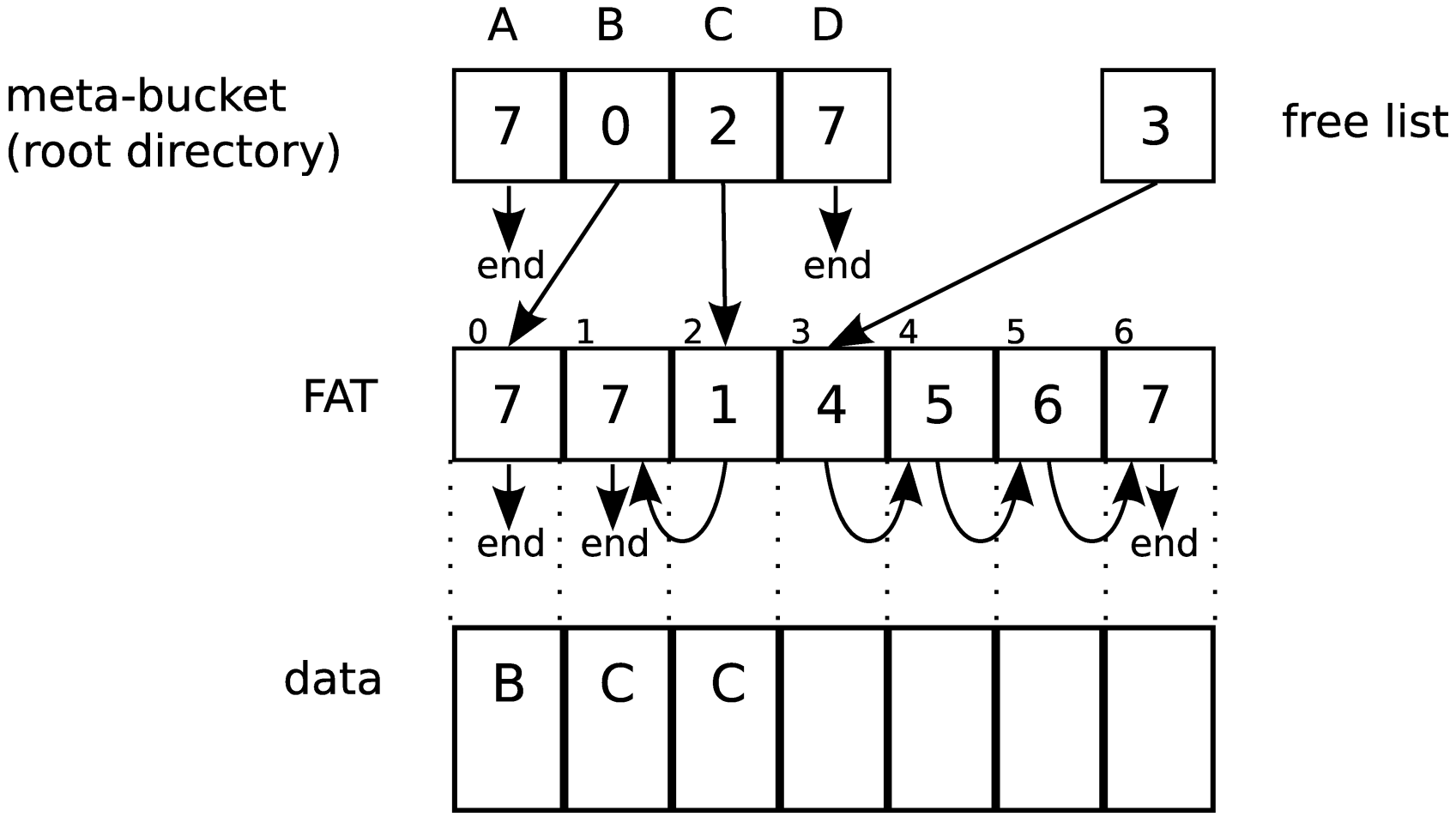
Problem : values disappear from the hash table without warning. We increase the size of the table when inserting, but we don't know when to decrease it.

With a perfectly random hash function, buckets are guaranteed to grow without bound !

# Plan

# Solution 1

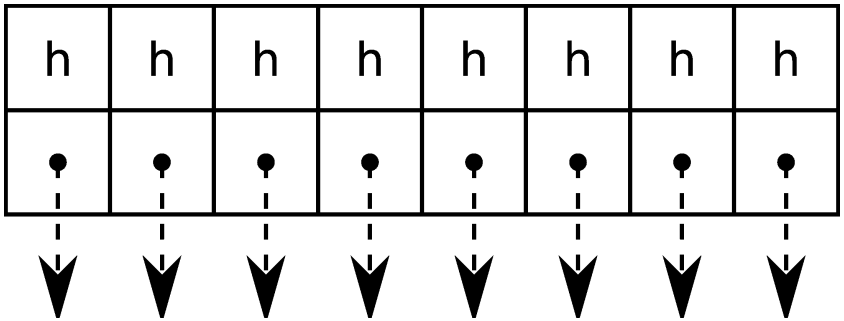
Short buckets within shared weak arrays, with FAT-like allocation. (*buckx*)



# Solution 2

Keep the hash values and call the equality function only when the hash values match.

Incremental resizing of the bucket and estimation of hash table occupancy.



# Benchmarks

| runtime | tables | time (s) | size (MiB) |
|---|---|---|---|
| 3.09.3 | naive | 101815 | >2048 |
| 3.09.3 | buckx | 44714 | 991 |
| 3.10.2 | 3.10.2 | 37472 | 936 |
| 3.10.2 | buckx | 43896 | 836 |
| 3.10.2 | buckx+ | 40269 | 854 |

# Plan

# Conclusion

Summary : hashconsing in Ocaml now works pretty well

No longer a bottleneck in static analyzer Frama-C :
one 300 000 LOC case is analyzed using 8Gb of memory

From the log files, it appears that this analysis allocates 5 millions elements worth of hasconsing tables

# Future Work

If there was a reason to keep working on the efficiency of hashconsing in Frama-C, ...

1. get rid of the cache for hash values.

```
module type Hashable = sig
type t
val hash : t -> int
val hash_in_weak : t Weak.t -> int -> int
val equal : t -> t -> bool
val equal_in_weak : t -> t Weak.t -> int -> bool
end
```

# Future Work

If there was a reason to keep working on the efficiency of hashconsing in Frama-C, ...

2. move the hashconsing implementation into the runtime.

# Thanks