Constructor unboxing
– or, how cpp decides a halting problem

Nicolas Chataing, Stephen Dolan, Gabriel Scherer, Jeremy Yallop

January 19, 2024

Low-level data representation feature for OCaml.
Theory surprise.




Single-constructor unboxing

```
type id = Id of int
```

```
Cons(Id 42, Nil)
```

Single-constructor unboxing

```
type id = Id of int

            Cons(Id 42, Nil)
```

Single-constructor unboxing (since 2016):

```
type id = Id of int [@@unboxed]

source:     Cons(Id 42, Nil)
repr:       Cons(   42, Nil)
```

Single-constructor unboxing

```
type id = Id of int

            Cons(Id 42, Nil)
```

Single-constructor unboxing (since 2016):

```
type id = Id of int [@@unboxed]

source:     Cons(Id 42, Nil)
repr:       Cons(   42, Nil)
space:      2       1   1
```

Opt-in: FFI concerns.

Constructor unboxing

Our proposed extension.

```
type bignum =
| Small of int [@unboxed]
| Big of Gmp.t [@unboxed]
```

Perf: 20% time speedup on `bignum` micro-benchmark.

Other locality benefits for space-bound programs.

Forbidding confusion

```
type t =
  | Id of int [@unboxed]
  | Error of error_code [@unboxed]
and error_code = int
```

Forbidding confusion

```
type t =
  | Id of int [@unboxed]
  | Error of error_code [@unboxed]
and error_code = int
Error: This declaration is invalid,
some [@unboxed] annotations introduce
overlapping representations.
```

A static analysis at type-declaration time:

- abstract/approximate types into *head shapes*
- fail on non-disjoint shapes

Precision tradeoff: performance, simplicity, portability.

Computing the head shape?

How to compute the head shape of a type?

Computing the head shape?

```
type 'a tree = Node of ('a * 'a tree) seq [@unboxed]
and 'a seq = Nil | Next of (unit -> 'a * 'a seq) [@unboxed]
type foo = Foo of int tree [@unboxed] | ...
```

```
  shape(int tree)
= shape((int * int tree) seq)
= shape(Nil) + shape(... -> ...)
= {(Imm, 0)} + {(Block, Obj.closure_tag)}
```

Computing the head shape?

```
type 'a tree = Node of ('a * 'a tree) seq [@unboxed]
and 'a seq = Nil | Next of (unit -> 'a * 'a seq) [@unboxed]
type foo = Foo of int tree [@unboxed] | ...

  shape(int tree)
= shape((int * int tree) seq)
= shape(Nil) + shape(... -> ...)
= {(Imm, 0)} + {(Block, Obj.closure_tag)}
```

Expanding a type definition is a β-reduction.
Call-by-name (head) normal form.

Computing the head shape?

```
type 'a tree = Node of ('a * 'a tree) seq [@unboxed]
and 'a seq = Nil | Next of (unit -> 'a * 'a seq) [@unboxed]
type foo = Foo of int tree [@unboxed] | ...
```

```
  shape(int tree)
= shape((int * int tree) seq)
= shape(Nil) + shape(... -> ...)
= {(Imm, 0)} + {(Block, Obj.closure_tag)}
```

Expanding a type definition is a β-reduction.
Call-by-name (head) normal form.

```
let rec
  tree a = seq (prod a (tree a))
  seq a = nil + (arrow unit (prod a (seq a)))
  foo = tree int + ...
in tree int
```

Cycles

```
type t = U of u [@unboxed] | Bar
and u = T of t [@unboxed]

let rec
  t = u + bar
  u = t
in t
```

Deciding termination?

(STLC, just functions, `let rec`, first-order)

Attempt 1: rule out cycles statically

“Statically”: without expanding definitions.

(As done for type synonym/aliases.)

Problem: too restrictive

```
type 'a seq = ...
type 'a tree = Node of ('a * 'a tree) seq [@unboxed]
```

Attempt 2: prevent repetition of whole types

Block if the same type expression comes up again.

```
type 'a bad = Loop of ('a * 'a) bad [@unboxed]
```

```
let rec
  bad a = bad (prod a a)
in
   bad int
→ bad (prod int int)
→ bad (prod (prod int int) (prod int int))
→ ...
```

Attempt 3: prevent repetition of head constructors

Abort if an expanded constructor comes again in head position.

Problem: too restrictive

```
let rec
  id a = a
  foo = id (id int)
in
   foo                       []
→ id (id int)                [foo]
→ id int                     [foo, id]
↛
```

Solution: annotate (sub)expressions with expansion context

Track when subexpressions *appeared* in the type,
not how they came to head position.

```
let rec
  id a = a
  delay a = id (id a)
  foo = delay int
in
   []foo
→ [foo](delay [foo]int)
→ [foo,delay](id ([foo,delay]id [foo]int)
→ [foo,delay](id [foo]int)
→ [foo]int
```

(Remark: similar to cpp termination control.)

Sound: ensures termination.
Complete (in the first-order fragment): only rejects non-normalizing terms.

Wait, is this problem decidable?

Types-list to the rescue

[TYPES] Reference request: decidability of head normalization for a pure first-order calculus with recursion.

Excellent replies by
Pablo Barenbaum, Martin Lester, Gordon Plotkin, Sylvain Salvati.

"Recursion does not always help." Plotkin (2022)
+ literature on Higher-Order Model Checking

Types-list to the rescue

[TYPES] Reference request: decidability of head normalization for a pure first-order calculus with recursion.

Excellent replies by
Pablo Barenbaum, Martin Lester, Gordon Plotkin, Sylvain Salvati.

"Recursion does not always help." Plotkin (2022)
+ literature on Higher-Order Model Checking

How to relate our first-order algorithm
to existing higher-order algorithms?

Types-list to the rescue

[TYPES] Reference request: decidability of head normalization for a pure first-order calculus with recursion.

Excellent replies by
Pablo Barenbaum, Martin Lester, Gordon Plotkin, Sylvain Salvati.

"Recursion does not always help." Plotkin (2022)
+ literature on Higher-Order Model Checking

How to relate our first-order algorithm
to existing higher-order algorithms?

Thanks! Questions?

Gordon Plotkin. Recursion does not always help. 2022.