Unfolding ML datatype declarations without loops

Nicolas Chataing, Gabriel Scherer

Partout, Inria Saclay, France

August 21, 2021
This talk

An OCaml feature we wanted: constructor unboxing.

A general (language-agnostic) problem we solved: unfolding of (recursive) type declarations, in a terminating way.
Constructor unboxing

Single-constructor unboxing: in OCaml since November 2016

    type id = Id of int [@@unboxed]

Extension proposed by Jeremy Yallop in March 2020:
OCaml RFC #14: constructor unboxing

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

(int and (Big of Gmp.t): disjoint representations)
Constructor unboxing

Single-constructor unboxing: in OCaml since November 2016

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

Extension proposed by Jeremy Yallop in March 2020:

**OCaml RFC #14: constructor unboxing**
Constructor unboxing

Single-constructor unboxing: in OCaml since November 2016

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

Extension proposed by Jeremy Yallop in March 2020:

OCaml RFC #14: constructor unboxing

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

(int and (Big of Gmp.t): disjoint representations)
Head, head shape

We define the *head* of an OCaml value, in \{Imm, Block\} × Z, by:

- the head of an immediate is the immediate itself
  \[\text{head}(42) = (\text{Imm}, 42)\]
- the head of a block is its tag
  \[\text{head}("foo") = (\text{Block}, \text{Obj.string_tag}) = (\text{Block}, 252)\]

We define the *head shape* of a type as set of heads of its values:

\[\text{head}(\tau) = \{\text{head}(v) \mid v : \tau\}\]
Unboxing specification

```
type bignum = match num with
| Small of int (* Block 0 *) | Small n -> ...
| Big of Gmp.t (* Block 1 *) | Big gmp -> ...
```

Unboxing constructors is valid if the head shapes remain disjoint.

```
type bignum = match num with
| Small of int [@unboxed] (* Imm \mathbb{Z} *) | Small n -> ...
| Big of Gmp.t (* Block 0 *) | Big gmp -> ...
```

Constructors: runtime-checkable disjointness.
(Note: This morality is language-independent.)
Problem (1/3)

How to compute the head shape of a type?

(In presence of recursive type declarations)
Problem (2/3)

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(unit -> (int * int tree) * ... seq)
= Imm 0 + function_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] | ...

\[
\text{shape}(\text{int } \text{tree}) \\
= \text{shape}(\text{(int } \star \text{ int } \text{tree}) \text{ seq}) \\
= \text{shape}(\text{Nil}) + \text{shape}(\text{unit} \rightarrow (\text{int } \star \text{ int } \text{tree}) \star \ldots \text{ seq}) \\
= \text{Imm } 0 + \text{function}_\text{shape}
\]

Expanding a type definition is a $\beta$-reduction.
Call-by-name normal form... with arbitrary recursion.
Problem (2/3)

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(unit -> (int * int tree) * ... seq)
  = Imm 0 + function_shape

Expanding a type definition is a $\beta$-reduction.
Call-by-name normal form... with arbitrary recursion.

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

How to prevent nontermination?
Problem (3/3)

How to compute the (CBN-)normal form of a type modulo unboxing?

(In presence of recursive type declarations.)

This is useful for many static analyses of types:
head shape, immediacy, etc.
Attempt 1: rule out cycles statically

“Statically”: without expanding definitions.

(As done for type synonym/aliases.)

Problem: too restrictive

```ocaml
type 'a seq = ...

type 'a tree = Node of ('a * 'a tree) seq [@unboxed]
```
Attempt 2: prevent repetition of whole types

Keep track of type inputs, abort if they come again during expansion.

Problem: may loop in presence of non-regular type parameters.

\[
\text{type 'a bad} = \text{Loop of ('a \times 'a) bad [}@\text{unboxed}\]

\[
\begin{align*}
\text{int bad} \\
\rightarrow & \ (\text{int} \times \text{int}) \text{ bad} \\
\rightarrow & \ ((\text{int} \times \text{int}) \times (\text{int} \times \text{int})) \text{ bad} \\
\rightarrow & \ ... 
\end{align*}
\]
Attempt 3: prevent repetition of head constructors

Keep track of constructors that have already been expanded. Abort if an expanded constructor comes again in head position.

Problem: too restrictive

```plaintext
type 'a id = Id of 'a [@unboxed]

type foo = Foo of int id id [@unboxed]

foo
→ int id id
→ int id
―
[]
[foo]
[foo, id]
```
Solution: annotate (sub)expressions with expansion context

type 'a id = Id of 'a [@unboxed]
type 'a delay = Delay of 'a id [@unboxed]

type foo = Foo of int delay delay [@unboxed]

foo[]
→ int[foo] delay[foo] delay[foo]
→ int[foo] delay[foo] id[foo,delay]
→ int[foo] delay[foo]
→ int[foo] id[foo,delay]
→ int[foo]

Track when subexpressions appeared in the type, not how they came to head position.
Solution: annotate (sub)expressions with expansion context

```ml
type 'a id = Id of 'a [@unboxed]
type 'a delay = Delay of 'a id [@unboxed]

type foo = Foo of int delay delay [@unboxed]

foo[]
→ int[foo] delay[foo] delay[foo]
→ int[foo] delay[foo] id[foo,delay]
→ int[foo] delay[foo]
→ int[foo] id[foo,delay]
→ int[foo]
```

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

(Stephen Dolan remarks: similar to cpp termination control.)
Termination proof

Suprisingly tricky!

https://github.com/ocaml/ocaml/pull/10479#issuecomment-876644067

With help from Stephen Dolan and Irène Waldspurger.
Completeness?

Our criterion: “Recursive calls” in type definitions must be guarded by a boxed constructor.

(Complete for the pure first-order calculus.)
Summary

Unboxed constructors: an optimization requiring type analysis.

Normalizing types in presence of cyclic references.

Thanks! Questions?