Unfolding ML datatype declarations without loops

Nicolas Chataing, Gabriel Scherer

1 Introduction

Unboxing a single-constructor datatype. ML-family languages support both *type abbreviations*, which provide a synonym for an existing type, and *datatypes* (sums/variants and records) that provide a new type (distinct from previous types) specified by its constructors or fields.

Since 4.06 (June 2016), OCaml additionally supports *unboxed datatypes*, which are single-constructor variants or single-field records that behave like datatypes during type-checking (they are distinct types) and abbreviations at runtime – constructor application or pattern-matching are erased.

```
type an_abbrev = int list
type a_datatype = Short of int | Long of int list
type an_unboxed_type = Short of int [@@unboxed]
```

(SML uses an explicit `datatype` keyword for datatype declarations, and Haskell uses `data` for datatypes, **`type`** for abbreviations and `newtype` for unboxed datatypes.)

Unboxing individual constructors within a datatype. In ocaml/RFCs#14 Jeremy Yallop proposed to allow unboxing a constructor even if the datatype has other constructors:

```
type partial_unboxing =
| Short of int [@unboxed]
| Long of int list
```

This example makes sense because, even if the `Short` constructor is elided at runtime, the representation is unambiguous: primitive integers `int` can always be distinguished from blocks like `Long of int list`.

For example, the pattern-matching clauses

```
| Short n -> ...
| Long li -> ...
```

would previously test whether the head constructor of the value is `Short` or `Long`. With the unboxed annotation, `Short` is not present at runtime, but we can instead check whether we have a primitive integer or `Long`.

In the general case, unboxing a constructor could be allowed whenever the representation of its parameter is disjoint/distinguishable from the representation of all the other values of the datatype. If this disjointness condition does not hold, we will not be able to implement pattern-matching on this datatype; the unboxing declaration must be rejected with an error.

Our work. We are working on an implementation of this proposal to allow individual constructor unboxing. In particular, to determine whether unboxing a given constructor should be allowed or rejected, we compute the *head shape* of type expressions and type declarations, that is some abstract over-approximation of the set of possible values used to check this disjointness condition.

Unfolding without loops? To compute the head shape of a type expression `int t`, we need to inspect/unfold the definition of the datatype `'a t`, which in turn requires computing the head shape of the parameters of its unboxed constructors. For example, with

```
type 'a t =
| Foo of foo [@unboxed]
| Bar of bar [@unboxed]
```

the head shape is the union of the shapes of `foo` and `bar`, whose computation may in turn require following (data)type definitions.

In presence of mutually-recursive datatypes, repeated unfolding may lead to non-termination. For a pathological example:

```
type loop = Loop of loop [@unboxed]
```

How can we compute type properties (in our case, head shape) by repeatedly unfolding datatype definitions without risking non-termination on cyclic definitions? This question is the topic of the present workshop submission.

Head tags and head shapes. We define the *head tag* h of a value to be either some (non-unboxed) datatype constructor C occurring at the head of the value, or a datatype constructor among a fixed set of *primitive* constructors $\widehat{\mathtt{t}}$ with distinguished representations ($\widehat{\mathtt{int}}$, $\widehat{\mathtt{string}}$, $\widehat{\mathtt{array}}$, $\widehat{\mathtt{tuple}}$, $\widehat{\mathtt{function}}$, etc.), or the worst approximation $\top$ which contains all possible values. We want to compute a *head shape* H for any type expression τ, which is just a list of possible head approximations.

$$h := C \mid \widehat{\mathtt{t}} \mid \top \qquad\qquad H := \emptyset \mid H, h$$

If we know when two head tags have disjoint low-level representations, and we know how to compute the head shape of a type expression, we can easily check the disjointness condition for unboxed constructors C^{unboxed} of τ: the unboxing annotation is valid if the tags in the head shape are pairwise disjoint.

In this document we do not discuss the first question (what is the low-level representation of each head tag and which definition of disjointness we use), which are low-level details related to the OCaml implementation. We only discuss how to compute those implementation-independent head shapes.

2 Type unfolding with dynamic cycle detection

Static or dynamic cycle detection? The problem of cycles also occur with type synonyms/abbreviations. OCaml will forbid cyclic type abbreviations such as

```
type 'a foo = ('a * 'a bar) and bar = 'a foo
```

This check is defined as a static check of well-formedness: the graph of *dependencies* from one abbreviation to another (the definition of `foo` mentions `bar`) must be acylic. Note that cyclic mentions in datatypes are allowed, for example this is valid:

```
type 'a foo = Cons of ('a * 'a bar)
and bar = 'a foo
```

We could follow the same approach, by considering that, unlike constructors, unboxed constructors create dependencies in this sense: a cycle of reference must go through at least one *boxed* constructor to be accepted. However, we found that this approach is too restrictive in practice. For example:

```
type 'a thunk = unit -> 'a
type 'a stream =
| Next of ('a * 'a stream) thunk [@unboxed]
| End
```

The static displine would consider that `'a stream` depends on itself (in an unboxed position) due to the occurrence of `'a stream` within an argument of `thunk`. But if we were to unfold the definition of `thunk`, we would notice that this recursive occurrence is under a function type, whose primitive tag $\widehat{\texttt{function}}$ does not depend on its input or output types.

The dynamic detection mechanism we propose is more fine-grained than the static check, and in particular accepts this declaration.

Naive dynamic cycle detection. A natural idea when performing a series of unfoldings to compute a head shape is to remember the set of type definitions that we have already expanded. If we encounter a type definiton that is already in this visited set, we are at risk of circularity and abort the computation, rejecting the definition.

However, this approach is disappointing in practice. Consider for example:

```
type 'a id = Id of 'a [@unboxed]
type t = Foo of int id id [@unboxed]
```

Computing the head shape of `t` would unfold `id` once, then in turn compute the shape of `int id`, and abort as `id` was already visited.

Call stacks. The two occurrences of `id` here are distinct and both occur in `t`. The second occurrence should thus count as a (second) dependency of `t` on `id`, not a use of `id` within itself!

Our solution is to track, for each type subexpression of our input, the definition in which it occurred (`int id` occurs in `t`). More generally, we track the path of unfoldings that led to this definition, which we call a *call stack* for this type expression. In this example, computing the shape of `t` (in the empty call stack) amounts to computing the shape of `int id id` in the call stack `[t]`, remembering that these expressions come from the definition `t`, which we can write `((int[t] id)[t] id)[t]`. Unfolding this definition brings us to the definition of `id`, so we now consider the expression `(int[t] id)[t]` (the parameter of the first `id`) in the call stack `[t, id]`. To compute this shape, we unfold the remaining `id`, but from `(int[t] id)[t]` we know that it comes from the call stack `[t]`; unfolding `id` again in this stack is not cyclic, we get `int[t]`, and terminate with the primitive tag $\widehat{\texttt{int}}$.

Note: the name *call stack* comes from viewing datatype definitions as (mutually recursive) functions, and the problem of head shape computation as the evaluation of a call to one of these recursive functions. Our call stacks really correspond call stacks for those functions, in the setting of *call-by-name* evaluation where the argument of a function is not computed until needed — but its call stack comes from its application site.

3 Our algorithm

Let us define a toy grammar for types τ and datatype declarations d.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid (\tau_i)^i\ t \\ d & ::= & \mathsf{type}\ (\alpha_i)^i\ t\ =\ \left(C_j\ \mathsf{of}\ \tau_j\right)^j\ \left(C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k\right)^k \end{array}$$

Each declaration comes with a family of (boxed) constructors and a family of unboxed constructors, either of which could be empty.

Our algorithm tracks the call stack in which type subexpressions appear. We represent this with *annotated types* $(\tau\ @\ l)$, which contain a call stack l at the top, and also on each subexpression.

$$\begin{array}{rcl} \tau\ @\ l & ::= & \alpha\ @\ l \mid \left((\tau_i\ @\ l_i)^i\ t\right)\ @\ l \\ l & ::= & \varnothing \mid l, t \end{array}$$

Head shape of a type declaration d***.***

$$\textsc{Typedecl}\ \frac{\left((\alpha_i\ @\ \varnothing)^i\ t\right)\ @\ \varnothing \Rightarrow R}{\left(\mathsf{type}\ (\alpha_i)^i\ t\ =\ \ldots\right) \Rightarrow R}$$

To compute the head shape of a type declaration type $(\alpha_i)^i t = \ldots$ we simply compute the head shape of the type expression $(\alpha_i)^i\ t$, with all type subexpressions annotated with the empty call stack $\varnothing$. This type declaration will be rejected by our implementation exactly if the computation returns `Cycle`, or if two head tags with non-disjoint representations are found in the result.

Head shape of an annotated type expression τ @ l. We can now define our algorithm as a judgment τ @ $l \Rightarrow R$, which takes an annotated type expression τ @ l and returns a *result* R, either a head shape or a `Cycle` error.

$$R \quad := \quad H \mid \texttt{Cycle}$$

$$\textsc{Var}\quad \frac{}{\prod \alpha l \Rightarrow \top}$$

$$\textsc{Cycle}\quad \frac{t \in l}{\left((\alpha_i \text{ @ } l_i)^i\ t\right) \text{ @ } l \Rightarrow \texttt{Cycle}}$$

$$\textsc{Prim}\quad \frac{(t, \widehat{\mathtt{t}}) \in \widehat{\mathrm{T}}}{\left((\alpha_i \text{ @ } l_i)^i\ t\right) \text{ @ } l \Rightarrow \widehat{\mathtt{t}}}$$

$$\textsc{Type}\quad \frac{t \notin l \qquad \begin{array}{c}\texttt{type}\ (\alpha_i)^i\ t\ =\ (C_j\ \texttt{of}\ \_)^j\ (C_k^{\texttt{unboxed}}\ \texttt{of}\ \tau'_k)^k \\ \forall k, \quad \tau'_k[\alpha_i \leftarrow \tau_i \text{ @ } l_i]^{@l,t} \Rightarrow R_k\end{array}}{\left((\tau_i \text{ @ } l_i)^i\ t\right) \text{ @ } l \Rightarrow \texttt{merge}\left((C_j)^j, (R_k)^k\right)}$$

When computing the shape of the datatype $(\alpha_i)^i\ t$, the datatype parameters α_i could get instantiated with types of any shape. Our rule Var thus gives those type variables the shape $\top$.

Before unfolding a type constructor, we check that it is not already in our call stack. If it is, the rule Cycle aborts with a `Cycle` result.

We assume a relation $\widehat{\mathrm{T}}$ from certain datatype constructors to primitive shapes, used in the rule Prim.

The rule Type performs the unfolding of a datatype definition. Our input is a datatype $(\tau_i \text{ @ } l_i)^i\ t$ at some call stack l. We lookup the definition of the datatype $(\alpha_i)^i\ t$ in the global datatype definition environment, split into a family of boxed constructors and a family of unboxed constructors. The resulting shape is obtained by concatenation of the tags of boxed constructors, and shapes of unboxed constructor arguments: for each boxed constructor C_j of τ_j we add the tag C_j, and for each unboxed constructor $C_k^{\texttt{unboxed}}$ of τ'_k we add the shape of τ'_k. There are two subtleties in the rule:

- Computing the shape of a τ'_k may fail with a `Cycle` error; we use a `merge` $(\ldots)$ operator that propagates this error, or concatenates the result shapes.
  We define it below.
- The τ'_k mention the formal datatype parameters $(\alpha_i)^i$ of the datatype declaration. We substitute them with the actual datatypes parameters $(\tau_i)^i$ used the type expression at hand. More precisely, we have annotated types $(\tau_i \text{ @ } l_i)^i$, which we substitute under a non-annotated type τ'_k. To get an annotated type as expected, we need to use a call stack for all type subexpressions of τ'_k that are not variables (each variable α_i gets the call stack l_i from our input). Those subexpressions get the call stack l, t, tracing the fact that they come from the expansion of t in the current context l. To summarize, our substitution operation $\tau[\sigma]^{@l}$ takes an unannotated type τ, a substitution σ from its type parameters to *annotated* types τ_i @ l, and an "ambiant call stack" l to use on all non-variable type expressions. We define it below.

$$\begin{array}{lcl}\texttt{merge}\,(\ldots, \texttt{Cycle}, \ldots) & = & \texttt{Cycle} \\ \texttt{merge}\,\left((H_i)^i\right) & = & (H_i)^i\end{array}$$

$$\begin{array}{rcl}\alpha[\sigma]^{@l} & = & \sigma(\alpha) \\ (\tau_i)^i\ t[\sigma]^{@l} & = & \left((\tau_i[\sigma]^{@l})\ t\right) \text{ @ } l\end{array}$$

3.1 Termination

We have a proof sketch (not a complete proof yet) of termination that goes as follow.

We say that the *parent* of a non-empty call stack l, t is the call stack l.

We have to prove that any potentially-infinite derivation of $d \Rightarrow R$ is finite. It suffices to prove that any path of applications of the rule type within such a derivation is finite. Indeed, the three other rules, Var, Cycle and Prim, terminate the derivation, and the width of each Type application is bounded by the maximal number of constructors of a type declaration in the global environment (which we assumed finite.

For a path P of Type applications we look at the input types $(\tau_p \text{ @ } l_p)^{p \in P}$ along this path, and in particular the path of locations $(l_p)^{p \in P}$. (We order P by position in the derivation, with the smallest element p_0 corresponding to the root of the derivation.)

Lemma. *If the stack l_p occurs at position p in the stack, all prefixes of l can be found at some earlier position $p' \leqslant p$.*

We build a *tree* structure on this path P. Note: the tree is not the derivation, is is overlayed over one linear path.

The root of the tree is the first element p_0 of the path. Remark that it is the unique element with the empty call stack $\varnothing$.

Any other element of the path has a call stack l, t; we place it as a children of the closest element earlier in the trace with the parent call stack l, which must exist by the lemma above. By construction, the parent of a child in the tree has the parent call stack.

Our termination argument is that this tree is finite (so the trace is finite):

- Its height is bounded: a call stack l can only mention each datatype once, so the height of any call stack (and thus of the tree) is bounded by the number of datatypes in the global environment.
- Each node has a finite number of children. Intuitively the argument is if a node corresponds to some type $(\tau_i)^i\ t$, each child corresponds to one sub-expression occurring in the expansion of the datatype t, with each sub-expression occurring at most once among the children. (This is the delicate part of the proof.)