Specifying the unboxability check on mutually recursive datatypes in OCaml

Simon Colin

March 12, 2018

1 Introdction

The OCaml programming language recently started allowing user to unbox single constructor single field types which allows the values to be represented only as the value of their field rather than a tag and the value, this allows a slight improvement in speed and memory usage. Such types are defined by using the `[@@unboxed]` annotation.

```
type name = Name of string [@@unboxed]
```

This type can contain float and non float values which could mean segmentation faults if unboxed.

Unfortunately a quirk of the language is that all base values (int, char, bool, ...) are stored on a single memory word except for float which is stored on two, this would be fine were it not for the fact that that the size of the fields of an array is determined by its first value. Indeed, this means that if we were to have a type that can contain both float and non float values and this type was unboxed, we would be able to achieve segmentation faults by putting both floats and non floats in the same array. This can be demonstrated by using the unsafe cast function `Obj.magic` as shown below :

```
# let array = Array.create_float 3;;
val array : float array = [|0.; 0.; 0.|]
# Array.set array 1 (Obj.magic 1);;
Segmentation fault (core dumped)
```

OCaml features generalized algebraic datatypes which among other things allow the definition of existentially quantified type variables, such type variables can be used to define a type able to contain both float values and non float values. The compiler/interpreter does not allow us to unbox such a type as demonstrated below :

```
# type ext = E : 'a -> ext [@@unboxed]
Error: This type cannot be unboxed because
      it might contain both float and non-float values.
      You should annotate it with [@@ocaml.boxed].
```

Determining whether a single type can contain both float values and non float is quite straightforward, however the matter becomes more complicated when dealing with several mutually recursive types. This was pointed out by user mmottl (`https://github.com/ocaml/ocaml/pull/606#issuecomment-248656482`) who tried code that amounted to the following :

```
# type ('a, 'b) t =
  | R : 'a * int -> ('a, int) t
  | I : 'a u -> ('a, int) t
  and 'a u = U : ('a, _) t -> 'a u [@@unboxed];;
Error: This type cannot be unboxed because
       it might contain both float and non-float values.
       You should annotate it with [@@ocaml.boxed].
```

To determine which types can be unboxed, we took inspiration from the way the variance(covariance, contravariance, ...) of type definitions is computed in OCaml and decided to use a mode which is either Separable (Sep), that is to say the type contains either only float values or only non float values, or Independent (Ind), in which case nothing is said about the values contained within the type. We then worked out what it meant for a type to be separable and then phrased the results of this reflection as a set of inference rules that should hold if all the types and their parameters are as they should be, should this not be the case we can identify an offending parameter or type and correct it or we are trying to unbox a type that cannot be unboxed and we return an error. Doing this until a fixpoint is reached or we fail allows us to check the set of type declarations and establish constraints on the type parameters. This set of inference rules was then implemented on a simplified model of the language where it correctly identified non unboxable types and returned correct constraints on unboxable ones.

The inference rules were worked out during regular meetings with Gabriel Scherer and I wrote the implementation while receiving regular feedback and advice from Gabriel Scherer.

2 Rules

2.1 Refining the model

One relevant feature of OCaml that was omitted in the introduction for the sake of brevity is that through the use of constraints, it is possible to extract single values from a given type, which led to the creation of a third mode Deepsep which defines a type that is made of only Deepsep types, the base types being Deepsep.

```
type 'a constrained_t = 'b constraint 'a = 'b * int

type must_be_boxed = Any : ('a * int) constrained_t -> must_be_boxed
```

`constrained_t` is only the first part of the tuple that it takes as a parameter which in the case of `must_be_boxed` is an existentially quantified variable

2.2 Notations

We have also defined a ”product” of modes that performs according to these rules:

$$\mathsf{Deepsep}.m = \mathsf{Deepsep} = m.\mathsf{Deepsep} \qquad \mathsf{Sep}.\mathsf{Ind} = \mathsf{Ind} = \mathsf{Ind}.\mathsf{Sep} \qquad m.m = m$$

The set of type declarations is noted as Def, a type definition is made of a list of pairs of type variables and modes that are the parameters as well as the mode that the parameter needs to have, the name, definition and the mode that this type needs to have.

$$\frac{}{\Gamma \vdash \mathsf{int} : m} \qquad \frac{}{\Gamma \vdash \mathsf{bool} : m} \qquad \frac{}{\Gamma \vdash \mathsf{char} : m} \qquad \frac{}{\Gamma \vdash \mathsf{float} : m}$$

The ground types have all the modes since they are Deepsep and Deepsep is a subset of Sep which itself is a subset of Ind.

$$\frac{Def; \Gamma \vdash A : m.\mathsf{Ind} \qquad Def; \Gamma \vdash B : m.\mathsf{Ind}}{\Gamma \vdash A \to B : m} \qquad \frac{Def; \Gamma \vdash A : m.\mathsf{Ind} \qquad Def; \Gamma \vdash B : m.\mathsf{Ind}}{\Gamma \vdash A \times B : m}$$

Types whose definition is not a single value are separable if the values are Ind and Deepsep if the values are Deepsep, this can be written as that they are of mode m if their internal values are of mode m.Ind. For example if $m = \mathsf{Sep}$, the first rules becomes

$$\frac{Def; \Gamma \vdash A : \mathsf{Ind} \qquad Def; \Gamma \vdash B : \mathsf{Ind}}{\Gamma \vdash A \to B : \mathsf{Sep}}$$

Whereas if $m = \mathsf{Deepsep}$ it becomes

$$\frac{Def; \Gamma \vdash A : \mathsf{Deepsep} \qquad Def; \Gamma \vdash B : \mathsf{Deepsep}}{\Gamma \vdash A \to B : \mathsf{Deepsep}}$$

$$\frac{(\mathsf{type}\ (\alpha_i : \_)^I t) \in Def \qquad \Gamma \vdash T[\alpha_i : A_i]^I : m}{Def; \Gamma \vdash (A_i)^I t : m}$$

For an instance of a parameterized type to be of a given mode, its body needs to be of that mode after you replace the type variables with the parameters given.

$$\frac{(\mathsf{type}\ (\alpha_i : m_i)^I t) \in Def \qquad \forall i \in I, \Gamma \vdash A_i : \mathsf{Sep}.m_i}{Def; \Gamma \vdash (A_i)^I t : m}$$

Gabriel{I defined a macro `\type` to have the right spacing (see in this rule). TODO: use it in all your rules.**}**

If all the parameters of an abstract or not parameterized type are of the mode that they are supposed to be then this type is of the mode it’s supposed to be.

$$\frac{Def = (\mathsf{type}\ (\alpha_i : m_i)^I t_j = T_j : m_j)^J \qquad \forall j, Def; (\alpha_i : m_i)^I \vdash T_j : m_j}{\vdash Def}$$

$$\frac{(\mathsf{type}\ (\alpha_i : m_i) t_i = \_ : m) \in Def \qquad \forall i, Def; \Gamma \vdash (: A_i) t_i : m_i}{Def; \Gamma \vdash (A_i) t_i : m}$$

The set of definitions is well formed if every type evaluates to the mode it’s supposed to be.

$$\frac{Def; \Gamma, \alpha : ind \vdash T : m}{Def; \Gamma \vdash \exists \alpha, T : m}$$

For types that feature an existential variable, this variable is capable of containing both float and non float values, this means that it will never be able to be of any mode other than Sep. For our purposes, we can choose to view a type *ext* featuring an existential variable a that is supposed to be of mode m as a type T parameterized by this variable and then check if T is of mode m, while keeping in mind that if T being of mode m requires a to be aything other than Sep the type cannot be safely unboxed.

3 Implementation

In the implementation, after defining the types to represent modes, type variables, type names, type definition bodies and type definitions, we defined base operations needed for dealing with them such as mode inclusion or mode product. Once these were defined it was possible to start dealing with the definitions, this was done at three levels :

- `check_type` which checks whether a definition body has a given type, if this is the case, it returns the empty set, otherwise it returns the set of offending type variables, in the event of a type that is not unboxable this function raises an error.
- `check_def` which calls this function on the non trivial (single base type) definitions it is asked to check
- `check` which calls `check_def` on every definition from the set of definitions to check, in the event of `check_def` not returning an empty list, check updates the definition in question and then starts again from the first definition.

```
type tyvar = Var of string
type tyname = Name of string
type mode = Sep
          | Ind
          | Deepsep

type ty = Unit
        | Int
        | Float
        | Bool
        | Char
        | Arrow of (ty * ty)
        | Cons of (ty * ty)
        | Or of (ty * ty)
        | Tyvar of tyvar
        | Param of (ty list * tyname)
        | Exists of tyvar * ty
        | Abstract of tyname

type def = Def of (tyvar * mode) list * tyname * ty * mode

val check_type : ty -> def list -> mode -> (tyvar * mode) list
val check_def : def -> def list -> mode -> (tyvar * mode) list
val check : def list -> def list
```

To check whether the functions were performing as expected, an example type as well as examples that are instances of the typical scenarios that we can expect were defined and checked.

4 Discussion

The choice to require a type featuring a constraint be deepsep is overly conservative, one could image a system that propagates constraints on the type in question and thus ensures that only the minimal set is required to be sep, however constraints are a quite advanced feature, as is unboxing types so it's worth considering whether having the most accurate set of requirements is a worthwhile investment of time, this is why the compromise was made to settle for this imperfect solution.

Another area for improvement that could not be dealt with because it was noticed too late is that equations can be put on existential type variables as shown below :

```
type 'a equations = E : bool -> ('b * int) equations
```

In this case `'a` must satisfy `'a = ('b * int)`

A quick fix for this would be to apply the current implementation if the existential variable doesn't appear in the equations, if it appears in a type that is deepsep, assuming the existential variable to be deepsep and if the type is sep, assuming the existential variable to be sep if the type is equal to the existential variable only. The implementation in its current state is thus not quite in agreement with the actual OCaml language and slightly too demanding, however it still represents an improvement over the current system.

Had time allowed for it it would also have been of interest to continue this work to its logical conclusion and actually implement this in the OCaml compiler.