Titre

Simon Colin

March 10, 2018

Gabriel{TODO: choose a title for your report :-**}**

1 Intro

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.

Gabriel{TODO: give an example of unboxed type definition here.**}**

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.

Gabriel{TODO: give an example of creation of a array of floats, an array of non-float, and point out that they have different representations in Memory. You could also give an example of a segfault occurring if you create a float array, and try to write `(Obj.magic 'a' : float)` in it. (`Obj.magic` is an unsafe cast function.)**}**

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.

Gabriel{Give a definition of GADT that contains an existentially quantified type variable, and can contain either float and non-float values. Point out that it is invalid to unbox it, and show the same example with an `unboxed` annotation and the error message.**}**

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. **Gabriel{**We need an example of mutually-recursive type definition where you want to use `unboxed`. You could use the one from the bug report of Markus Mottl, and point to the bug report (give the URL etc.).**}**

```
type any\_t = Any 'a -> any\_t
```

This type can contain float and non float values which could mean segmentation faults if unboxed. **Gabriel{**You shouldn't need explicit spacing constructs (bigskip), and also you don't need to escape underscore characters_ inside `lstlisting` or `\code` commands**}**

To determine which types can be unboxed, we 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 **Gabriel{**Independant**}**Independent (Ind), in which case nothing is said about the values contained within the type. **Gabriel{**I defined macros for Ind, Sep and Deepsep, so that they are printed in a consistent way; currently they have a different font in math parts and in non-math parts. The macros can only be used in math mode; if you need to mention the modes in the middle of text, put dollar signs around them. (Look at the source of this paragraph for an example.). So TODO: replace all uses of Ind, Sep or Deepsep in your document by these macros.**}** 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. **Gabriel{**Remark: I like this paragraph above. I think that the earlier part where you explained the problem (the reason for the work) could be more clear, but your description here of the work you did is nice.**}**

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.

Gabriel{You need to add a code example which demonstrates the problem: a declaration of a type `'a constrained` with a constraint, and then another declaration using this type that cannot be safely unboxed, although the parameter passed to `constrained` is separable.**}**

2.2 Notations

We have also defined a "product" of modes such that **Gabriel**[Deepsep * anything]**{**(1) use m rather than "anything", and (2) I think you are using . instead of * below.**}** is Deepsep, Ind * Sep is Ind and a type * itself is that same type.

Gabriel{You could explain the rule for multiplications in a math section instead of prose:**}**

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

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.

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

Gabriel{TODO: use `int` or int instead of *int* – same thing for other ground types.**}**

The base types **Gabriel**[of all]**{**have all**}** the modes.

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

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

Types whose definition is not a single value are separable if the values are independant 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. **Gabriel{**For example, if $m = \mathsf{Sep}$, this rules becomes equivalent to: (give the simplified rule in a `mathpar`) where if $m = \mathsf{Deepsep}$, this rules becomes equivalent to: ...**}**

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

$$\frac{(type(\alpha_i : \_)t = (K_j \ of \ (A_i)^{I_j})^J) \in Def \qquad \forall j \in J, \Gamma \vdash K_j[\alpha_i : A_i]^{I_j} : m}{\Gamma \vdash (A_i)^I t : m}$$

Simon{Isn't this last rule included in the one just before?**}Gabriel{**The second rule indeed is strange (I think the second premise on the top-right is wrong**}** and subsumed by the one above. In the judgment for types, when we encounter a parametrized type $(A_i)^i t$, we don't need to depend on how t was declared,

only on its mode in the Def part. It is in the judgment for definitions or type declarations that we should distinguish the various ways to declare a type.

For an instance of a parameterized type to be of a given mode **Gabriel**[m]**{**TODO: when you refer to mathematics inside text, use the math mode with dollars. For example, "m" here should be "m" instead.**}**, its body needs to be of that mode after you replace the type variables with the parameters given.

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

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.

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

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

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 with an existential variable, they are of a given mode if the parameterized type with the same body and the existential type variable as single parameter being of this type without requiring the existential type variable to be separable or deepsep since this existential variable can contain both float values and non float values.**Gabriel{**I needed to read this sentence three times to understand it. Can you rephrase it?**}** **Gabriel{**TODO: give examples of types (containing existential quantifications) that are and are not separable, so that people have concrete examples to run the rule on?**}**

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:**Gabriel{**TODO: use LaTeX lists instead of plain lists below**}** - `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.

Gabriel{you should show the concrete signatures with types:**}**

```
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.**Gabriel{**Can you give an example?**}** 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.

The idea of using inference rules to compute a fixpoint was inspired by the way variance is computed. **Gabriel{**This information could actually be mentioned at the end of the information, as it may help readers follow the rest of the work. (You could be more explicit, instead of just "variance", say something like: "variance (covariance, contravariance) of type definitions", to remind people what it is.)**}**