Unboxing Mutually Recursive Type Definitions in OCaml

Simon Colin, Rodolphe Lepigre[1], and Gabriel Scherer[2]

[1] Inria, LSV, CNRS, ENS Paris-Saclay, France – rodolphe.lepigre@inria.fr
[2] Inria, France – gabriel.scherer@inria.fr

October 2018

Abstract

In modern OCaml, single-argument datatype declarations (variants with a single constructor, records with a single immutable field) can sometimes be "unboxed". This means that their memory representation is the same as their single argument, omitting an indirection through the variant or record constructor, thus achieving better memory efficiency. However, in the case of generalized/guarded algebraic datatypes (GADTs), unboxing is not always possible due to a subtle assumption about the runtime representation of OCaml values. The current correctness check is incomplete, rejecting many valid definitions, in particular those involving mutually-recursive datatype declarations. In this paper, we explain the notion of *separability* as a semantic for the unboxing criterion, and propose a set of inference rules to check separability. From these inference rules, we derive a new implementation of the unboxing check that properly supports mutually-recursive definitions.

1 Introduction

Version 4.04 of the OCaml programming language, released in November 2016, introduced the possibility to *unbox* single-constructor variants and single-immutable-field records. In other words, a value inhabiting such a datatype is exactly represented at runtime as the value that it contains, rather than as a pointer to a *block* containing a tag for the constructor and the contained value. The removal of this indirection is called *constructor unboxing*, or *unboxing* for short, and it allows for a slight improvement in speed and memory usage.

In the current version of OCaml, unboxing must be explicitly requested with `[@@unboxed]` as shown in the following example:

```
type uid = UId of int [@@unboxed]
```

One of the main interests of unboxing is that it allows the incorporation of semantic type distinctions without losing runtime efficiency – the mythical zero-cost abstraction. For example, the elements of `uid` are distinct from those of `int` for the type checker, but they have the same runtime representation. Unboxing resolves a tension between software engineering and performance.

Unboxing becomes even more interesting when it is combined with advanced features such as existential types, with GADTs, or higher-rank polymorphism, with polymorphic record fields, which are otherwise only accessible in boxed form.

```
type 'a data = { name : string ; data : 'a }
type any_data = Any_data : 'a data -> any_data [@@unboxed]

type proj = { proj : 'a. 'a -> 'a -> 'a } [@@unboxed]
```

1.1 Unboxing and dynamic floating-point checks

OCaml uses a uniform memory representation, one machine word for all values. Multi-word data such as floating-point numbers, records or arrays are represented by a word-sized pointer to a block on the heap.

For local computations involving floating-point numbers, the compiler tries to optimize by storing short-lived floating-point values directly, without the pointer indirection; this is called *floating-point unboxing*, a different form of unboxing optimization. The boxing indirection needs to be kept for values passed across function boundaries, or passed through data structures that expect generic OCaml values.

To avoid indirection costs on typical numeric computations, a specific representation exists for floating-point arrays, in which floating-point numbers are stored unboxed, as two consecutive words in memory. The compiler uses this optimized representation when an array is statically known to have type `float array`. It also performs a dynamic optimization when creating a new non-empty array: it checks whether its first element is a (boxed) floating-point value, and in that case uses the special floating-point array representation. As a consequence, all writes to such an array need to be unboxed first, and reads produce data in already-unboxed form, which meshes well with the local floating-point unboxing optimizations.

However, this optimization crucially relies on an underlying assumption: the inhabitants, the values of any given type are either all boxed floating-point values, or none of them are. We don't know of a standard name for this property, so we call it *separability*.

Indeed, if there were non-separable types, containing both floating-point and non-floating-point values, then this optimization would be unsound. This is demonstrated by the following example, which relies on the unsafe/forbidden casting function `Obj.magic`.

```
let despicable : float array = [| 0.0 ; Obj.magic 42 |]
(* Produces: "segmentation fault (core dumped)" *)
```

The array of floating point numbers that is constructed here is stored using the optimized representation. As a consequence, its elements must be unboxed prior to being inserted into the actual array. Here, the segmentation fault is precisely triggered when attempting to unbox the value `42`, which is not stored in a block but as an immediate memory word: dereferencing it as a float pointer accesses forbidden memory.

Although the above example requires unsafe features, a similar situation also arises when using a single-constructor GADT whose parameter is existentially quantified, and can hence contain either floating-point or non-floating-point values.

```
type any = Any : 'a -> any
```

The above datatype is the GADT formulation of the existential type $\exists\alpha.\,\alpha$: it can contain any OCaml value. However, those values are "boxed" under the `Any` constructor. In particular, the value `Any 0.0` is *not* directly represented as a floating-point value, but as pointer to a block stored on the heap that is tagged with the `Any` constructor, following by (the OCaml representation of) the floating-point number.

If the `any` datatype were allowed to be unboxed, we would have constructed a type breaking the separability assumption. The above example of segmentation fault could then be reproduced by a well-typed program as follows.

```
type any = Any : 'a -> any [@@unboxed]
(* The above type is rightfully rejected by OCaml, it cannot be unboxed. *)

let array = [| Any 0.0 ; Any 42 |]
```

The current implementation correctly rejects this `any` datatype, but it also rejects valid unboxed definitions that would not introduce non-separable types, when mutually-recursive datatypes are involved. The following real-world example of an interesting definition that is wrongly rejected was given by Markus Mottl.[1]

```
type (_, _) tree =
  | Root  : { mutable value : 'a; mutable rank : int } -> ('a, [`root ]) tree
  | Inner : { mutable parent : 'a node }                -> ('a, [`inner]) tree

and _ node = Node : ('a, _) tree -> 'a node [@@ocaml.unboxed]
(* The above type is incorrectly rejected by OCaml, it could be unboxed. *)
```

Yet another example of such wrongful rejection was encountered by the second author, in the context of the Bindlib library (Lepigre and Raffalli, 2018). It resembles the definitions of `any_type` and `'a data` that were given earlier, but it is rejected as the two datatypes are defined mutually-recursively (for more details, see Lepigre and Raffalli, 2018, Section 3.1).

1.2 The existing check

The existing implementation of the compiler check for unboxing was implemented by Damien Doligez in 2016, when constructor unboxing was introduced. It proceeds by inspecting the parameter of every unboxed GADT constructor. If it is an existential variable `'a`, the definition is rejected. If it is a type expression whose value representation is known, such as a function `foo -> bar` or a product `foo * bar`, then it is separable and the definition is accepted. The difficult case arises when the parameter is of the form `(foo, bar, ...) t`. For example, if the parameter has type `'a t`, where `'a` is an existential variable, then this definition must be rejected if `t` is defined as `type 'a t = 'a`, but it can be accepted if it is defined as `type 'a t = int * 'a`, for example.

In the current check, `(foo, bar, ...) t` is expanded according to the definition of `t`, and its expansion is checked recursively. In the case where `t` is an abstract datatype, the check correctly fails as soon as one of the parameter is an existential. If `t` is part of the same block of mutually-recursive definitions, its definition may not be known yet, and the check fails although it could have succeeded had the definition been known. Finally, it is worth noting that because of recursive datatypes, the expansion process may not terminate. As a consequence, there is a hard limit on the number of expansions.[2]

1.3 Our approach: inference rules for separability

We propose to replace the current check using a "type system" for separability. In other words, we introduce inference rules to approximate the semantic notion of separability: a type is separable if the values it contains are all floating-point numbers, or if none of them are. This

[1] https://github.com/ocaml/ocaml/pull/606#issuecomment-248656482
[2] This limit was originally set too high, and it had to be reduced to avoid type-checking slowdowns.

approach is very similar to how the variance of datatype parameters is handled in languages with subtyping, including OCaml, both in theory and in practice. In the case of variance, the "types" are annotations such as `covariant` or `contravariant`, and the "terms" that are being checked are types and datatype definitions.

Here, the "types" are *separability modes* indicating whether the corresponding type parameters need to be separable for the whole datatype to be separable. Modes include `Sep` (separable), the mode of types or parameters that must be separable, and `Ind` (indifferent), the mode of types or parameters on which no separability constraint is imposed.[3] For example, in the case of `type 'a t = 'a * int`, the parameter `'a` has mode `Ind` since the values it contains are all pairs, independently of the type used to instantiate `'a`. For `type ('a, 'b) second = 'b`, the parameter `'a` has mode `Ind`, but the parameter `'b` has mode `Sep`. Indeed, if `'b` is not instantiated with a separable type, then the whole definition cannot be separable either. The separability behavior of a parametrized datatype is characterized by a *mode signature*, which gives a choice of mode for the type parameters that guarantees that the whole datatype will be separable. For instance, the previous two examples would have mode signatures `('a:Ind) t` and `('a:Ind, 'b:Sep) second`.

If a type that is being checked is defined in terms of a type constructor `(foo, bar, ...) t`, and if the mode signature of `t` is known, then its definition does not need to be unfolded as in the legacy implementation. Indeed, it is enough to simply check the parameter instances (`foo`, `bar`, ...) against the modes of the signature. For example, if we encounter `(foo, bar) second`, then we only need to check that `bar` is separable.

In the case of a block of mutually-recursive datatype definitions, a mode signature must be constructed for each definition of the block. However, this cannot be done separately for each definition due to dependencies. As a consequence, we proceed by computing a fixpoint, as for variance: we iteratively refine an approximation of the mode signatures for the block, updating when encountering conflicts, until a mode signature that requires no update is found: it is a valid signature for the block. The number of possible modes assignments for each parameter is finite, so this fixpoint computation always terminates. Note that our inference rules do not talk about the fixpoint computation, or about algorithmic aspects in general. It is a simpler, declarative specification for the correctness of mode assignments, from which the checking algorithm can be derived. It can also be used to reason about the semantic correctness of our check.

1.4 Contributions

We claim the following three contributions:

- A clear explanation of separability. The notion of separability evidently existed in Damien Doligez's mind when unboxed datatypes where implemented in 2016, but we had to rediscover it to understand the check, and we took this opportunity to document separability and to give it a precise semantic definition.
- A set of inference rules for separability of type expressions and datatype definitions.
- An implementation of a separability check derived from these inference rules, within the OCaml compiler. It is compatible with mutually-recursive definitions, which were previously always rejected.

A preliminary version of this work was produced during an internship of the first author (Colin, 2018). We give here a full treatment of GADTs, while the internship report used first-class existential types, without equations. We also moved from a prototype implementation, separate

[3] There is actually a third mode `Deepsep` (deeply separable) that will be explained in the next section.

from the OCaml type-checker and defined on simpler data-structures, to a production-ready implementation of the separability check in (an experimental fork of) the OCaml type-checker.

2 Type language and separability modes

As mentioned in the introduction, our approach to unboxability checking relies on the notion of *separability* of a type, which was already introduced in an intuitive way.

Definition 1 (Separability)**.** *A type is said to be separable if and only if its inhabitant contain either only floating-point values, or only non-floating point values.*

Intuitively, our final goal is to make sure that all type definitions have a separable body. That is, for all definition `type ('a, 'b, ...) t = <type_body>`, we want to check that the type `<type_body>` is separable under some assumptions on the separability of parameters. Before going into formal definitions, two potential sources of difficulties must be discussed: GADTs, which are the only possible source of non-separability, and the (related) typing constraints, which can be used to extract possibly non-separable subcomponents of a type.

2.1 GADTs using type equalities and existential quantifiers

Generalized/guarded algebraic datatypes, GADTs for short extend the usual variant datatypes using a slightly different syntax. They are parametrized datatypes `(_, _, ...) t` in which the typing constraints on the parameters may vary depending on the variant constructor. Moreover, existentially quantified type variables may appear in GADT constructors.[4] Examples of GADTs illustrating these features are given below.

```
type _ data =
  | Char : char -> char data
  | Bool : bool -> bool data

type any_function = Fun : ('a -> 'b) -> any_function

type _ first = First : 'c -> ('c * 'd) first
```

As it turns out, GADTs can be decomposed into more primitive components: non-GADT algebraic datatypes, *type equalities* and *existential types*. For instance, the above examples can be encoded as follows, using an imaginary extension of the OCaml syntax explained below.[5]

```
type 'a data =
  | Char of char with ('a = char)
  | Bool of bool with ('a = bool)

type any_function = Fun of exists 'a 'b. 'a -> 'b
```

[4] See the OCaml manual (https://caml.inria.fr/pub/docs/manual-ocaml-4.07/extn.html#sec252) for a more thorough introduction to GADTs.

We are intentionally unclear about whether the "G" means "generalized" or "guarded", because both words have been used in the literature. In the present article, "guarded" makes more sense as we concentrate on the pains introduced by equality constraints, or equality *guards*. "Generalized" is also a rather empty name, as there are many other ways to generalize algebraic datatypes.

[5] The translation can be defined in a systematic way, see for example Simonet and Pottier (2007).

```
type 'a first = First of exists 'b 'c. 'b with ('a = 'b * 'c)
```

In the above, we use the new syntax `exists 'a. <type_expr>` for first-class existential quantification over one or several type variables, and `<type_expr> with ('a = <type_expr>)` for guarding a type expression with an equality constraint. In our formal description of the separability check, we will use the syntax $\exists\alpha.\tau$ for existentials, and the syntax $\tau\restriction(\alpha = \kappa)$ for equality guards, where α is a type variable, and τ and κ are type expressions.

The semantic intuition behind these two type-formers is the following. An existential type $\exists\alpha.\tau$ can be understood as the union over all types κ of $\tau[\alpha := \kappa]$. For example, the values of the type `exists 'a. 'a -> 'a` contains the values of `int -> int`, but also the values of `bool -> bool` and the values of `(char -> bool) -> (char -> bool)`. [6] An equality guard $\tau\restriction(\alpha = \kappa)$ exactly corresponds to τ in the case where the constraint $(\alpha = \kappa)$ is satisfied, and it is empty otherwise. Note that guards whose left-hand-side is a type variable suffice to express GADTs: all guards equate a type parameter to its instance in the "return type" of the variant constructor.

2.2 Equality guards and deep separability

In the introduction, the parametrized type constructors `'a t` that were considered either had the mode signature `('a : Ind) t`, meaning that `'a t` is always separable no matter what `'a` is, or they had mode signature `('a : Sep) t`, meaning that `'a t` is separable only if `'a` is itself separable. However, these two modes are not always sufficient due to equality guards.

The equality guards used in GADTs[7] introduce the ability for parametrized types to "peek" into the definition of their parameters in ways that affect our separability check. Consider, for example, the unboxed version of the `_ first` datatype, which is accepted by the current constructor unboxing check.

```
type _ first =
  | First : 'b -> ('b * 'c) first [@@unboxed]
```

Using this definition, `('b * int) first` has the same memory representation as `'b`, so it is separable if and only if `'b` is separable. In other words, the separability of type `foo first` does not on the separability of type `foo`, but rather on the separability of a *sub-component*, in the sense of syntactic inclusion, of the type `foo`.

To account for this situation, we introduce a third separability mode `Deepsep` (deeply separable). For a closed type expression (a type expression with no free type variables) to be `Deepsep`, all its sub-components, including the type expression itself, must be separable.

2.3 Formal syntax of types

We define the syntax of the type expressions and datatypes that we consider in the top and middle parts of Figure 1. It is a representative subset of the OCaml grammar of types,[8] with first-class existential types $\exists\alpha.\tau$ and type restrictions $\tau\restriction(\alpha = \kappa)$ to represent GADTs with

[6] The interpretation of existential quantifiers as unions (and dually, of universal quantifiers as intersections) is very common in realizability semantics, for example.

[7] Constrained datatype definitions such as `type 'a t = 'b constraint 'a = 'b * int` may also involve equality guards, but we chose to ignore this feature here since it poses exactly the same problem.

[8] For instance, we omit object types and polymorphic variants, but they could be handled just like products.

Syntax of type expressions:

$$
\begin{array}{rll}
\tau, \kappa ::= & \alpha, \beta & \text{type variable} \\
& \mid \texttt{float} \mid \texttt{int} \mid \texttt{bool} & \text{builtin types} \\
& \mid t(\tau_1, \ldots, \tau_n) & \text{(parametrized) type constructor} \\
& \mid \tau \to \kappa & \text{function type} \\
& \mid \tau_1 \times \cdots \times \tau_n & \text{product/record type} \\
& \mid \forall \alpha.\tau & \text{polymorphic type} \\
& \mid \exists \alpha.\tau & \text{existential type} \\
& \mid \tau \restriction (\alpha = \kappa) & \text{equality guard}
\end{array}
$$

Syntax of datatypes:

$$
\begin{array}{rll}
A, B ::= & C_1 \text{ of } \tau_1 \mid \ldots \mid C_n \text{ of } \tau_n & \text{boxed variant} \\
& \mid C \text{ of } \tau \ [@@\texttt{unboxed}] & \text{unboxed variant} \\
& \mid \{\texttt{mutable? } l_1 : \tau_1 \ ; \ \ldots \ ; \ \texttt{mutable? } l_n : \tau_n\} & \text{boxed record} \\
& \mid \{l : \tau\} \ [@@\texttt{unboxed}] & \text{unboxed record} \\
& \mid \tau & \text{type synonym}
\end{array}
$$

Sub-component relation on type expressions:

$$
\frac{}{\tau \lhd \tau} \qquad \frac{\tau_1 \lhd \tau_2 \qquad \tau_2 \lhd \tau_3}{\tau_1 \lhd \tau_3}
$$

$$
\frac{i \in [1; n]}{\tau_i \lhd t(\tau_1, \ldots, \tau_n)} \qquad \frac{1 \in \{1, 2\}}{\tau_i \lhd \tau_1 \to \tau_2} \qquad \frac{i \in [1; n]}{\tau_i \lhd \tau_1 \times \cdots \times \tau_n}
$$

$$
\frac{\tau \lhd \kappa \qquad \alpha \notin \tau}{\tau \lhd \forall \alpha.\kappa} \qquad \frac{\tau \lhd \kappa \qquad \alpha \notin \tau}{\tau \lhd \exists \alpha.\kappa} \qquad \frac{i \in \{1, 2\}}{\tau_i \lhd (\tau_2 \restriction (\alpha = \tau_1))}
$$

Figure 1: Syntax of type expressions, datatypes, and sub-component relation.

finer-grained rules, as explained in Section 2.1. We also include first-class universal types $\forall \alpha.\tau$, although they are only allowed in record or method fields in OCaml.

To define deep separability, we first need to define the syntactic sub-components of a type (see Section 2.2). We define a sub-component relation $\tau \lhd \kappa$ ("τ is a sub-component of κ") at the bottom of Figure 1. The first two rules make the relation reflexive and transitive, and the others give the immediate sub-components for each type-former.

The definition of the sub-component relation is careful to preserve the scoping of variables. For example, $\tau \lhd \forall \alpha.\kappa$ holds only if α does not occur free in τ, which we write $\alpha \notin \tau$. In particular, whenever $\tau \lhd \kappa$ holds, the free type variables of τ are included in those of κ; the sub-components of a closed type expression are also closed types.

2.4 Separability modes

Separability modes m, n, their order structure $m < n$ and mode composition $m \circ n$ are defined in Figure 2. Modes are totally ordered from less to more demanding. We may use derived notations such as $m \leqslant n$, $\min(m, n)$ or $\max(m, n)$ for the non-strict ordering, the minimum or

Separability modes (or simply, modes):

$$m, n ::= \texttt{Ind} \mid \texttt{Sep} \mid \texttt{Deepsep}$$

`Ind`	indifferent
`Sep`	separable
`Deepsep`	deeply separable

Mode composition $m \circ n$:

$$\begin{aligned} \texttt{Ind} \circ m &:= \texttt{Ind} \\ \texttt{Sep} \circ m &:= m \\ \texttt{Deepsep} \circ m &:= \texttt{Deepsep} \end{aligned}$$

Order structure:
$\texttt{Ind} < \texttt{Sep} < \texttt{Deepsep}$

Figure 2: Separability modes and mode operations.

the maximum of two modes.

A mode m expresses a requirement on a type expression, which comes from its context: for the whole expression to be valid, some of its sub-components must have the separability property m. The operation $m \circ n$ expresses composition of those requirements in our inference rules (given in the next section). For example, if $t(\alpha)$ requires its parameter α to have mode m for the whole expression to be separable, and $u(\beta)$ requires its parameter β to have mode n for the whole expression to be separable, then $t(u(\beta))$ is separable when β has mode $m \circ n$.

2.5 Contexts and mode signatures

Our separability judgments in Section 3 make use of contexts Γ, representing separability assumptions on the type variables $(\alpha, \beta \dots)$ that are in scope. Moreover, we also rely on mode signatures Σ, representing the separability requirements on the datatype constructors $t(\overline{\alpha})$ that are available in the scope. Contexts and mode signatures are defined in Figure 3.

$$\Gamma ::= \varnothing \mid \Gamma, \alpha : m \qquad \Sigma ::= \varnothing \mid \Sigma, t(\overline{\alpha : m})$$

$$\frac{\Gamma \leqslant \Gamma' \qquad m \leqslant m'}{\Gamma, \alpha : m \leqslant \Gamma', \alpha : m'} \qquad \frac{\Sigma \leqslant \Sigma' \qquad \overline{\alpha} = \overline{\alpha'} \qquad \overline{m \geqslant m'}}{\Sigma, t(\overline{\alpha : m}) \leqslant \Sigma', t(\overline{\alpha' : m'})}$$

Figure 3: Contexts, mode signatures, and their order

This figure also defines the extension of the order on modes to an order on contexts and on mode signatures. The order on contexts is just a point-wise extension of the mode order, imposing that two comparable contexts have the same type variables. The order on signatures imposes the same datatype constructors on both sides, but requires their parameter modes to be pointwise comparable in the opposite order, $(\geqslant)$ rather than $(\leqslant)$: parameters are in contravariant position. For example, we have $t(\alpha : \texttt{Sep}, \beta : \texttt{Sep}) \leqslant t(\alpha : \texttt{Ind}, \beta : \texttt{Ind})$.

3 Separability inference

The formal deduction rules that we will use to assess the separability of datatype definitions are defined in Figure 4. Type expressions are checked by judgments of the form $\Sigma; \Gamma \vdash \tau : m$, which can intuitively be read in either direction, from Γ to m or conversely:

- If the type variables respect the modes in Γ, then the type expression τ has the mode m.

- For the type expression τ to be safe at mode m, then its type variables need to have at least the modes given in Γ.

Inference rules at type-expression level:

$$\frac{(\alpha : m) \in \Gamma}{\Sigma; \Gamma \vdash \alpha : m} \qquad \frac{t(\alpha_1 : m_1, \ldots, \alpha_n : m_n) \in \Sigma \qquad (\Sigma; \Gamma \vdash \tau_i : m \circ m_i)_{1 \leqslant i \leqslant n}}{\Sigma; \Gamma \vdash t(\tau_1, \ldots, \tau_n) : m}$$

$$\frac{\Sigma; \Gamma \vdash \tau : m \circ \texttt{Ind} \qquad \Sigma; \Gamma \vdash \kappa : m \circ \texttt{Ind}}{\Sigma; \Gamma \vdash \tau \to \kappa : m} \qquad \frac{(\Sigma; \Gamma \vdash \tau_i : m \circ \texttt{Ind})_{1 \leqslant i \leqslant n}}{\Sigma; \Gamma \vdash \tau_1 \times \cdots \times \tau_n : m}$$

$$\frac{\Sigma; \Gamma, \alpha : n \vdash \tau : m}{\Sigma; \Gamma \vdash \forall \alpha . \tau : m} \qquad \frac{\Sigma; \Gamma, \alpha : \texttt{Ind} \vdash \tau : m}{\Sigma; \Gamma \vdash \exists \alpha . \tau : m} \qquad \frac{\Sigma; \Gamma \vdash \tau : m \qquad m \geqslant n}{\Sigma; \Gamma \vdash \tau : n}$$

$$\frac{\forall \Gamma' \geqslant \Gamma, \quad \Sigma; \Gamma' \vdash (\kappa_1 = \kappa_2) \implies \Sigma; \Gamma' \vdash \tau : m}{\Sigma; \Gamma \vdash \tau \restriction (\kappa_1 = \kappa_2) : m} \qquad \frac{\forall m, \quad \Sigma; \Gamma \vdash \tau_1 : m \iff \Sigma; \Gamma \vdash \tau_2 : m}{\Sigma; \Gamma \vdash (\tau_1 = \tau_2)}$$

Inference rules at datatype level:

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{decl}} C_1 \text{ of } \tau_1 \mid \ldots \mid C_n \text{ of } \tau_n} \qquad \frac{\Sigma; \Gamma \vdash \tau : \texttt{Sep}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} C \text{ of } \tau \ [@@\texttt{unboxed}]}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{decl}} \{l_1 : \tau_1 \ ; \ \ldots \ ; \ l_n : \tau_n\}} \qquad \frac{\Sigma; \Gamma \vdash \tau : \texttt{Sep}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} \{l : \tau\} \ [@@\texttt{unboxed}]} \qquad \frac{\Sigma; \Gamma \vdash \tau : \texttt{Sep}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} \tau}$$

Inference rule at the datatype declaration block level:

$$\frac{\exists (\overline{m_i})_{1 \leqslant i \leqslant n}, \ \Sigma_{\mathsf{block}} := t_1(\overline{\alpha_1} : \overline{m_1}), \ldots, t_n(\overline{\alpha_n} : \overline{m_n}) \qquad (\Sigma_{\mathsf{env}}, \Sigma_{\mathsf{block}}; \overline{\alpha_i} : \overline{m_i} \vdash_{\mathsf{decl}} A_i)_{1 \leqslant i \leqslant n}}{\Sigma_{\mathsf{env}} \vdash (t_i(\overline{\alpha_i}) := A_i)_{1 \leqslant i \leqslant n} \dashv \Sigma_{\mathsf{block}}}$$

Figure 4: Inference rules for separability.

Type expressions. The rule for parametrized datatypes $t(\tau_1, \ldots, \tau_m)$ uses mode composition; for example, if t is a one-argument type constructor with signature $t(\alpha : n)$ and we want $t(\tau)$ to have mode m, then τ needs to have the mode $m \circ n$.

The rules for concrete datatypes (functions and products, but also most other OCaml type-formers if we were to add them) use $m \circ \texttt{Ind}$ on their arguments. If m is `Sep` or `Ind`, then $m \circ \texttt{Ind}$ is `Ind`, which corresponds to not requiring anything of the sub-components: the values at type $\tau_1 \to \tau_2$ are all functions, never floats, regardless of τ_i. If m is `Deepsep`, then we do need to check the sub-components, and indeed $\texttt{Deepsep} \circ \texttt{Ind}$ is `Deepsep`.

A value is at the universal type $\forall \alpha . \tau$ only if it belongs to all the $\tau[\kappa/\alpha]$ for all κ. In particular, it is enough to prove the separability at just one of these $\tau[\kappa/\alpha]$, the universal type has even less values, so we can assume the arbitrary mode n of this particular κ by adding $\alpha : n$ in the context. Conversely, $\exists \alpha . \tau$ is inhabited by all the $\tau[\kappa/\alpha]$, so τ has to have the desired mode for all possible modes of κ. Instead of requiring the premise to hold for all possible modes n, we equivalently ask for the most demanding mode `Ind`.

The conversion rule allows to forget some information about a type $\tau : m$ by exporting it at a smaller mode $n \leqslant m$. For example, all `Sep` types are also `Ind` types.

Equality constraints. The rule for equality constraints is the most complex rule in the system. A first remark is that $\kappa\restriction(\tau_1 = \tau_2)$ always has less elements than κ: it has the same elements when the equality holds, or no elements otherwise. In particular, if $\kappa : m$ holds in the current context Γ, then $\kappa\restriction(\tau_1 = \tau_2) : m$ should also hold in Γ.

When we see an equality constraint, we gain more information, which should allow us to mode-check more types. The way our rules represent this information gain is by moving from the current mode context Γ to a stronger mode context Γ'. More precisely, to check $\kappa\restriction(\tau_1 = \tau_2) : m$ in Γ, the rule asks to check $\kappa : m$ in *any* context $\Gamma' \geqslant \Gamma$ that is consistent with the assumption $(\tau_1 = \tau_2)$. This corresponds to the $\Gamma' \vdash (\tau_1 = \tau_2)$ hypothesis, which will be explained shortly. For example, if Γ is $\alpha : \texttt{Ind}, \beta : \texttt{Sep}$, and we observe the equality $(\alpha = \beta)$, then in particular we know that $\alpha : \texttt{Sep}$ also holds: if the two types are equal, they must have the same mode. Our rule will type-check the body type κ in stronger contexts Γ' with $\alpha : \texttt{Sep}, \beta : \texttt{Sep}$.

A context Γ is valid for a type equality, written $\Gamma \vdash (\tau_1 = \tau_2)$, if the two types τ_1 and τ_2 have exactly the same mode in Γ. Remark that it is not enough to ask that, for a given mode m, both types have mode m; for example, all types trivially have mode `Ind`. Instead, we ask that for *any* mode m, either τ_1 and τ_2 have mode m, or neither of them have it. This is equivalent to requiring τ_1 and τ_2 to have the same "best", maximal mode.

In our example where Γ is $\alpha : \texttt{Ind}, \beta : \texttt{Sep}$, note that we do not have $\Gamma \vdash (\alpha = \beta)$: β has mode `Sep` but α does not. The modes $\Gamma' \geqslant \Gamma$ that satisfy $\Gamma' \vdash (\alpha = \beta)$ are $\Gamma_{\texttt{Sep}} := \alpha : \texttt{Sep}, \beta : \texttt{Sep}$, and $\Gamma_{\texttt{Deepsep}} := \alpha : \texttt{Deepsep}, \beta : \texttt{Deepsep}$. To check that $\kappa\restriction(\alpha = \beta) : m$ holds in Γ, our rule asks us to check that $\kappa : m$ holds in both $\Gamma_{\texttt{Sep}}$ and $\Gamma_{\texttt{Deepsep}}$. But in fact we, the implementers of the checking algorithm, know that making stronger assumptions in the context makes more mode-checks pass, so it suffices to check in context $\Gamma_{\texttt{Sep}}$.[9]

Finally, it is interesting to consider the derivation of the following judgment, which represents in our system the key ingredient of the `_ first` GADT that led to the introduction of `Deepsep` in Section 2.2.

$$\alpha : m \vdash \exists \beta\gamma.\, \beta\restriction(\alpha = \beta \times \gamma) : \texttt{Sep}$$

We expect to accept this type declaration only in the case where $\alpha : \texttt{Deepsep}$, this assumption guaranteeing that β will be separable. Opening the existentials puts β, γ in the context at `Ind`:

$$\alpha : m, \beta : \texttt{Ind}, \gamma : \texttt{Ind} \vdash \beta\restriction(\alpha = \beta \times \gamma) : \texttt{Sep}$$

Let us now reason by case analysis on m to show that only $m = \texttt{Deepsep}$ has a valid derivation of this judgment.

- If m is `Deepsep`, then any $\Gamma' \geqslant \Gamma$ with $\Gamma' \vdash (\alpha = \beta \times \gamma)$ has $\beta : \texttt{Deepsep}, \gamma : \texttt{Deepsep}$ since otherwise $\beta \times \gamma : \texttt{Deepsep}$ cannot hold. In particular, $\Gamma' \vdash \beta : \texttt{Sep}$ holds as expected, so the judgment is derivable.
- If m is `Sep` or `Ind`, then $\Gamma' := \alpha : \texttt{Sep}, \beta : \texttt{Ind}, \gamma : \texttt{Ind}$ satisfies $\Gamma' \geqslant \Gamma$ and $\Gamma' \vdash (\alpha = \beta \times \gamma)$, but we do not have $\Gamma' \vdash \beta : \texttt{Sep}$: the judgment is not derivable.

Datatypes. The judgment for datatypes is simple: a datatype is accepted if its set of values is separable. Boxed variant or record definitions are always separable, so no premises are required. Unboxed variants/records with parameter type τ, or type synonyms for τ require $\tau : \texttt{Sep}$.

Definition blocks. The judgment for definition blocks has an input signature, $\Sigma_{\texttt{env}}$ in the rule, which lists assumptions that we make on the datatype constructors provided by the typing

[9] We could formulate the rule to only check the unique minimal context Γ', but Fact 2 in Section 4 suggests that such a unique context may not always exist.

environment, and an output signature, Σ_{block} in the rule, which is a valid signature for the current block of mutually-recursive definitions. A definition block is valid if each datatype definition it contains is valid. Note that each definition is checked with the full Σ_{block} in the signature context: all mutually-recursive datatype constructors are available in scope.

Meta-theory. The following two lemmas can be easily proved by structural induction on typing derivations.

Lemma 1 (Cut elimination)**.** *A separability derivation can always be rewritten so that all occurrences of the conversion rule only have axiom rules as their premises or, equivalently, using the following more primitive rule.*

$$\frac{(\alpha : m) \in \Gamma \qquad m \geqslant n}{\Sigma; \Gamma \vdash \alpha : n}$$

Lemma 2 (Monotonicity)**.** *The following rule is admissible.*

$$\frac{\Sigma \leqslant \Sigma' \qquad \Gamma \leqslant \Gamma' \qquad \Sigma; \Gamma \vdash \tau : m \qquad m \geqslant m'}{\Sigma'; \Gamma' \vdash \tau : m'}$$

4 Semantics

Due to space limitations, we moved our presentation of separability semantics to Appendix A. It contains a precise semantic characterization of separability judgments in the flavor of set-theoretic or realizability models, and a discussion of soundness, principality and completeness. Many results are left as conjectures – we explain why some of them are surprisingly delicate.

5 Integration into OCaml

We are now going to highlight some important points of our implementation in the OCaml type-checker. The corresponding code has been proposed for integration through a GitHub pull request, that is visible at the following URL.

https://github.com/ocaml/ocaml/pull/2188

Our implementation is derived from the type system given in Section 3 by inferring mode signatures for type definitions. It only assigns mode signatures that can be justified by our syntactic type system. For example, if `('a : Sep, 'b : Ind) t` is assigned to a type definition `type ('a, 'b) t = A`, then $\Sigma_{\mathsf{env}} \vdash t(\alpha, \beta) := A \dashv t(\alpha : \texttt{Sep}, \beta : \texttt{Ind})$ must be derivable using the inference rules of Figure 4.

5.1 Inferring block signatures with a fixpoint

Our main checking function constructs a block signature Σ_{block} given an environment Σ_{env} and a block of type definitions $(t_i(\overline{\alpha_i}) := A_i)_{1 \leqslant i \leqslant n}$:

```
val check : Env.t -> type_definition ConstrMap.t -> mode_signature ConstrMap.t
```

In this signature, `type_definition ConstrMap.t` maps datatype constructors $t_i(\overline{\alpha})$ to datatype definitions A_i, and `mode_signature ConstrMap.t` them to a mode signature $t_i(\overline{\alpha : m})$.

It is not possible to directly compute mode signatures for mode definitions, due to recursive types: we need to know the mode signature of the type constructors in order to assign them a mode signature. This circularity is solved by using a fixpoint computation: we iteratively refine an approximation of the mode signatures.

The fixpoint computation starts with the most permissive mode signature, which only requires mode `Ind` for the variables of every type constructor of the block. At each iteration, the separability of every type of the block is checked against the current approximation of the mode signatures, accumulating more precise constraints whenever required. If the mode signatures coming from these constraints are more demanding than those of the current approximation, we define them as the new approximation and continue. Otherwise, we have found a mode signature that validates the judgment – it is sound. In fact, it is the most permissive mode signature that we can find by iteration in this way.

5.2 Management of GADTs

As explained in Section 2.1, our type system for separability does not directly account for GADTs: they are encoded using existential quantifiers and equality guards. Our implementation handles only GADTs, which correspond to a very specific mode of use of existentials and guards within type declarations.

Consider, for example, `type 'a fun = Fun : ('b -> 'c) -> ('b -> 'c) fun`. The return type `('b -> 'c) fun` determines the equality guard ($'\mathtt{a} = '\mathtt{b} \to '\mathtt{c}$), and the existential types are the free type variables $'\mathtt{b}$, $'\mathtt{c}$ of the declaration. In general, unboxable GADT types contain existential quantifiers only at the toplevel, immediately followed by one equation for each type parameter, with finally a concrete parameter type τ.

The first thing to note is that all existential quantifiers occurs at the top level: they exactly correspond to the free variables of the parameter type. We can infer a mode signature for the parameter type τ and, following our rule for existentials, check that the modes inferred for the free existential variables in τ are `Ind` and fail otherwise. Finally, the mode signature for the definition of the GADT is obtained by removing the existential variables from the inferred mode signature.

The delicate matter with in handling of GADTs is unsurprisingly the management of equality guards. Recall that an unboxed GADT $\mathtt{type}\ t(\overline{\alpha}) = \mathtt{K} : \tau \to t(\overline{\kappa})$ is encoded as $\exists\overline{\beta}.\tau\restriction(\overline{\alpha = \kappa})$, where the $\overline{\beta}$ are the type variables free in $\tau, \overline{\kappa}$ and the $\overline{\kappa}$ do not contain any α_i. The idea of the implementation is to first infer a mode context Γ such that $\Sigma; \Gamma \vdash \tau : \mathtt{Sep}$, assigning modes to the variables in $\overline{\alpha}$ and $\overline{\beta}$. Equations are then discharged one by one, refining Γ in the process, before the existential variables $\overline{\beta}$ are checked to be `Ind` and eliminated to create the final context for the GADT parameters only – the mode signature. Every equation, taken in any order, is managed according to the following three cases:

- An equation of the form $(\alpha_i = \beta_j)$ leads to Γ being updated with $\{\alpha_i \mapsto \Gamma(\beta_j), \beta_j \mapsto \mathtt{Ind}\}$.
- An equation of the form $(\alpha_i = \kappa)$ with $\Gamma \cap FV(\kappa)$ only containing `Ind` leads to the equation being discarded without any update to Γ.
- Any other equation (i.e., equations $(\alpha_i = \kappa)$ with $\Gamma \cap FV(\kappa)$ not only containing `Ind`) leads to Γ being updated with $\{\alpha_i \mapsto \mathtt{Deepsep}, FV(\kappa) \mapsto \mathtt{Ind}\}$.

These transformations respect the following invariant: if the equation $(\alpha = \kappa)$ updates Γ into Γ_0, then any $\Gamma' \geqslant \Gamma_0$ that satisfies $\Gamma' \vdash (\alpha = \kappa)$ is above the original Γ ($\Gamma' \geqslant \Gamma$). In other words, strengthening the resulting context Γ_0 with the equation we just handled would give a context as permissive or more than the original context Γ.

5.3 Cyclic types

The OCaml type system accepts equi-recursive types. By default, any cycle in types must go through a polymorphic variant or object type constructor, but the `-rectypes` option generalizes this to any ground type constructor. A type such as `('a -> 'b) as 'b`, for example, gives a cyclic representation to the infinite type `'a -> 'a -> 'a -> ...`, and must be supported by our implementation.

To support cyclic types, we extend our checking rules to support a form of coinduction: each time we reduce a judgment to simpler premises (for example from $\Gamma \vdash \tau_1 \to \tau_2 : m$ to $\Gamma \vdash \tau_i : m \circ \texttt{Ind}$), we record in the premises that we have previously encountered the judgment $\tau_1 \to \tau_2 : m$. If, later, we encounter the same judgment to prove, we terminate immediately with a success.

With this rule, it is possible to prove both

$$\alpha : \texttt{Ind} \vdash ((\alpha \to \beta)\ \texttt{as}\ \beta) : \texttt{Sep} \qquad\qquad \alpha : \texttt{Deepsep} \vdash ((\alpha \to \beta)\ \texttt{as}\ \beta) : \texttt{Deepsep}$$

Informally, the reason why recursive occurrences of the same judgment can be considered an immediate success is that we "made progress" between the first occurrence of the cyclic type β and its second occurrence in $\alpha \to \beta$: the second occurrence is "guarded" under a value constructor, and the set of values of β we are classifying as separable, or deeply separable, is not the one we started from – that would be an invalid cyclic reasoning – but a copy of it occurring deeper in the type structure.

However, some cyclic types such as $t(\alpha)\ \texttt{as}\ \alpha$ have a less clear status, as the recursive occurrence is not guarded under a computational type constructor (arrow, product...), but under an abbreviation. Is this type well-defined if, for example, $t(\alpha)$ is defined as $t(\alpha) := \alpha$? OCaml rejects some of these dubious-looking circular types, but instead of trusting our fate to the rest of the type-checker we decided to account for them in our theory. We split our set of co-inductive hypotheses (the list of judgments that we are trying to prove) into a set of "safe" hypotheses Θ_{safe}, which can be used immediately, and a set of "unsafe" hypotheses, which can only be used after a computation type constructor has been traversed.

Here are some rules of the corresponding formal system, extended with coinductive hypotheses, which guided our OCaml implementation:

$$\frac{\forall i \in \{1,2\}, \quad \Sigma; \Gamma; \Theta_{\mathsf{safe}}, \Theta_{\mathsf{unsafe}}, (\tau_1 \to \tau_2 : m); \varnothing \vdash \tau_i : m \circ \texttt{Ind}}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \vdash \tau_1 \to \tau_2 : m}$$

$$\frac{t(\alpha_1 : m_1, \ldots, \alpha_n : m_n) \in \Sigma \qquad (\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}}, (t(\tau_1, \ldots, \tau_n) : m) \vdash \tau_i : m \circ m_i)_{1 \leqslant i \leqslant n}}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \vdash t(\tau_1, \ldots, \tau_n) : m}$$

$$\frac{(\tau : m') \in \Theta_{\mathsf{safe}} \qquad m' \geqslant m}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \vdash \tau : m} \qquad\qquad \frac{(\tau : m) \in \Theta_{\mathsf{unsafe}}}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \not\vdash \tau : m}$$

The arrow rule has its premises under a computational type constructor, so it passes all coinductive hypotheses, including the new assumption on $\tau_1 \to \tau_2$, to its safe set. A datatype constructor may be just an abbreviation, so it adds the new hypothesis to the unsafe set.

Finally, whenever a judgment needs to be proved, it immediately succeeds if a stronger hypothesis is in the safe set. On the other hand, if our hypothesis is already in the unsafe set, then we know that it cannot be proven without recursively assuming itself, and we can in fact fail with an error.

5.4 Non-conservativity

Section A.2, Fact 2 shows that our modes do not admit principal judgments – in particular our judgments are non-principal. In particular, the following OCaml declaration can be given either signatures $t(\alpha : \texttt{Ind}, \beta : \texttt{Sep})$ and $t(\alpha : \texttt{Sep}, \beta : \texttt{Ind})$.

```
type ('a, 'b) t = K : 'c -> ('c, 'c) t
```

Our implementation will choose one of the two minimum signatures, depending on the order in which constraints are handled – see Section 5.2. This means that some correct uses of the type will be disallowed: no matter which one is chosen, one of the two following declarations will be rejected:

```
type t1 = T1 : (int, 'a) t -> t1
type t2 = T2 : ('a, int) t -> t2
```

On the other hand, the current implementation, which expands the definition of `t`, accepts both definitions.

We have decided to accept these completeness regressions. Our implementation is cleaner, accepts important examples rejected by the current implementation, and is safer in its handling of cycles, without fuel. In contrast, the counter-examples we could build are fairly esoteric, and we have not found any of them in the current testsuite or user programs. Only time will tell if this assumption is reasonable; we discuss ideas to recover principality (and accept both `t1` and `t2`) in Future Work Section 6.1.

6 Related and future work

6.1 Future work

Richer modes The implementation of our unboxability check is satisfactory in the sense that it accepts most valid (unboxed) definitions. There is however room for improvement, especially in the handling of type equality guards, be they in GADTs or in toplevel equality constraints.

For instance, the way we handle equations of GADTS (see the previous section) is in some sense incomplete, and our language of modes is not principal. We considered extending our modes with modes of the form $\exists(\beta : m).\, m'$ and $m \restriction (\alpha = \tau)$. However, this would significantly increase the complexity of the theory and the implementation, only to handle corner cases that may not be worth it.

Another simpler approach to regain an impression of principality would be to support disjunctions of modes. In the problematic example $t(\alpha, \beta) := \beta \restriction (\alpha = \beta)$, there are two minimum modes $t(\alpha : \texttt{Ind}, \beta : \texttt{Sep})$ and $t(\alpha : \texttt{Sep}, \beta : \texttt{Ind})$, so this type could be given the principal mode $t(\alpha : \texttt{Ind}, \beta : \texttt{Sep}) \vee (\alpha : \texttt{Sep}, \beta : \texttt{Ind})$.

Automatic unboxing Currently, unboxable type declarations are never unboxed automatically, the user has to explicitly ask for it. Automatic unboxing has been considered, but it could break existing code using the foreign function interface. For example, a C function receiving an OCaml value from an unboxable but non-unboxed type currently needs to unbox it explicitly. The same action on an (automatically) unboxed type would fail if the C code is not changed accordingly.

Disjoint GADT unboxing One could wish to unbox multi-constructors GADTs in the case where typing constraints lead to the mutual exclusion of the different constructors.

```
type _ value =
  | Int : int -> int value
  | Bool : bool -> bool value
  | Pair : ('b * 'c) -> ('b * 'c) value
```

While each ground instance of this value type has a single possible constructor and could be unboxed, pattern-matching on an `'a value` would then have to be disallowed: pattern-matching learns the value of `'a` from inspecting the GADT constructor, which is not present anymore in the unboxed representation. It seems fishy to only allow pattern-matching on partially-determined instances of the type, and we did not investigate further.

6.2 Just get rid of the damn float thing

The fairly elaborate sufferings we just went through are caused by the dynamic unboxing optimization for arrays of floating point numbers. If this dynamic optimization were removed, we would not need separability anymore and the unboxing check could also be removed. This dynamic check also has consequences on other features: by making a $\top$ type (our $\exists\alpha.\,\alpha$) unsound, it prevents extending the relaxed value restriction to generalize contravariant variables (Garrigue, 2004, page 11).

There is an ongoing debate in the OCaml community on this idea. An experimental configure flag `-no-flat-float-array` can be set to disable dynamic flat representation optimizations in the implementation, and benefit from the simpler type theory. Since 4.06 (November 2017), a new primitive monomorphic type `floatarray` exists that is specialized for unboxed float arrays, and can be used by users intending to use this optimization, but it lacks library support and convenient array-indexing notations. The problem is with generic code, written against parametric `'a array` value and then applied in numeric programs on float array, whose performance would be silently degraded with an important slowdown. In other terms, removing the dynamic optimization would be acceptable for experts authors of numerical code willing to modify their codebase, but hurt the performance of programs written naively by beginners.

6.3 Related work

We discussed the existing implementation of the unboxability check in Section 1.2.

The approach presented in our work is largely inspired from the way the variance of type declarations is handled in languages with subtyping (Abel, 2008; Scherer and Rémy, 2013).

The memory representation of values used in OCaml and similar languages finds its origin in Lisp-like languages (Leroy, 1990). Despite the advantage of allowing every value to have the same, single-word representation, this approach also has obvious limitations in terms of performances due to the introduction of indirections. As a consequence, ways of lowering this overhead in certain scenarios have been investigated, one possibility being to mix tagged and untagged representations (Peterson, 1989; Leroy, 1990, 1992). Another idea that has been investigated is to consider unboxed values as first-class citizens, although distinguished by their types (Peyton Jones and Launchbury, 1991).

References

Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 2008.

Simon Colin. Specifying the unboxability check on mutually recursive datatypes in OCaml, 2018. Internship report (under the supervision of Gabriel Scherer).

Jacques Garrigue. Relaxing the value restriction. In *FLOPS*, 2004.

Rodolphe Lepigre and Christophe Raffalli. Abstract representation of binders in ocaml using the bindlib library. In *LFMTP*, 2018.

Xavier Leroy. Efficient data representation in polymorphic languages. In *PLILP*, 1990.

Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL*, 1992.

John Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *FPCA*, 1989.

Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *FPCA*, 1991.

Gabriel Scherer and Didier Rémy. Gadts meet subtyping. In *ESOP*, 2013.

Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *TOPLAS*, 2007.

A Semantics

We give a semantic model of types, datatypes and modes in Figure 5. The general idea is to be able to interpret the judgment $\Gamma \vdash \tau : m$ as follows: "if we replace each type variable $(\alpha : m_\alpha)$ of Γ by a closed/ground type with the correct separability mode m_α, then τ will really have the separability mode m".

Ode to semantics. In Section 3 we have given a set of inference rules to prove judgments of the form $\Sigma; \Gamma \vdash \tau : m$, guided by the intuition that this *should* provide evidence that τ is separable. Inference rules are our key contribution as they can easily be turned into a checking or inference algorithm, but they are also subtle, and may very well be wrong. They are hard to audit by someone else who does not trust us, the authors: there are a lot of details to check.

The point of a semantics is to provide an "obvious" counterpart to inference rules. A formal definition of separability, or whatever property an inference system is trying to capture, that is self-evident, can be easily checked and trusted by other people – in particular, it does not depend on the previous inference rules in any way. It serves as a specification, can use arbitrary mathematical operations, and does not need to be computable or close to an algorithm. Finally, one wishes to prove that the syntactic inference rules and the semantics coincide in some sense; this implies that we can trust the inference rules as much as we trust the semantics.

Our semantics can also be understood as an idealized "model" of the programming language we are studying: it contains the assumptions that we make about the language and type system, and can be compared to OCaml to ensure that those assumptions are correct.

Ground value semantics. To define what it means for a closed type $\underline{\tau}$ to have the mode m, we use the set-theoretic intuition introduced earlier: "its values are all floating-point numbers, or none of them are". To do so formally, we introduce a syntax of closed/ground values, and specify which ground values inhabit which ground types.

Our notion of ground/closed values is idealized; in particular, we represent all values at a function type as an opaque `function` blob, as if all functions were represented in the same way in the programming languages we model. They are not, but the differences in representation are irrelevant to reason about separability.

In the figure we define ground values $\underline{v}$ (just data, no term variables), ground types $\underline{\tau}$ (no free type variables), ground datatypes (no free type variables or parameters), and blocks of datatype definitions σ, which assign a datatype to each datatype constructor of a signature Σ.

Finally, we define a series of semantic judgments using the symbol $(\models)$. The judgment $\underline{v} \models^\sigma \underline{\tau}$ specifies when a value inhabits a type, relative to a block of definitions σ to interpret type constructors. The judgment $\underline{v} \models^\sigma \underline{A}$ specifies when a value inhabits a type parameter; note in particular how the judgments for `unboxed` datatypes reflect what happens in an implementation.

The judgments $\underline{\tau} \models^\sigma m$ and $\underline{A} \models^\sigma m$ specify when a type expression or datatype respect the separability mode m. They are defined in terms of our set-theoretic characterizations, separable(X). Finally, $A \models^\sigma t(\overline{\alpha : m})$ specifies when a parametrized datatype A respects a mode signature, and $\sigma \models \Sigma$ specifies that a definition block respects a signature block.

In Figure 6, we use these specifications to build a semantic counterpart for each of our syntactic judgments. This lets us easily formulate soundness and partial completeness results. For example, $\Sigma; \Gamma \models \tau : m$ is the semantic counterpart of the judgment $\Sigma; \Gamma \vdash \tau : m$. It captures what it means for the judgment to hold: for any valid definitions $\sigma \models \Sigma$, and any choice of ground types $\underline{\gamma}$ valid for Γ ($\underline{\gamma} \models^\sigma \Gamma$), replacing the variables in τ by the ground types in $\underline{\gamma}$ gives a ground type $\underline{\gamma(\tau)}$ at mode m, that is, $\underline{\gamma(\tau)} \models^\sigma$ holds.

Ground/closed values $\underline{v}$, type expressions $\underline{\tau}$, datatypes $\underline{A}$,
context valuations $\underline{\gamma}$, datatype signature valuations σ

$$
\begin{array}{lll}
\underline{v} ::= & \texttt{true}, \texttt{false} & \text{booleans} \\
| & \text{int}(n \in \mathbb{N}) & \text{integers} \\
| & \text{float}(x \in \mathbb{R}) & \text{float} \\
| & (\underline{v_1}, \ldots, \underline{v_n}) & \text{tuple} \\
| & \texttt{function} & \text{function} \\
| & \{l_1 : \underline{v_1}; \ldots; l_n : \underline{v_n}\} & \text{record} \\
| & C_i\ \underline{v} & \text{variant}
\end{array}
\qquad
\begin{array}{rcl}
\text{closed}(\tau) & := & \forall \alpha,\ \alpha \notin \tau \\
\mathsf{GType} & := & \{\underline{\tau} \mid \text{closed}(\tau)\} \\
\text{closed}(A) & := & \forall \tau \in A,\ \text{closed}(\tau) \\
\mathsf{GDatatype} & := & \{\underline{A} \mid \text{closed}(A)\} \\
\underline{\gamma} & \in & \mathcal{P}(\mathsf{TypeVar} \to \mathsf{GType}) \\
\sigma & \in & \mathcal{P}(\mathsf{TypeConstructor} \to \mathsf{GDatatype})
\end{array}
$$

Values at a ground type expression $\underline{v} \vDash^{\sigma} \underline{\tau}$

$$
\frac{}{\texttt{true}, \texttt{false} \vDash^{\sigma} \underline{\texttt{bool}}} \qquad \frac{}{\text{int}(n) \vDash^{\sigma} \underline{\texttt{int}}} \qquad \frac{}{\text{float}(x) \vDash^{\sigma} \underline{\texttt{float}}} \qquad \frac{}{\texttt{function} \vDash^{\sigma} \underline{\tau_1 \to \tau_2}}
$$

$$
\frac{\left(\underline{v_i} \vDash^{\sigma} \underline{\tau_i}\right)_{1 \leqslant i \leqslant n}}{(\underline{v_1}, \ldots, \underline{v_n}) \vDash^{\sigma} \underline{(\tau_1 \times \cdots \times \tau_n)}} \qquad \frac{\exists \underline{\tau} \in \mathsf{GType}, \quad \underline{v} \vDash^{\sigma} \underline{\kappa[\tau/\alpha]}}{\underline{v} \vDash^{\sigma} \underline{\exists \alpha.\kappa}} \qquad \frac{\forall \underline{\tau} \in \mathsf{GType}, \quad \underline{v} \vDash^{\sigma} \underline{\kappa[\tau/\alpha]}}{\underline{v} \vDash^{\sigma} \underline{\forall \alpha.\kappa}}
$$

$$
\frac{(t(\overline{\alpha}) := A) \in \sigma \qquad \underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{A[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}}{\underline{v} \vDash^{\sigma} \underline{t(\tau_1, \ldots, \tau_n)}} \qquad \frac{(\underline{\tau_1} = \underline{\tau_2}) \implies \underline{v} \vDash^{\sigma} \underline{\kappa}}{\underline{v} \vDash^{\sigma} \underline{\kappa \restriction (\tau_1 = \tau_2)}}
$$

Values at a ground datatype $\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{A}$

$$
\frac{\underline{v} \vDash^{\sigma} \underline{\tau_i} \qquad 1 \leqslant i \leqslant n}{C_i\ \underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{C_1 \text{ of } \tau_1 \mid \cdots \mid C_n \text{ of } \tau_1}} \qquad \frac{\underline{v} \vDash^{\sigma} \underline{\tau}}{\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{C \text{ of } \tau\ [@@\texttt{unboxed}]}}
$$

$$
\frac{\left(\underline{v_i} \vDash^{\sigma} \underline{\tau_i}\right)_{1 \leqslant i \leqslant n}}{\{l_1 : \underline{v_1}; \ldots; l_n : \underline{v_n}\} \vDash^{\sigma}_{\mathsf{decl}} \underline{\{l_1 : \tau_1; \ldots; l_n : \tau_n\}}} \qquad \frac{\underline{v} \vDash^{\sigma} \underline{\tau}}{\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{\{l : \tau\}\ [@@\texttt{unboxed}]}} \qquad \frac{\underline{v} \vDash^{\sigma} \underline{\tau}}{\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{\tau}}
$$

Ground types at a mode $\underline{\tau} \vDash^{\sigma} m$, ground valuations at a context $\underline{\gamma} \vDash^{\sigma} \Gamma$

$$
\begin{array}{rcl}
\text{isfloat}(\underline{v}) & := & \exists x,\ \underline{v} = \text{float}(x) \\
\text{separable}(X) & := & (\forall \underline{v} \in X,\ \text{isfloat}(v)) \vee (\forall \underline{v} \in X,\ \neg\, \text{isfloat}(v))
\end{array}
$$

$$
\frac{}{\underline{\tau} \vDash^{\sigma} \texttt{Ind}} \qquad \frac{\text{separable}(\{\underline{v} \mid \underline{v} \vDash^{\sigma} \underline{\tau}\})}{\underline{\tau} \vDash^{\sigma} \texttt{Sep}} \qquad \frac{\forall \tau' \lhd \tau, \quad \underline{\tau'} \vDash^{\sigma} \texttt{Sep}}{\underline{\tau} \vDash^{\sigma} \texttt{Deepsep}} \qquad \frac{\forall (\alpha : m) \in \Gamma, \quad \underline{\gamma(\alpha)} \vDash^{\sigma} m}{\underline{\gamma} \vDash^{\sigma} \Gamma}
$$

Remark: we have $\underline{\exists \alpha.\, \alpha} \nvDash^{\sigma} \texttt{Sep}$, which means that not all ground types are separable.

Ground datatypes at a mode $\underline{A} \vDash^{\sigma}_{\mathsf{decl}} m$,
parametrized datatypes at a mode signature $A \vDash^{\sigma} t(\overline{\alpha : m})$,
datatype definitions at a block signature $\sigma \vDash^{\sigma} \Sigma$

$$
\frac{\text{separable}(\{\underline{v} \mid \underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{A}\})}{\underline{A} \vDash^{\sigma}_{\mathsf{decl}} \texttt{Sep}} \qquad \frac{\forall \underline{\gamma} \vDash^{\sigma} \overline{\alpha : m}, \quad \underline{\gamma(A)} \vDash^{\sigma}_{\mathsf{decl}} \texttt{Sep}}{A \vDash^{\sigma} t(\overline{\alpha : m})} \qquad \frac{\forall (t(\overline{\alpha}) := A) \in \sigma, \quad A \vDash^{\sigma} \Sigma(t)}{\sigma \vDash \Sigma}
$$

Figure 5: Ground semantics

$$\frac{\forall \sigma \vDash \Sigma, \forall \gamma \vDash^{\sigma} \Gamma, \quad \gamma(\tau) \vDash^{\sigma} m}{\Sigma; \Gamma \vDash \tau : m} \qquad \frac{\forall \sigma \vDash \Sigma, \forall \gamma \vDash^{\sigma} \Gamma, \quad \gamma(A) \vDash^{\sigma}_{\mathsf{decl}}}{\Sigma; \Gamma \vDash_{\mathsf{decl}} A}$$

$$\frac{\forall \sigma_0 \vDash \Sigma_0, \quad \sigma_0, \sigma \vDash \Sigma_0, \Sigma}{\Sigma_0 \vDash \sigma \dashv \Sigma}$$

Figure 6: Judgment semantics

One does not have to trust the inference rules of our syntactic judgment $\Sigma; \Gamma \vdash \tau : m$, which may very well be wrong, to trust that this semantic judgment $\Sigma; \Gamma \vDash \tau : m$ captures a good notion of separability. One need to read the declarative rules of figures 5 and 6, which we designed to be obvious. It is now evident what the right statements should be for soundness and completeness of our syntactic inference system: it is sound if the syntactic judgment implies the semantic judgment (all judgments with a syntactic derivation are semantically true), and complete if the converse holds (all true judgments can be established by a syntactic derivation).

A.1 (Maybe-)Soundness and (in-)completeness

In a research paper, the perfect way to evaluate a syntactic system of inference rules is to provide an independent declarative semantics for it, state the corresponding soundness/completeness results, and prove them.

In real-world implementations of a programming language, the syntactic system is rarely written down (often, only the algorithm inspired from the rules is kept), the declarative semantics are only vague intuitions in the mind of the rule designers, and the statements are never formulated precisely enough to dream of a proof.

In this imperfect research paper, we have precise rules, and an independent declarative semantics, and precise statements... but most proofs are missing. In fact, it is fairly non-obvious that those proofs exist: as we were doing this precise formalization work we have discovered that completeness does not hold in presence of constraints, and even that the system is not principal – not only the inference rules, but even the semantics.

This subsection contains the best state of our knowledge and hopes about the formal status of our inference rules.

Conjectured Lemma 3 (Type expression soundness)**.** $\Sigma; \Gamma \vdash \tau : m \implies \Sigma; \Gamma \vDash \tau : m$

Remark: it is not obvious that the existential rule is sound, for example. It proves $\exists \alpha. \tau : m$ under a premise $\alpha : \texttt{Ind} \vdash \tau : m$. The values of $\exists \alpha. \tau$ are the union of all the $\underline{\tau[\sigma/\alpha]}$, and our induction hypothesis tells us that each $\underline{\tau[\sigma/\alpha]}$ indeed has the mode m. But separability is not at all stable by union: both `int` and `float` are separable, but their union is not. One needs to argue that either all $\underline{\tau[\sigma/\alpha]}$ are inhabited by floating-point numbers only, or that none of them contain floating-point numbers, and this is a subtle property of the structure of algebraic datatypes.

Conjectured Theorem 1 (Soundness)**.** $\Sigma_0 \vdash \sigma \dashv \Sigma \implies \Sigma_0 \vDash \sigma \dashv \Sigma$

Fact 1 (Incompleteness)**.** *There exists a τ such that $\varnothing; \varnothing \vDash \tau : m$ but $\varnothing; \varnothing \nvDash \tau : m$.*

Proof. Consider the two types $\tau_1 := \exists \beta. \beta \restriction (\texttt{int} = \texttt{bool})$ and $\tau_2 := \exists \alpha \beta. \beta \restriction (\alpha = \alpha \times \texttt{int})$. They are semantically separable, but our type system cannot prove it. The reason why they are semantically separable is that the equations never hold in our semantic model with finite ground types, so the constrained type is empty and thus trivially separable.

We could make our syntactic inference rules more complete by adding more equality-reasoning power to the judgment $\Sigma; \Gamma \vdash (\tau_1 = \tau_2)$. For the first case ($\texttt{int} = \texttt{bool}$), incompatible type constructors should not be considered equal. In the second case, ($\alpha = \alpha \times \texttt{int}$), one could add an occurs-check to rule out such recursive types, but note that this property, true in our model, may not hold in the real world – it does not in OCaml when `-rectypes` is used. □

This result suggests that equality-reasoning can be fairly subtle and that we should not expect syntactic completeness on types with constraints. We can still hope to have completeness in their absence.

Definition 2 (Constraint-free)**.** *A type τ, datatype A or definition block σ is* constraint-free *if it does not contain any type equality constraint ($\kappa \restriction (\alpha = \tau')$). We write $\mathsf{CF}(\tau)$, $\mathsf{CF}(A)$ or $\mathsf{CF}(\sigma)$.*

Conjectured Lemma 4 (Completeness on constraint-free type expressions)**.**

$$\mathsf{CF}(\tau) \wedge \Sigma; \Gamma \vDash \tau : m \implies \Sigma; \Gamma \vdash \tau : m$$

Conjectured Theorem 2 (Completeness on constraint-free signatures)**.**

$$\mathsf{CF}(\sigma) \wedge \Sigma_0 \vdash \sigma \dashv \Sigma \implies \Sigma_0 \vDash \sigma \dashv\vDash \Sigma$$

A.2 Constraint-free principality

Unfortunately, there is more bad news to come for equality constraints.

Fact 2 (Semantic non-principality)**.** *There exists a parametrized datatype A with $A \vDash^{\varnothing} t(\Gamma_1)$ and $A \vDash^{\varnothing} t(\Gamma_2)$, but $A \nvDash^{\varnothing} t(\min(\Gamma_1, \Gamma_2))$.*

Proof. Over two parameters α, β, take $A := \alpha \restriction (\alpha = \beta)$. Both $\Gamma_1 := \alpha : \texttt{Ind}, \beta : \texttt{Sep}$ and $\Gamma_2 := \alpha : \texttt{Sep}, \beta : \texttt{Ind}$ are admissible signatures for A. In particular, our inference rules can verify them: any Γ' with $\Gamma' \vdash \alpha = \beta$ has $(\alpha : \texttt{Sep}, \beta : \texttt{Sep})$. However, the minimum signature $\alpha : \texttt{Ind}, \beta : \texttt{Ind}$ is not valid for A. □

Note the example in the above proof can be represented as an OCaml datatype as follows.

```
type (_, _) strange_eq =
  | Strange_refl : 'a -> ('a, 'a) strange_eq [@@unboxed]
```

This results shows that some limitations of the system are not due to our typing rules, but a fundamental lack of expressiveness of the current modes and mode signatures as a specification/reasoning language. We would need a richer language of modes, keeping track of equalities between types, to hope to get a principal system.

Lemma 5. *If $\Sigma; \Gamma_1 \vdash \tau : m$ and $\Sigma; \Gamma_2 \vdash \tau : m$ and $\mathsf{CF}(\tau)$ then $\Sigma; \min(\Gamma_1, \Gamma_2) \vdash \tau : m$*

Proof. This proof is done by induction on the two derivations at once, but we need to use the Cut Elimination Lemma 1 first to be able to assume, by inversion/syntax-directedness, that the rules on both sides are the same. □

Corollary 1 (Constraint-free principality)**.**
If $\Sigma; \Gamma \vdash \tau : m$ with $\mathsf{CF}(\tau)$, then there exists a minimal context Γ_{min} such that $\Sigma; \Gamma_{\mathsf{min}} \vdash \tau : m$.
If $\Sigma_0 \vdash \sigma \dashv \Sigma$ with $\mathsf{CF}(\sigma)$, then there exists a minimal signature Σ_{min} such that $\Sigma_0 \vdash \sigma \dashv \Sigma_{\mathsf{min}}$.