Unboxed Data Constructors:
Or, How cpp Decides a Halting Problem[*]

NICOLAS CHATAING, ENS Paris, France
STEPHEN DOLAN, Jane Street, UK
GABRIEL SCHERER, Inria, France
JEREMY YALLOP, University of Cambridge, UK

We propose a new language feature for ML-family languages, the ability to selectively *unbox* certain data constructors, so that their runtime representation gets compiled away to just the identity on their argument. Unboxing must be statically rejected when it could introduce confusion, that is, distinct values with the same representation.

We discuss the use-case of big numbers, where unboxing allows to write code that is both efficient and safe, replacing either a safe but slow version or a fast but unsafe version. We explain the static analysis necessary to reject incorrect unboxing requests. We present our prototype implementation of this feature for the OCaml programming language, discuss several design choices and the interaction with advanced features such as Guarded Algebraic Datatypes.

Our static analysis requires expanding type definitions in type expressions, which is not necessarily normalizing in presence of recursive type definitions. In other words, we must decide normalization of terms in the first-order λ-calculus with recursion. We provide an algorithm to detect non-termination on-the-fly during reduction, with proofs of correctness and completeness. Our algorithm turns out to be closely related to the normalization strategy for macro expansion in the cpp preprocessor.

CCS Concepts: • **Software and its engineering** → **Data types and structures**; **Functional languages**; *Source code generation*; • **Theory of computation** → *Type structures*.

Additional Key Words and Phrases: data representation, sum types, tagging, boxing, recursive definitions, termination

1 INTRODUCTION

1.1 Sum Types and Constructor Unboxing

A central construct of ML-family programming languages is *algebraic datatypes*, in particular *(disjoint) sum types*, also called *variant types*. In OCaml syntax:

Authors' addresses: Nicolas Chataing, ENS Paris, France, nicolas.chataing@gmail.com; Stephen Dolan, Jane Street, UK, stedolan@stedolan.net; Gabriel Scherer, Inria, France, gabriel.scherer@gmail.com; Jeremy Yallop, University of Cambridge, UK, jeremy.yallop@cl.cam.ac.uk.

```
type rel_num =                        let non_negative = function
| Pos of nat                          | Pos(_) -> true
| Zero                                | Zero -> true
| Neg of nat                          | Neg(_) -> false
```

Values of this sum type are of the form Pos(n) or Zero or Neg(n); in particular they start with a *(data) constructor*, here Pos or Zero or Neg, followed by *arguments* (zero, one or several arguments), which is data carried along the constructor. In mathematical terms this corresponds to a *sum* or *coproduct* $A + B$ between sets of values, rather than a *union* $A \cup B$, because one can always tell from which side of the sum a value is coming from, by pattern-matching.

In mathematics, the coproduct between sets is typically implemented as a (disjoint) union:

$$A + B \ \overset{\text{def}}{=}\ (\{0\} \times A) \uplus (\{1\} \times B) \qquad\qquad \sum (A_i)^{i \in I} \ \overset{\text{def}}{=}\ \biguplus_{i \in I} (\{i\} \times A_i)$$

using a cartesian product of the form $\{i\} \times S$ to build the pairs of a *tag* value i and an element of S.

Software implementations of programming languages use the same approach: the representation of sum types in memory typically includes not only the data for their arguments, but also a *tag* representing the constructor; pattern-matching on the value is implemented by a test on this tag – often followed by accessing the arguments of the constructor. For example, the standard OCaml compiler will represent the value Neg(n) by a pointer to a block of two consecutive machine words in heap memory, with the first word (the block *header*) containing the tag (among other things), followed by the argument of type nat in the second word. We say that the parameter of type nat is *boxed* in this representation: it is contained inside another value, a "box".

Boxing induced by data constructors introduces a performance overhead. In the vast majority of cases it is negligible, as ML-family language implementations heavily optimize the allocation of heap blocks and often benefit from good spatial locality. There are still some performance-critical situations where the overhead of boxing (compared to carrying just a nat) is significant.

One common, easy case where boxing can be avoided is for sum types with a single constructor: `type t = Foo of arg`. Values of this type could be represented exactly as values of type arg, as there are no other cases to distinguish than Foo. Haskell provides an explicit newtype binder for this case. In OCaml, this is expressed by using the @@unboxed attribute on the datatype declaration.

```
newtype Fd = Fd Int -- Haskell       type fd = Fd of int [@@unboxed] (* OCaml *)
```

Note that the programmer could have manipulated values of type int directly, instead of defining a single-constructor type fd (file descriptor) isomorphic to int; but often the intent is precisely to define an isomorphic but incompatible type, to have explicit conversions back and forth in the program, and avoid mistaking one for the other. Single-constructor unboxing makes efficient programming more expressive or safe, or expressive/safe programming more efficient.

1.2 Constructor Unboxing

In the present work, we introduce a generalization of single-constructor unboxing, that we simply call *constructor unboxing*. It enables unboxing one or several constructors of a sum type, as long as disjointedness is preserved.

Our main example for this selective unboxing of constructors is a datatype zarith of (relative) numbers of arbitrary size, that are represented either by an OCaml machine-word integer (int), when they are small enough, or a "big number" of type Gmp.t implemented by the GMP library [Granlund and contributors 1991]:

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

The [@unboxed] attribute requests the *unboxing* of the data constructor Small, that is, that its application be represented by just the identity function at runtime. This request can only be satisfied if it does not introduce *confusion* in the datatype representation, that is, two distinct values (at the source level) that get the same representation. In our zarith example, the definition can be accepted with an unboxed constructor, because the OCaml representation of int (immediate values) is always disjoint from the boxed constructor Big **of** Gmp.t (heap blocks). Otherwise we would have rejected this definition statically.

```
type clash = Int of int [@unboxed] | Also_int of int [@unboxed]
Error: This declaration is invalid, some [@unboxed] annotations introduce
         overlapping representations.
```

While constructor boxing is cheap in general, unboxing the Small constructor of the zarith type does make a noticeable performance difference, because arithmetic operations in the fast path are very fast, even with an explicit overflow check, so the boxing overhead would be important. On a synthetic benchmark, unboxing the Small constructor provides a 20% speedup.

1.3 Head Shapes

We propose a simple criterion to statically reject unboxing requests that would introduce confusion – several distinct values with the same runtime representation – parameterized on two notions:

- The *head* of a value, an approximation/abstraction of its runtime representation.
- The *head shape* of a type, the (multi)set of heads of values of this type.

Our static analysis computes the head shape of datatype definitions, and rejects definitions where the same head occurs several times.

In addition, we require that the head of a value be efficiently computable at runtime. In presence of an unboxed constructor, the head shape of its argument type is used to compile pattern-matching. The generated code may branch at runtime on the head of its scrutinee.

We provide a definition of heads that is specific to the standard OCaml runtime representation. Other languages would need to use a different definition, but the static checking algorithm and the pattern-matching compilation strategy are then fully language-agnostic.

1.4 A Halting Problem

To compute the head shape of types, and therefore to statically check unboxed constructors for absence of confusion, we need to unfold datatype abbreviations or definitions. Example:

```
type num = int                          type id =
and name = Name of string [@unboxed]    | By_number of num [@unboxed]
                                        | By_name of name [@unboxed]
```

To check the final definition of id, we must determine that num is the primitive type int and that name has the same representation as a primitive string, through a definition-unfolding process that corresponds to a form of β-reduction.

In the general case, type definitions may contain arbitrary recursion. In particular, unfolding definitions may not terminate for some definitions; we need to detect this to prevent our static analysis from looping indefinitely.

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

This practical problem is in fact exactly the halting problem – deciding whether terms have a normal form – for the *pure, first-order λ-calculus with recursion*. We present an algorithm to detect non-termination on the fly, during normalization. Running the algorithm on a term either

terminates with a normal form, or it stops after detecting a loop in finite time. We prove that our algorithm is correct (it rejects all non-terminating programs) and complete (it accepts all terminating programs). The proof of correctness requires a sophisticated termination argument.

It turns out that this termination-monitoring algorithm is related to the approach that the cpp preprocessor uses to avoid non-termination for recursive macros – another example where non-termination must be detected on the fly.

1.5 Contributions and Outline

We claim the following contributions:

(1) We specify the feature of "constructor unboxing".
(2) We implement this feature in an experimental version of the OCaml compiler[1], and hope to eventually merge it upstream. Section 4 details various design considerations that arose when scaling the feature to a full programming language.
(3) We perform a case study on zarith (Section 2), demonstrating that the feature can noticeably improve the performance of safe OCaml code, or noticeably improve safety by removing the need for unsafe OCaml code that was previously used for this purpose.
(4) We provide a static analysis to reject unboxing requests that would introduce confusion. Our formal treatment (Section 3) is independent from the OCaml programming language.
(5) As an unplanned side-effect, we propose an *on-the-fly termination checking* algorithm for the pure, first-order λ-calculus with recursion (Section 6). We prove that this algorithm is correct (it rejects all non-terminating programs) and complete (it accepts all terminating programs).
(6) There are unexpected parallels between our termination checking work and an algorithm written in 1986 by Dave Prosser [Prosser 1986], to ensure termination of cpp macro expansions. In Section 7 we compare our algorithm to cpp's – for which correctness has, to our knowledge, not been proved.

Our Related Work Section 8 has a mix of discussions on production programming languages, experimental language designs, and theoretical work on termination of recursive λ-calculi. We list unboxing-related features in other languages and implementations (GHC Haskell, MLton, F♯, Scala, Rust). We mention research projects that provide an explicit description language for data layout (Hobbit, Dargent, Ribbit). Finally, we discuss Rust niche-filling optimizations in more detail.

2 CASE STUDY: BIG INTEGERS

2.1 A Primer on OCaml Value Representations

In the reference implementation of OCaml, sum types have the following in-memory representation:

- *constant* constructors (without parameters), such as [] or None, are represented as *immediate values*, exactly like OCaml integers. The immediate values used are 0 for the first constant constructor in the type, 1 for the second one, etc.
- *non-constant* constructors, with one or several parameters, are represented by pointers to memory *blocks*, that start with a header word followed by the representation of each parameter. The header word contains some information of interest for the GC and runtime system, including a one-byte *tag*: 0 for the first non-constant constructor, 1 for the second one, etc., and the *arity* of the constructor – its number of parameters.

Some primitive types that are not sum types – strings, floats, lazy thunks, etc. – have special support in the OCaml runtime. They are also represented as memory blocks with a header word, using a dozen reserved high tag values that cannot be used by sum constructors. They also include

---

[1]We included an anonymized git repository with our submission.

a tag `Custom_tag` (255) for blocks whose parameters are foreign data words accessed only through the C Foreign Function Interface (FFI).

OCaml distinguishes immediate values from pointers to blocks (both word-sized) by reserving the least significant bit for this purpose: immediate values are encoded in odd words, while pointers are even words. (In particular, OCaml integers are 63-bits on 64-bit machines.) Pattern-matching can check this immediate-or-pointer bit first, to distinguish constant from non-constant constructors, and then switch on a small immediate value or block tag.

2.2 Unsafe Zarith

The Zarith library [Miné and Leroy 2012] provides an efficient OCaml implementation of arbitrary-precision integers, on top of the reference C library Gmp [Granlund and contributors 1991].

Some users of arbitrary-precision integers perform a majority of their computations on very large integers, way larger than the "small" integers that fit a machine word. On the other hand, many users perform computations that rarely, if ever, overflow, but they need the guarantee that the result will remain correct even in presence of occasional overflows. For this latter use-case, we want to minimize the overhead of Zarith compared to using machine-sized integers directly – in OCaml, the `int` type. We want to ensure that when operating on "small" integers, the operation only performs machine-size arithmetic and bound checks, without any memory allocation, nor any call to non-trivial C functions; in other words, we want a "fast path" for small integers.

Zarith uses a type `Zarith.t` whose inhabitants are either machine-sized OCaml integers (type `int`) or a "custom" value, a pointer to a memory block with tag `Custom_tag` and the gmp digits as arguments. This type cannot be expressed in OCaml today, so Zarith has to use the low-level, unsafe compiler intrinsics to perform unsafe checks and casts, giving up on the type- and memory-safety usually guaranteed by the OCaml programming language.[2]

```
type t                       (* int or gmp integer (in a custom block) *)
external is_small_int: t -> bool = "%obj_is_int" (* imm-or-block bit *)
external unsafe_to_int: t -> int = "%identity"   (* unsafe cast *)
external of_int: int -> t = "%identity"          (* unsafe cast *)

external c_add: t -> t -> t = "ml_z_add"        (* slow path, in C *)
let add x y =
  if is_small_int x && is_small_int y then begin (* ``fast path'' addition *)
    let z = unsafe_to_int x + unsafe_to_int y in
    (* Overflow check -- Hacker's Delight, section 2.12 *)
    if (z lxor unsafe_to_int x) land (z lxor unsafe_to_int y) >= 0
    then of_int z else c_add x y
  end else (* ``slow path'' *) c_add x y
```

2.3 Unboxed Zarith

Using our experimental OCaml compiler with constructor unboxing, we can write instead:

```
type custom_gmp_t [@@shape [custom]] (* gmp integer (in custom block) *)
type t = Small of int [@unboxed] | Big of custom_gmp_t [@unboxed]
```

---

[2]A previous version of Zarith would even implement the small-integer fast path in assembly code on some architectures to use native overflow-checking instructions instead of bit-fiddling checks in OCaml. But the cost of switching from OCaml to the assembly FFI in the fast path in fact made this version slower than the OCaml version – it was also painful to maintain.

```
external c_add: t -> t -> t = "ml_z_add"
let add a b = match a, b with
  | Small x, Small y ->
      let z = x + y in
      (* Overflow check -- Hacker's Delight, section 2.12 *)
      if (z lxor x) land (z lxor y) >= 0
      then Small z else c_add a b
  | _, _ -> c_add a b
```

This code is equivalent to the previous version, generates exactly the same machine code, but does not require any unsafe casts. One still has to trust the FFI code of c_add to respect the intended memory representation, and we trust the annotation shape [custom] that claims that the abstract type custom_gmp_t is only inhabited (via the FFI) by Custom_tag-tagged blocks. But the unsafe boundary has been pushed completely off the fast path; it can disappear completely in other examples not involving bindings to C libraries.

On a synthetic microbenchmark, we observed that our new version has essentially the same performance as the previous unsafe code, and is 20% more efficient than a boxed version – using a sum type without [@unboxed] annotations.

Case-study conclusion. Unboxing relieves some of the tension between safety and efficiency in performance-critical libraries. Users sometimes have to choose between safe, idiomatic sum types or more efficient encodings that are unsafe and require more complex code. In some cases, such as Zarith, constructor unboxing provides a safe, clear and efficient implementation.

2.4 Other Use-Cases

Let us briefly mention a few other use-cases for constructor unboxing.

A ropes benchmark. Our original design proposal Yallop [2020] includes a similar performance experiment on ropes (trees formed by concatenating many small strings), reporting a 30% performance gain on an example workload – a similar performance ballpark to our 20%. The example was implemented using unsafe features only as unboxing was not implemented at the time; we can express it as follows:

```
type rope =
  | Leaf of string [@unboxed]
  | Branch of { llen: int; l:rope; r:rope }
```

Coq's native_compute *machinery.* Another use-case where constructor unboxing could provide safety is the representation of Coq values in the native_compute implementation of compiled reduction, first introduced in Boespflug, Dénès and Grégoire [2011]. native_compute is a Coq tactic that compiles a Coq term into an OCaml term such that evaluating the OCaml term (by compilation then execution, in the usual call-by-value OCaml strategy) computes a strong normal form for the original Coq term. It uses an unsafe representation of values that mixes (unboxed) functions, sum constructors, and immediate values, and could be defined as a proper OCaml inductive if constructor unboxing was available. (The relation with constructor unboxing was pointed out to us by Jacques-Henri Jourdan, Guillaume Melquiond and Guillaume Munch-Maccagnoni.)

A partial sum-type presentation of dynamic values. This feature could make some forms of dynamic introspection of runtime values more ergonomic than what is currently exposed in the Obj module. We could think of defining a sort of "universal type" as follows:

```
type dyn =
| Immediate of int [@unboxed]
| Block of dyn array [@unboxed]
| Float of float [@unboxed]
| String of string [@unboxed]
| Function of (dyn -> dyn) [@unboxed]
| Custom of custom [@unboxed]
| ...
and custom [@@shape [custom]]

let to_dyn : 'a -> dyn =
  fun x -> (Obj.magic x : dyn)
```

This interface cannot cover all needs – for example it is not possible to distinguish between Int32, Int64 by their tags, they would be lumped together in the Custom case – and we may need to restrict it due to shape approximations required by portability concerns – see Section 4.2. But it would still provide a pleasant pattern-matching interface for unsafe value introspection code that people write today using the Obj module directly.[3]

Non-use-case: magic performance gains in many places. One should *not* hope that there are plenty of performance-sensitive OCaml codebases lying around today that would get a noticeable performance boost by sprinkling a few (or many) unboxing annotations. In the vast majority of cases, unboxing provides no noticeable performance improvement. There are two reasons:

(1) Allocating values in the OCaml minor heap is *really fast.* In the boxed version of the Zarith benchmark, checking against integer overflow is slower than allocating the Small constructor. Most programmers overestimate the performance cost of boxing, it makes little difference for most workloads.[4]

(2) In the few cases of performance-sensitive programs where boxing would add noticeable overhead, the authors of the program already chose a different implementation strategy to avoid this boxing, for example using unsafe tricks as Zarith.

The second point is common to most language design for performance: *existing* performance-sensitive codebases are written with the existing feature set in mind, and typically do not present low-hanging fruits for a new performance-oriented feature. The benefits rather come from giving more, better options (safer, simpler, more idiomatic) to write performant code in the *future.*

It is also reassuring for users to know that a new idiom made available is "zero-cost". Even in cases where in fact the non-optimized approach would have perfectly fine performance, there is a real productivity benefit for users to know that a given change has zero performance impact. For example, this can avoid the need to write specific benchmarks to reassure reviewers of a change.

3 HEADS AND HEAD SHAPES

To formalize constructor unboxing, we use a simple language of types τ and datatype definitions d. This captures sum types in typical ML-inspired, typed functional programming languages.

---

[3]A   Github   code   search   for   Obj   usage   suggests   https://github.com/rickyvetter/bucklescript/blob/cbc2bd65ce334e1fc83e1c4c5bf1468cfc15e7f9/jscomp/ext/ext_obj.ml#L25 for example.

[4]One should be careful that the cost of boxing is not only the extra allocation, but the reduced memory locality of wider data representations. Locality effects are hard to measure accurately and in particular are not captured well in micro-benchmarks. Still, our prediction remains that unboxing makes a very small difference for most use-cases.

Notation. We write $(e_i)^{i \in I}$ for a family of objects e_i indexed over I. (Placing the domain as a superscript is reminiscent of the exponential notation A^I for function spaces.) We often omit the indexing domain I, writing just $(e_i)^i$. Indexing domains I, J, K, etc. are meta-variables that we understand as denoting finite, totally ordered sets – for example, integer intervals $[0; n]$.

3.1 Types, Datatype Declarations, Values

$$\text{Types} \ni \tau \quad ::= \quad \alpha \mid t \, (\tau_i)^i \mid \widehat{t} \, (\tau_i)^i$$
$$\text{TyDecls} \ni d \quad ::= \quad \text{type } t \, (\alpha_i)^i = (C_j \text{ of } (\tau_{j,k})^{k \in K_j})^j \, (C_u^{\text{unboxed}} \text{ of } \tau_l)^l$$

A datatype definition introduces a datatype constructor t parameterized over the family of type variables $(\alpha_i)^i$ as a sum type made of a (possibly empty) family of boxed constructors $(C_j \text{ of } (\tau_{j,k})^k)^j$, where each C_j expects a family of arguments at types $(\tau_{j,k})^{k \in K_j}$, and a (possibly empty) family of unboxed constructors $(C_l^{\text{unboxed}} \text{ of } \tau_l)^l$ each expecting a single argument of type τ_l.

A type τ is either a type variable α, an instance $t \, (\tau_i)^i$ of a datatype (the $(\tau_i)^i$ instantiate the datatype parameters $(\alpha_i)^i$), or an instance $\widehat{t} \, (\tau_i)^i$ of some primitive type constructor $\widehat{t}$, such as integers, floats, functions, tuples, strings, arrays, custom values, etc.

Closed types (without type variables) in this grammar are inhabited by values v defined by the following grammar, with a simple typing judgment $v : \tau$ expressing that v has type τ.

$$\text{Values} \ni v \quad ::= \quad C \, (v_k)^{k \in K} \mid C^{\text{unboxed}} \, v \mid \widehat{v}$$

$$\frac{\text{type } t \, (\alpha_i)^i = (C_j \text{ of } (\tau_{j,k})^k)^j \, \dots \quad (v_k : \tau_{j,k}[\alpha_i \leftarrow \tau_i']^i)^k}{C_j \, (v_k)^k : t \, (\tau_i')^i} \qquad \frac{\text{type } t \, (\alpha_i)^i = \dots (C_l^{\text{unboxed}} \text{ of } \tau_l)^l \quad v : \tau_l[\alpha_i \leftarrow \tau_i']^i}{C_l^{\text{unboxed}} \, v : t \, (\tau_i')^i}$$

$$\frac{\text{language-specific rules for primitive values at primitive types}}{\widehat{v} : \widehat{t} \, (\tau_i)^i}$$

3.2 Low-Level Representation of Values

Unboxed constructors intrinsically depend on a notion of low-level data representation.

We assume given a set Data of low-level representations, and a function

$$\text{repr} : \text{Value} \times \text{Type} \to \text{Data}$$

that determines the data representation of each value.

We further assume two properties of the repr function:

(1) Injectivity: if $v_1, v_2 : \tau$ have no unboxed constructors and $v_1 \neq v_2$, then $\text{repr}(v_1, \tau) \neq \text{repr}(v_2, \tau)$.
(2) Unboxing: for any C^{unboxed} of τ' in τ and $v : \tau'$, we have

$$\text{repr}(C^{\text{unboxed}} \, v, \tau) = \text{repr}(v, \tau')$$

Our injectivity assumption merely states that our representation was correct before the introduction of constructor unboxing. Our static analysis rejects some unboxed constructor definitions to extend this property to well-typed values with unboxed constructors.

3.2.1 For OCaml. In the specific case of OCaml, writing $\mathbb{Z}_m$ for the set of machine integers, we claim that the representation of values in the reference OCaml implementation can be modeled as:

$$\text{Data}_{\text{OCaml}} \ni w \quad ::= \quad \text{Imm} \, (n \in \mathbb{Z}_m) \mid \text{Block} \, (t \in \mathbb{Z}_m) \, (a_0, \dots, a_{n-1})$$
$$\text{BlockArgs} \ni a \quad ::= \quad (w \in \text{Data}) \mid (n \in \mathbb{Z}_m)$$

As we mentioned in Section 2.1, the low-level representation of OCaml values is either an *immediate* value, which we approximate as living in $\mathbb{Z}_m$, or a *block* starting with a header word containing

a *tag* in $\mathbb{Z}_m$ followed by several words of block arguments $a_0 \ldots a_{n-1}$. Block arguments can be either valid OCaml values themselves or arbitrary machine words. Note that this representation loses some information, for example: immediate values live in a smaller space with one less bit available, and the tag t of a block determines whether its arguments must be valid OCaml values (most tags) or machine words (Custom_tag, String_tag, Double_tag, Double_array_tag, etc.).

We can now define the repr function going from well-typed source values to their low-level representation. In OCaml, constant constructors (taking no argument) are represented as immediates, while non-constant constructors are represented as blocks, and the representation of a constructor in each category depends on its position (indexed starting at 0) in the type declaration.

$$
\begin{array}{rcll}
\text{repr}(C\ \emptyset, \tau) & \stackrel{\text{def}}{=} & \text{Imm } i & C \text{ is } \tau\text{'s } i\text{-th constant constructor} \\
\text{repr}(C\ (v_k)^{k \in K}, \tau) & \stackrel{\text{def}}{=} & \text{Block } i\ (\text{repr}(v_k, \tau_k))^K & C \text{ of } (\tau_k)^k \text{ is } \tau\text{'s } i\text{-th non-constant constructor} \\
\text{repr}(C^{\text{unboxed}}\ v, \tau) & \stackrel{\text{def}}{=} & \text{repr}(v, \tau') & C^{\text{unboxed}} \text{ of } \tau' \text{ is an unboxed constructor of } \tau
\end{array}
$$

As required, this repr function is injective on boxed constructors and erases unboxed constructors. The representations of some primitive values include:

$$
\begin{array}{rcl}
\text{repr}(\text{true}, \text{bool}) & \stackrel{\text{def}}{=} & \text{Imm } 0 \\
\text{repr}((v_1, v_2), (\tau_1 \times \tau_2)) & \stackrel{\text{def}}{=} & \text{Block } 0\ (\text{repr}(v_1, \tau_1), \text{repr}(v_2, \tau_2)) \\
\text{repr}(\textbf{fun } \texttt{x -> x+y}, \tau_1 \rightarrow \tau_2) & \stackrel{\text{def}}{=} & \text{Block } \texttt{Closure\_tag} \ldots \\
\text{repr}(3.14, \text{float}) & \stackrel{\text{def}}{=} & \text{Block } \texttt{Double\_tag} \ldots \\
\text{repr}(\text{"Hello"}, \text{string}) & \stackrel{\text{def}}{=} & \text{Block } \texttt{String\_tag} \ldots
\end{array}
$$

3.3 Heads and Head Shapes

We assume given a set Head of value *heads*. The head h of a value v represents an easily computable abstraction/approximation of the low-level representation of the value v: we assume a function

$$\text{head}_{\text{data}} : \text{Data} \rightarrow \text{Head}$$

computing the head of a value representation, and define

$$\text{head}(v, \tau) \quad \stackrel{\text{def}}{=} \quad \text{head}_{\text{data}}(\text{repr}(v, \tau))$$

Note that if two values have different heads, then they are necessarily distinct.

Our static analysis will run on arbitrary type definitions allowed by our syntax of type declarations, and reject certain type declarations that would introduce conflicts, that is, allow distinct values with the same representation. The cleanest way we found to model this was to define the head shape of a type expression as a *multiset* of heads, which may contain duplicate elements.

NOTATION ($\mathcal{M}(S)$, $M(x)$, $\{\!\{ \ldots \}\!\}$, $\lfloor M \rfloor_{\text{set}}$, $\max(M_1, M_2)$, $M_1 + M_2$). *We write* $\mathcal{M}(S)$ *for the set of multisets of elements of S, $M(x)$ for the number of occurrences of x in the multiset M, $\{\!\{ \ldots \}\!\}$ for multiset comprehension, and $\lfloor M \rfloor_{\text{set}}$ (in $\mathcal{P}(S)$) for the set of elements of a multiset M (in $\mathcal{M}(S)$).*

We use two standard union-like operations on multisets: the maximum and the sum, defined by:

$$\max(M_1, M_2)(x) \stackrel{\text{def}}{=} \max(M_1(x), M_2(x)) \qquad (M_1 + M_2)(x) \stackrel{\text{def}}{=} M_1(x) + M_2(x)$$

We define the *head shape* $\text{headshape}_{\text{ClosedTypes}}(\tau)$ of a closed type expression τ as the multiset of heads of values of this type. We extend this notation to constructor declarations instantiated at a

closed return type, which we call "type components" ϵ as they come up in type declarations.

$$\mathsf{TyComps} \ni \epsilon \quad ::= \quad \tau \mid C \text{ of } (\tau_i)^i : \tau \mid C^{\mathsf{unboxed}} \text{ of } \tau' : \tau$$

$$
\begin{aligned}
\mathsf{headshape}_{\mathsf{ClosedTypes}} &\quad : \quad \mathsf{ClosedTypes} \to \mathcal{M}(\mathsf{Heads}) \\
\mathsf{headshape}_{\mathsf{ClosedTypes}}(\tau) &\quad \overset{\mathsf{def}}{=} \quad \{\!\{ \mathsf{head}(v, \tau) \mid v : \tau \}\!\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{headshape}_{\mathsf{ClosedTyComps}}(C \text{ of } (\tau_i)^i : \tau) &\quad \overset{\mathsf{def}}{=} \quad \{\!\{ \mathsf{head}(C\ (v_i)^i, \tau) \mid (v_i : \tau_i)^i \}\!\} \\
\mathsf{headshape}_{\mathsf{ClosedTyComps}}(C^{\mathsf{unboxed}} \text{ of } \tau' : \tau) &\quad \overset{\mathsf{def}}{=} \quad \mathsf{headshape}_{\mathsf{ClosedTypes}}(\tau')
\end{aligned}
$$

Finally, we can extend the notion of head shapes to *open types* (or type components) containing type variables α, by taking the union of all their closed instances. We write $\tau \uparrow \tau'$ if τ' is a closed type or type component) that instantiates the free type variables of τ.

$$
\begin{array}{c}
\epsilon' \text{ closed} \\
\epsilon' = \epsilon[\alpha_i \leftarrow \tau_i]^i \\
\hline
\epsilon \uparrow \epsilon'
\end{array}
\qquad
\begin{aligned}
\mathsf{headshape}_{\mathsf{TyComps}} &\quad : \quad \mathsf{TyComps} \to \mathcal{M}(\mathsf{Heads}) \\
\mathsf{headshape}_{\mathsf{TyComps}}(\epsilon) &\quad \overset{\mathsf{def}}{=} \quad \max\{\mathsf{headshape}_{\mathsf{ClosedTyComps}}(\epsilon') \mid \epsilon \uparrow \epsilon'\}
\end{aligned}
$$

Note that we take the maximum of all closed instances, not their sum. In particular, if all the closed instances have head shapes that are sets (they do not contain any duplicates), then the headshape of the open type is itself a set. In particular, in absence of unboxed constructors, $\mathsf{headshape}(\alpha)$ is typically equal to the set Heads of all heads seen as a multiset (assuming that each shape is in the image of at least one well-typed value). If we had used a sum in our definition above, then $\mathsf{headshape}(\alpha)$ would have duplicates as soon as two distinct types have heads in common.

3.3.1 For OCaml. In the specific case of OCaml, we define the head of an immediate $n \in \mathbb{Z}_{\mathsf{m}}$ as just the pair (Imm, n), and the head of a block of tag $t \in \mathbb{Z}_{\mathsf{m}}$ as the pair (Block, t).

$$
\mathsf{Heads}_{\mathsf{OCaml}} \overset{\mathsf{def}}{=} \{\mathsf{Imm}, \mathsf{Block}\} \times \mathbb{Z}_{\mathsf{m}}
\qquad
\begin{aligned}
\mathsf{head}_{\mathsf{data},\mathsf{OCaml}}(\mathsf{Imm}\ n) &\overset{\mathsf{def}}{=} (\mathsf{Imm}, n) \\
\mathsf{head}_{\mathsf{data},\mathsf{OCaml}}(\mathsf{Block}\ t\ (a_0, \dots, a_{n-1})) &\overset{\mathsf{def}}{=} (\mathsf{Block}, t)
\end{aligned}
$$

Note that, while our previous choice of Data and repr functions for OCaml model an existing representation, and are not visible to the users – even in presence of unboxed constructors – the definition of heads and the function head : Value → Heads are new design choices that have the user-visible impact of accepting or rejecting certain unboxed constructors in datatype declarations.

We could use a finer-grained notion of head (for example we could include the arity n of a block in its head), which allows to distinguish more types and thus accept more unboxed type definitions.

Conversely, a coarser-grained notion of head would be more portable to other implementations. For example, an implementation that would represent constructors by user-visible name rather than position could not use our notion of head as is. We discuss this portability question in Section 4.2.

Another reason to choose a coarser-grained notion of head is to have a simpler model to explain to users, at the cost of rejecting more declarations; for example, one could restrict unboxing to immediate types by using a pessimistic $\top$ shape for all types containing blocks.

Finally, our implementation defines a concrete syntax of head shapes that denote *sets* of heads and is easy to use in computations. Elements of this *head shape syntax* are pairs of approximations, one for immediates and one for blocks. Approximations are defined as either a finite set of machine words (including in particular the empty set $\emptyset$) or the wildcard shape $\top$ representing all heads.

$$
\begin{aligned}
\mathsf{HeadShapeStx} \ni H &\overset{\mathsf{def}}{=} \mathsf{ImmShape} \times \mathsf{BlockShape} \\
\mathsf{ImmShapes}, \mathsf{BlockShapes} &\overset{\mathsf{def}}{=} \{\top\} \cup \mathsf{FinSet}(\mathbb{Z}_{\mathsf{m}})
\end{aligned}
$$

$$
\begin{array}{llll}
[\![\_]\!] & : & \text{HeadShapeStx} \to \mathcal{P}(\text{Heads}) \qquad & [\![\_]\!] \quad : \quad \text{ImmShapes} \cup \text{BlockShapes} \to \mathcal{P}(\mathbb{Z}_\mathsf{m}) \\
[\![(I, B)]\!] & \overset{\text{def}}{=} & \{\text{Imm } i \mid i \in [\![I]\!]\} & [\![\top]\!] \quad \overset{\text{def}}{=} \quad \mathbb{Z}_\mathsf{m} \\
& & \uplus \{\text{Block } t \mid t \in [\![B]\!]\} & [\![S]\!] \quad \overset{\text{def}}{=} \quad S \qquad\qquad (S \in \text{FinSet}(\mathbb{Z}_\mathsf{m}))
\end{array}
$$

For example, our shape syntax for OCaml integers is $(\top, \emptyset)$, our shape syntax for OCaml booleans is $(\{0, 1\}, \emptyset)$, our shape syntax for lists or options (one constant constructor and one non-constant constructor) is $(\{0\}, \{0\})$, and our shape syntax for custom blocks is $(\emptyset, \{\texttt{Custom\_tag}\})$.

This syntax is a correct abstraction of multisets that happen to be mere sets. It does not let us express multisets with conflicts. We can directly implement the non-disjoint union $H_1 \cup H_2$ of two syntactic shapes, and also implement the disjoint union $H_1 \uplus H_2$ as a partial operation that returns a syntactic shape if H_1, H_2 are disjoint, and is undefined otherwise – if the resulting multiset has duplicates, and cannot be represented as a syntactic shape.

$$
\begin{array}{llll}
(I_1, B_1) \cup (I_2, B_2) & \overset{\text{def}}{=} & (I_1 \cup I_2, B_1 \cup B_2) \\
\top \cup x\,,\ x \cup \top & \overset{\text{def}}{=} & \top
\end{array}
\qquad\qquad
(I_1, B_1) \uplus (I_2, B_2) \quad \overset{\text{def}}{=} \quad (I_1 \uplus I_2, B_1 \uplus B_2)
$$

In the general case, the definition $\text{headshape}(\tau)$ of head shapes for open types may be difficult to compute, as it contains a quantification over all closed extensions of τ. The OCaml value representation is very regular, which makes it easy to compute shapes of type variables, constructor declarations and primitive types. We can represent them directly in our head shape syntax:

$$
\begin{array}{rcll}
\text{headshape}_{\text{OCaml}}(\alpha) & \overset{\text{def}}{=} & (\top, \top) \\
\text{headshape}_{\text{OCaml}}(C \text{ of } \emptyset) & \overset{\text{def}}{=} & (\{i\}, \emptyset) & C \text{ is the } i\text{-th constant constructor at its type} \\
\text{headshape}_{\text{OCaml}}(C \text{ of } (\tau_k)^k) & \overset{\text{def}}{=} & (\emptyset, \{i\}) & I \neq \emptyset,\ C \text{ is the } i\text{-th non-constant constructor at its type} \\
\text{headshape}_{\text{OCaml}}(\widehat{\mathsf{t}}\ (\tau_i)^i) & \overset{\text{def}}{=} & & \text{the immediates and tags of primitive type constructor } \widehat{\mathsf{t}}
\end{array}
$$

This definition of $\text{headshape}_{\text{OCaml}}$ on base types and boxed constructor definitions agrees with the generic definition headshape, in the sense that $\text{headshape}_{\text{TyComps}}(\epsilon) = [\![\text{headshape}_{\text{OCaml}}(\epsilon)]\!]$ for the OCaml value representation and our choice of heads.

3.4 Sum Normal Form

To compute the head shape of a type expression τ, we must unfold type definitions and traverse unboxed constructors. This transformation is of independent interest, we formalize it in this section.

We define a grammar of *sum normal forms* S that capture the result of this unfolding process, and a (partial) normalization judgment $\tau \Rightarrow S$ that computes the head normal form of a type.

$$
\begin{array}{lcl}
S & ::= & \emptyset \mid \eta \mid S + S \\
\eta & ::= & \alpha \mid C \text{ of } (\tau_i)^i \mid \widehat{\mathsf{t}}\ (\tau_i)^i
\end{array}
\qquad
\begin{array}{c}
\text{VAR} \\
\hline
\alpha \Rightarrow \alpha
\end{array}
\qquad
\begin{array}{c}
\text{PRIM} \\
\hline
\widehat{\mathsf{t}}\ (\tau_i)^i \Rightarrow \widehat{\mathsf{t}}\ (\tau_i)^i
\end{array}
$$

$$
\begin{array}{c}
\text{CONSTR} \\
\hline
\text{type } t\ (\alpha_i)^i = (C_j \text{ of } (\tau_{j,k})^k)^j\ (C_l^{\text{unboxed}} \text{ of } \tau_l)^l \qquad (\tau_l[\alpha_i \leftarrow \tau_i]^i \Rightarrow S_l)^l \\
\hline
t\ (\tau_i)^i \Rightarrow \displaystyle\sum_j C_j \text{ of } (\tau_{j,k}[\alpha_i \leftarrow \tau_i]^i)^k + \sum_l S_l
\end{array}
$$

A sum normal form is a multiset of components η written as a formal sum, that are either a boxed constructor, a type variable or a primitive type constructor. The normalization judgment unfolds datatype declarations, sums boxed constructors and the normal form of the unboxed arguments.

In presence of mutually-recursive definitions, some type expressions may "loop" forever: they don't have a sum normal form. Consider for example:

$$\text{type loop} \;=\; \text{Int}^{\text{unboxed}} \text{ of } \widehat{\text{int}} \mid \text{Loop}^{\text{unboxed}} \text{ of loop}$$

Fortunately, the problem of whether a given type expression τ has a sum normal form is in fact decidable. We discuss our decision procedure in Section 6.

3.5 Rejecting Conflicts

Finally, given a type declaration d, our static analysis computes a head shape $\text{headshape}(S)$ of its sum normal form S by summing the head shape of each component of the sum. Our static analysis accepts the definition if and only if $\text{headshape}(S)$ does not contain any duplicates.

$$\frac{t\;(\alpha_i)^i \Rightarrow S}{\text{headshape}_{\text{decl}}(\text{type } t\;(\alpha_i)^i \;=\; \dots) \overset{\text{def}}{=} \text{headshape}_{\text{snf}}(S)}$$

$$\text{headshape}_{\text{snf}}(\emptyset) \quad\overset{\text{def}}{=}\quad \emptyset$$

$$\text{headshape}_{\text{snf}}(\eta) \quad\overset{\text{def}}{=}\quad \text{headshape}_{\text{TyComps}}(\eta)$$

$$\text{headshape}_{\text{snf}}(S_1 + S_2) \quad\overset{\text{def}}{=}\quad \text{headshape}_{\text{snf}}(S_1) + \text{headshape}_{\text{snf}}(S_2)$$

The result of this analysis can be easily computed, in the case of OCaml, by using our head shape syntax: the head shape of $S_1 + S_2$ is conflict-free if $\text{headshape}_{\text{OCaml}}(S_1) \uplus \text{headshape}_{\text{OCaml}}(S_2)$ is defined and has a conflict otherwise.

3.6 Pattern-Matching Compilation

When checking a type declaration with unboxed constructors, we record for each unboxed constructor C^{unboxed} of τ the head shape of its type parameter τ. When compiling pattern-matching clauses using an unboxed constructor in a pattern, say $C^{\text{unboxed}}p$, we implement matching on C^{unboxed} of τ as a condition that the head of the scrutinee must belong to the shape of τ. (The details of how to do this depends of course on the pattern-matching compilation algorithm of the language.)

We know that this approach is always sound, thanks to the property that none of the other scrutinees (starting, at the source level, with a different constructor) may have a head belonging to the head shape of τ. Note that this property, enforced by our static analysis, is in fact slightly stronger than the absence of conflicts: not only must the representation of inhabitants of τ be distinct from all the other possible scrutinees, they should furthermore have distinct heads.

The runtime cost of checking the head depends on the language and the notion of head chosen. For our choice of heads for OCaml, it is exactly as costly as checking the head constructor of a value, so this does not add overhead on pattern-matching. A finer-grained notion of head that would inspect the value "in depth" could add a higher cost – to balance against the space savings of accepting more unboxing.

4 SCALING TO A FULL LANGUAGE

In this section, we describe less formally all the "other issues" that we had to consider to scale this proposed feature to a full programming language, namely OCaml.

4.1 Handling All the Tricky Cases

We did not encounter any conceptual issue when scaling this approach to all the primitive types supported by the OCaml runtime. (This is not too surprising given that our heads are closely modeled on the existing runtime data representation.) In the interest of demonstrating the difference between

the simple situation of datatypes with a simple representation and everything else, let us give here an *exhaustive* list of all the tricky cases.

(1) `Double_array_tag` is used to represent nominal records whose fields are all `float`, and also values of type `float array`.

To determine the shape of a record type, we must call the same logic that the type-checker uses to the decide the "unboxed float record" criterion and use $(\emptyset, \{\text{Double\_array\_tag}\})$ instead of $(\emptyset, \{0\})$ in that case.

For arrays, we use the head shape $(\emptyset, \{0, \text{Double\_array\_tag}\})$ in all cases. Note that OCaml supports a configuration option to disable the unboxing of float array (supporting this representation adds some dynamic checks on array operations), but we decided to use the pessimistic shape with both tag values independently of the configuration value, to avoid having the compiler statically reject some programs only in some specific configuration.

(2) Values of type `ty Lazy.t` have an optimized representation where they may be represented by a lazy thunk of tag `Lazy_tag`, a computed value of tag `Forward_tag`, *or* directly a value of type `ty`, under some conditions on `ty`. The corresponding shape is the maximum of $(\emptyset, \{\text{Lazy\_tag}, \text{Forcing\_tag}, \text{Forward\_tag}\})$ and of the shape of `ty`. (OCaml 5 added a third tag `Forcing_tag` to detect concurrent forcing; it was trivial to adapt our analysis.)

Note that Section 3.3.1 defined $\text{headshape}_{\text{OCaml}} \widehat{t} \, (\tau_i)^i$ as depending only on $\widehat{t}$, not the τ_i; here we are handling lazy values in a more precise way due to their non-uniform representation. (We could also approximate them to the uniform shape $\top$.)

(3) Function closures may have either tags `Closure_tag` or `Infix_tag` (used for some mutually-recursive functions). The pattern-matching code that we generate on sum types with an unboxed function type (which is useful for Coq `native_compute`) is slightly less good than it could be because these two tags are not consecutive (247 and 249), so we are slightly tempted to renumber the tags in the future.

(4) Exceptions, and in general inhabitants of extensible sum types, use tag `Object_tag`. Object types themselves have an obscure and complex data representation due to various optimizations, and we just assigned them the top shape $\top$.

4.2 Portability of our Heads

The language of head shapes makes some aspects of the low-level representation of values visible to users of the surface language. This comes at the risk of complexity, but also at the risk of reducing the portability of the language by setting in stone certain representation choices, that would rule out other implementations.

One could think of making constructor-unboxing a "best effort" feature to avoid this downside, by simply emitting a warning in the case where an unboxing annotation would introduce a conflict under the current implementation, and keeping the constructor boxed. We decided against this for now, because we believe that advanced performance features such as constructor unboxing are used when users reason about the performance of their application, that is, when data representation is part of their specification for the code they are writing. In this context, silently ignoring representation requests is arguably a bug: it breaks the specification the user has in mind.

Instead we are trying to discuss with other implementors of OCaml to find whether we should make our heads more coarse-grained in some places, to increase portability without breaking relevant examples of interest. In particular, the alternative backend `js_of_ocaml` compiles OCaml to JavaScript, and uses native JavaScript numbers for most OCaml numeric types (`int`, `float`, `nativeint`, `int32`). We are planning to quotient the difference between those types in our language of shape, to improve portability.

4.3   Abstract Types with Shapes

Abstract types can readily be given the top shape ⊤. We also support annotating an abstract type with a shape restriction [@shape ..], which gives a head shape for this type.

For abstract types coming from the interface of modules or functor parameters, those shape annotations are checked when checking that the interface conforms to the implementation.

For abstract types used to represent values only populated by the FFI, these shape annotations have to be trusted, in the same way that the OCaml FFI trusts foreign functions to respect their type provided on the OCaml side.

We used this feature in our Zarith example in Section 2.3 to allow unboxing the Big constructor, whose argument is an abstract type custom_gmp_t of GMP numbers implemented through the FFI.

4.4   Are We Really First-Order?

Parametrized type definitions fall in the first-order fragment because OCaml does not support higher-kinded types.

Note that some designs for higher-kinded types in related languages are restricted to higher-order "type constructors" that do not create β-redexes, so they do not necessarily have the expressiveness of the full higher-order λ-calculus.

On the other hand, the OCaml module system does provide higher-order abstractions through functors: a type in a functor may depend on a parametrized type in the functor argument. However, unfolding of type definitions remain first-order in nature:

- When we are checking the body of a functor and encounter a type that belongs to a module parameter, it is handled as any other type declaration.
- When we encounter a type expression containing a functor application, e.g. Set.Make(Int).t, the type-checker has access to the signature of the functor application Set.Make(Int) and we check its type t.

Another way to think of the treatment of functor application is that the OCaml type-checker performed β-reduction of functor applications before we compute shapes. In other words, in this work, we consider the module language as a strongly-normalizing higher-order subset whose normal forms are first-order.

4.5   OCaml Features Subsumed by Head Shapes

The OCaml type-checker currently contains three subsystems that we believe would be subsumed by our head shape analysis:

(1) It contains an analysis of the "unboxed form" of a type (due to the presence of unboxing for single-constructor variants and single-field records) that corresponds to our notion of sum normal form of a type, and would benefit from our normalizing algorithm to compute those in presence of recursion.

(2) It defines a property called [@@immediate] for abstract types, which claims that the inhabitant of the type are all immediate values. (This is used by the runtime to specialize ad-hoc polymorphic functions such as comparison and serialization.) Computing the head shape subsumes immediateness-checking.

(3) To implement unboxing of single-variant GADTs, it must perform an intricate static analysis to reject attempts to unbox existentials, which would make the float array optimization unsound. (Don't ask.) We believe that this static analysis, detailed in Colin, Lepigre and Scherer [2019], could be subsumed by our head shape computation.

We are planning to simplify the compiler implementation by removing all the existing logic to implement these separate aspects, and replace them by a shape computation.

5 POTENTIAL EXTENSIONS

We have considered the following aspects, but have not implemented them. They are not necessary to consider upstreaming a first useful version of constructor unboxing.

5.1 Shape Constraints on Type Variables

Section 4.3 shows how abstract types can now be annotated with constraints on the head shapes of their inhabitants. A related feature would be to constrain the type variables of parametrized types:

```
type ('a [@shape any_block]) block_option = None | Some of 'a [@unboxed]
type ('a [@shape immediate]) imm_option = None of unit | Some of 'a [@unboxed]
```

The type 'a block_option is similar to the standard 'a option datatype, but its type parameter 'a may only be instantiated with type expressions whose shape is included in the shape any_block – any block tag, but no immediate value. This restriction allows unboxing the Some constructor without conflicting with the None immediate value. Conversely, imm_option may only be instantiated with immediate types, and its Some constructor can also be unboxed as we made None of unit a block constructor.

Similarly to abstract types with shapes (Section 4.3), this feature provides modularity. We can construct large types with specialized representations by composing together smaller parameterized types (or functors), with shape assumptions on the boundaries between the various definitions.

This change is easy conceptually, but requires non-trivial changes to the OCaml compiler where type variables do not carry kind information. It goes in the same direction as other "layout" changes experimented with by Jane Street, so some of the implementation work can be shared.

5.2 Harmless Cycles

Our algorithm to compute shapes unfolds potentially-recursive type definitions and monitors termination: it stops when encountering a cycle in the definition. Currently our prototype rejects all definition that contain such cycles. But the cycles fall in two categories: most cycles are "harmful cycles" that must be rejected, but there are "harmless cycles" that could be accepted.

```
type harmful = A | Loop of harmful [@unboxed]
type harmless = Loop of harmless [@unboxed]
```

Both those examples are rejected by our prototype as their shape computation detects a cycle. harmful cannot soundly be accepted, as there would be a confusion between the values A, Loop(A), Loop(Loop(A)), etc. On the other hand, allowing the unboxing of harmless would not in fact introduce any confusion as the type would be empty – without any inhabitant. Said otherwise, cycles in shape computations can be interpreted as smallest fixpoints; most of those fixpoints contain conflicts but a few are the empty set of value.

Accepting harmless cycles should be of medium difficulty. From an implementation perspective it is not easy to distinguish harmless cycles and accept them, it is substantially more work than rejecting all cycles. Besides, there is no point in writing types such as harmless in practice – just write an empty type directly. So this is naturally left as future work.

There is however one good reason to do more work there, which is related to data abstraction. Consider the following example:

```
type 'a foo                    type weird = Loop of weird foo [@unboxed]
```

This weird definition is accepted by our shape analysis. For the abstract type 'a foo we assume the shape ⊤ of any possible value – this does not depend on the parameter 'a. Then weird foo has the same shape ⊤ and the definition of weird is accepted. (Adding any other constructor to weird would make it rejected.)

However, we could later learn that the type `'a foo` is in fact defined as **type** `'a foo = 'a`. If we perform the substitution, we get a harmless cycle. In other words, rejecting harmless cycles breaks the substituability property for abstract types. This is a nice meta-theoretical property, and breaking it may result in surprising software engineering situations that are problematic in practice.

5.3 Unboxing by Transformation

In our work, unboxed constructors act as the identity on the representation of their arguments. One could generalize this by allowing constructor unboxing to be realized by a non-identity transformation on its arguments – chosen to be more efficient than the default constructor representation. Applying non-identity transformations could avoid conflicts in value representations, allowing more unboxing requests. Consider for example:

```
type 'a t = A of bool | B of 'a option
```

Our work only supports unboxing the constructor A in this example. Unboxing B is not supported:

- if A is not unboxed, then we would have a confusion between blocks constructed by A and those coming from the Some constructor of the option.
- if A is unboxed, then we would have a confusion between immediates corresponding to the **false** value (in A) and the None value (in B).

It would however be possible to unbox the constructor B if we accepted to change the representation of the constructor A **of** bool [@unboxed]. Instead of storing a bool value directly (an immediate in $\{0, 1\}$), we could transform the bool value to store it as an immediate in $\{1, 2\}$ for example, avoiding a conflict with the None value (immediate 0) unboxed from B. Examples of representative approaches follow.

```
(* no shifting, but a different boolean type *)
type 'a t1 =                          and fake_bool =
  | A of fake_bool [@unboxed]           | Fake_false [@tag 1]
  | B of 'a option [@unboxed]           | Fake_true [@tag 2]
```

The type `t1` is in fact not an example of a transformation associated with an unboxed constructor: instead we assume that it is possible to specify a non-standard choice of tag at declaration time – the imaginary [@tag 2] attribute. Supporting this would be an easy change, but it requires using a non-standard boolean type and thus requires code changes for the user, making it cumbersome or impractical in many situations.

```
  (* explicit shifting *)                (* fully inferred transformation *)
  type 'a t2 =                           type 'a t3 =
    | A of bool [@unboxed by (add 1)]      | A of bool [@unboxed]
    | B of 'a option [@unboxed]            | B of 'a option [@unboxed]
```

The type `t2` performs a transformation at unboxing time (adding 2 to the value) that is specified by the user. Pattern-matching code would then have to be careful to undo this transformation on the fly (by subtracting 1). Note that (add 1) is not an arbitrary OCaml term here, it must be part of a dedicated transformation DSL that we know to invert efficiently.

Another valid choice would be to have the constructor B transform its argument in a way that leaves its block values unchanged but shifts its None value from the immediate 0 to an immediate outside $\{0, 1\}$ or to a block of non-zero tag (with no argument). Note that unboxing B would come at a higher runtime cost as the transformation (to apply at construction time and unapply at matching time) is more complex.

Finally the type `t3` assumes a version of constructor unboxing that implicitly infers such transformations to satisfy the user's unboxing request. This is not the design approach that we have

used in our OCaml work, but it corresponds to unboxing strategies in some other programming languages – see our discussion of Rust's niche-filling optimizations in our Related Work Section 8.2.

Arbitrary transformations can be supported, as long as we can express the corresponding abstract transformation on shapes. For OCaml and the notion of heads that we proposed, a natural space of transformations are those that would change the head of a value, and leave the rest unchanged, by:

(1) applying a mapping to its immediate values, and/or turning some of them into blocks (constant blocks with fixed tags, or non-constant blocks with the transformed immediate as argument)
(2) applying a mapping to the tag of its blocks, leaving its arity and arguments unchanged

The set of transformations of interest is also constrained by performance considerations. In particular, turning an immediate into a non-constant block requires an allocation and memory indirection (constant blocks can be preallocated), which is precisely what we wanted to avoid by unboxing the constructor. It may still be beneficial if this transformation occurs only for some inputs that are rare in practice, with all other cases unboxed.

Supporting tag choice requests as in t1 should be easy; user-specified transformations as in t2 would be of medium difficulty, depending on the expressiveness of the transformations. We are not planning to work on full transformation inference in the context of OCaml.

5.4 Using Unboxing to Describe Existing Representation Tricks

Some subtle data-representation choices of the OCaml compiler and runtime, mentioned in Section 4.1, could in fact be presented as unboxing, possibly with further extensions.

Flat float arrays. OCaml arrays use a uniform representation (using tag 0) except for arrays of elements represented as floats, which have the tag Double_array_tag and a custom representation. The OCaml runtime (written in C) checks the array tag on each low-level operation to determine how to access the array. OCaml cannot currently express the type of custom arrays of float-represented values, but our proposed shape annotations (Section 5.1) would make it possible to do so:

```
type ('a [@shape double]) double_array [@@shape double_array]
```

With constructor unboxing we can then express the array representation trick in safe OCaml code:

```
type 'a array =
  | Any : 'a generic_array -> 'a array
  | Double : ('d [@shape double]). 'd double_array -> 'd array
```

The type 'a generic_array is a type that does not exist in OCaml today, of uniform arrays with tag 0. The constraint 'd [@shape double] indicates that (in our proposed OCaml extension) matching a value of type 'd array with the Double constructor reveals that 'd is represented as double. This definition would suffice for defining array-accessing functions in pure OCaml, but for array creation the runtime checks dynamically if its argument is a float. Implementing the check in OCaml would require exposing an extra (inelegant and non-parametric, but safe) primitive:

```
type 'a double_check =
  | IsAny : 'a double_check
  | IsDouble : ('d [@shape double]). 'd double_check
val check_if_double : 'a -> 'a double_check
```

Lazy forwarding. The representation of a lazy value may be a block of tag Lazy_tag, for a thunk that has not yet been evaluated to a result, or a block of tag Forward_tag storing a result, or sometimes this resulting value directly. The OCaml runtime sometimes "shortcuts" forward blocks when they are moved around, replacing them by their value directly, with a dynamic check that this does not introduce an ambiguity – the value should not itself have tag Lazy_tag or Forward_tag.

Let us try to express this shortcutting trick in OCaml rather than in the runtime code in C. Let us assume an imaginary `[@unboxed unsafe]` attribute that does not perform any static confusion check for this constructor – we intentionally do not provide this in our current design proposal, which focuses on safe uses. We could then write:

```
type 'a lazy_state =                                    and 'a lazy = 'a lazy_state ref
| Thunk of (unit -> 'a) [@tag lazy_tag]
| Forward of 'a      [@tag forward_tag]
| Forward_unboxed of 'a [@unboxed unsafe]


let make_forward (v : 'a) : 'a lazy_state =
  if List.mem (Obj.tag (Obj.repr v)) Obj.[lazy_tag; forward_tag; double_tag]
  then Forward v
  else Forward_unboxed v
```

One could even think of an imaginary `[@unboxed dynamic]` variant where the compiler is in charge of inserting this dynamic check:

```
type 'a lazy_state =
| Thunk of (unit -> 'a) [@tag lazy_tag]
| Forward of 'a      [@tag forward_tag] [@unboxed dynamic]
```

6 OUR HALTING PROBLEM

In Section 3.4, we mention that computing the head shape of a type requires unfolding datatype definitions, and that this unfolding process may not terminate in presence of mutually-recursive datatype definitions.

In the present section, we discuss this problem in more detail, and present a novel algorithm to normalize safely in presence of recursion: it either returns the sum normal form of a type, or reports (in a finite amount of time) that the definition loops and no sum normal form exists.

First, we remark that this problem corresponds to the *halting problem* for a specific fragment of the pure λ-calculus (just function types, no products, booleans, natural numbers etc.), namely the first-order fragment with arbitrary recursion. Consider the following example:

```
                                                        type handle =
  type 'a id = 'a                                       | By_number of int id [@unboxed]
  type name = Name of string [@unboxed]                 | By_name of name [@unboxed]
                                                        | Opaque of string
```

It can be rephrased as a λ-term with recursive definitions as follows:

```
let rec id(a) = a
and name = string
and handle = sum (id int) (sum name (box string))
```

In this translation, we use a free variable `sum` as a binary operator to separate constructor cases, and a free variable `box` over the translation of type expressions appearing under a constructor. (Other free variables encode primitive types.) The sum normal form of any type in the definition environment above can be read back from a normal form of the translated λ-terms in presence of the recursive definitions. (More precisely, we only need a "weak" normal form that does not reduce under box applications.)

The algorithm that we present in this section decides the halting problem for the first-order pure λ-calculus with recursive definitions, also called "order-1 recursive program schemes" in the literature,

with an arbitrary reduction strategy. This is not unreasonable, given that the halting problem for this fragment is already known to be decidable, as demonstrated for example in Khasidashvil [2020] in the first-order case and in Plotkin [2022]; Salvati and Walukiewicz [2015] (for example) in the more general setting of the pure simply-typed λ-calculus with arbitrary recursive definitions!

6.1 On-The-Fly rather than Global Termination Checking

The normalization arguments in previous work on recursive program schemes are *global* in nature; they reason by normalizing all mutually-recursive definitions at once [Khasidashvil 2020], or at least they compute a termination bound that depends on the size of the whole mutually-recursive system [Plotkin 2022].

In our setting, the set of mutually-recursive definitions potentially contains large type definitions in scope, that are either explicitly mutually-recursive, or depend on each other through recursive modules. Normalizing datatype definitions without unboxed constructors is immediate as they are their own normal forms, but we also have to normalize through OCaml type *abbreviations* which are widely used. (We have not included type abbreviations in our Section 3 as the abbreviation **type** t = τ can be understood in this context as syntactic sugar for **type** t = Abbrev^unboxed **of** τ.) Expanding all abbreviations is also known to potentially generate very large structural types for some use-cases, so the OCaml type checker uses careful memoization to only expand on-demand during type inference.

Another issue with a global termination analysis is that OCaml type definitions *change often* due to functor applications.[5] Some type definitions in a functor body rely on abstract (or concrete) types from the functor argument. When the functor is applied to a module parameter, we get a new instance of those definitions where previously abstract type constructors from the formal argument are concrete, which may even introduce new recursive dependencies. Any global termination analysis done on all type definitions would thus have to be partially recomputed on functor applications – and delimiting the part of the computations to rerun may not be obvious in presence of recursive dependencies.

Performing a global termination analysis thus runs the risk of large computational costs in practice, which is all the more frustrating that we expect unboxed constructors themselves to be rarely used, being an advanced feature. It should come at no cost when not used and at little cost when used sparingly.

Instead, we propose an *on-the-fly* termination checking algorithm. Without any static precomputation on the set of mutually-recursive definitions (which may be large and/or change often during type checking), our algorithm takes a term and monitors its reduction sequence: it maintains some information on the side that is updated during reduction, and may "block" the reduction if it detects that it is about to loop forever. We must provide the following guarantees:

- Correctness: reduction sequences that are never blocked by the termination monitor are always finite; they cannot diverge.
- Completeness: if a reduction sequence is blocked by the termination monitor, then it would have diverged in absence of monitoring.

The existence of a termination monitor that is sound and complete implies decidability of the halting problem for the reduction being considered. We have not found termination results in the existing literature whose proofs would suggest this termination-monitoring approach; to the best of our knowledge, this approach is novel for the pure first-order λ-calculus with recursive definitions. (But the literature on term rewriting systems is vast and our knowledge of it very partial.)

---

[5]Functor in the ML-family sense of a module (possibly carrying type components) parameterized by another module.

We consider this as a notable contribution of our work whose interest is independent of constructor unboxing. Already in the OCaml compiler, there are other parts of the type checker that need to normalize type definitions (including unboxed single-constructor datatypes), and rely on the unprincipled approach of passing a fixed amount of "fuel" and failing with an error once it is exhausted. We plan to rewrite these computations using on-the-fly normalization checking. We hope that other language implementors could use this approach to work with recursive type definitions, and there may be other use-cases thanks to the generality of the language considered.

Remark: our termination-monitoring approach means that we are only adding some bookkeeping logic to a head-shape computation that we need to do anyway, that happens rarely (single-constructor unboxing is a rarely used, opt-in feature), and whose cost is bounded by a very small constant in practice, the depth of type definitions that need to be unfolded to compute the head shape (at most 5 for reasonable OCaml programs). In particular, we know with certainty that head shape computations will add no noticeable compile-time overhead to the compilation process of real-world OCaml programs.

6.2 Intuition

Attempt 1: detect repetition of whole terms. A first idea to prevent non-termination is to perform a simple cycle detection: block the reduction sequence if we encounter a term that was already part of the reduction sequence. This approach is obviously complete: we have found a cycle that can diverge, but it is not correct in presence of "non-regular recursion", that can generate infinitely many distinct terms. Consider for example (in λ-calculus syntax):

```
let rec loop(a) = loop(list a) in loop(int)
```

In this environment `loop(int)` reduces to `loop(list int)`, then `loop(list (list int))`, etc., without ever repeating the same term or even the same subterm in reducible position.

Attempt 2: detect repetition of head functions. The second idea is that, if detecting repeating of whole terms is too coarse-grained, we should instead track repetitions of the *head function* of the term, in the usual sense of the topmost function/rule/constructor in reducible position. This approach would prevent the infinite reduction sequence for `loop int` in the example above, by blocking at the second redex with the same head `loop`. It is easy to show that it is sound for termination, given that the number of distinct heads is finite. However, this approach is incomplete, it blocks reduction sequences that would have normalized, for example:

```
let rec id(a) = a in id (id int)
```

This reduction sequence needs to reduce the function `id` twice before reaching a normal form.

Solution: trace head functions for each subexpression. Our solution is a refinement of detecting repetition of heads. Instead of tracking the heads that have been expanded in the whole term, we trace a *different* set of heads for each subterm, corresponding to the set of function heads whose expansions were necessary to have the subterm appear in the term.

In the first example above, we start with the term `loop int` where all subterms are annotated with the empty trace (no expansion happened). We can write this as `[]loop []int`: the subterms `loop int` and `int` are both in the empty trace. The first step of the reduction results in the annotated term `[loop]loop ([loop]list []int)`: the subterm `[]int` was unchanged by the expansion, but the surrounding context `loop (list □)` *appeared* in the reduction of `loop` in the empty trace, so this part of the new term gets annotated with `[loop]`. At this point, our algorithm blocks the redex `loop (list int)` as it is already annotated with the function head `[loop]`.

In the second example, `[]id ([]id []int)` reduces to its argument `[]id []int` unchanged – this argument was already present in the term before the expansions, it did not *appear* during the reduction. `[]id []int` can in turn reduce to `[]int` which is an (annotated) normal form.

6.3 Formalizing our Algorithm

We start from a grammar for programs p in the first-order λ-calculus with recursive definitions, containing in particular terms t, and introduce a distinct category of *annotated* terms $\bar{t}$ whose function-call subterms carry an expansion trace l (a list of function names without duplicates).

$$
\begin{array}{llll}
p & ::= & \text{let rec } D \text{ in } t & \text{recursive programs} \\
D & ::= & \emptyset \mid D, f(x_i)^i = t & \text{function definitions} \\
f & & & \text{function name} \\
x & & & \text{first-order variable} \\
t & ::= & x \mid f(t_i)^i & \text{term}
\end{array}
$$

$$
\begin{array}{llll}
\bar{p} & ::= & \text{let rec } D \text{ in } \bar{t} & \text{annotated programs} \\
\bar{t} & ::= & x \mid f(\bar{t}_i)^i \ @ \ l & \text{annotated term} \\
l & ::= & \emptyset \mid l, f & \text{(we require } f \notin l) \\
C[\Box] & ::= & \Box \mid f((\bar{t}_i)^i, C[\Box], (\bar{t}_j)^j) \ @ \ l & \text{annotated reduction context}
\end{array}
$$

We extend the usual notion of β-reduction $t \rightsquigarrow t'$ to annotated terms. Expanding a function call $f(\bar{t}_i)^i$ is only possible if its trace does not already contain f – otherwise this term is stuck, we call it a *blocked redex*. The arguments $(\bar{t}_i)^i$ are annotated terms, but the body t' of the definition of the function f is a non-annotated term: the body and its subterms *appear* in the reduction sequence at this point, and we use an *annotating* substitution $t'[\sigma]^{@l}$ to annotate them, where σ is a substitution from variables to annotated terms and l is the trace to use to annotate new subterms.

Definition 6.1 ($\bar{p} \rightsquigarrow \bar{p}'$, $t[\sigma]^{@l}$).

$$
\frac{(f(x_i)^i = t') \in D \qquad f \notin l}{\text{let rec } D \text{ in } C[f(\bar{t}_i)^i \ @ \ l] \rightsquigarrow \text{let rec } D \text{ in } C[t'[(x_i \leftarrow \bar{t}_i)^i]^{@l,f}]}
$$

$$
\begin{array}{lll}
x[\sigma]^{@l} & = & \sigma(x) \\
f(t_i)^i[\sigma]^{@l} & = & f(t_i[\sigma]^{@l})^i \ @ \ l
\end{array}
$$

Note that the annotating substitution $t[\emptyset]^{@l}$ annotates each function call of an unannotated term t with the trace l.

NOTATION ($\lfloor \bar{t} \rfloor$). *Let us write $\lfloor \bar{t} \rfloor$ (or $\lfloor \bar{p} \rfloor$) for the unannotated term (or program) obtained from erasing all annotations from $\bar{t}$ (or $\bar{p}$).*

We are only interested in annotated terms that were obtained starting from an initial annotated term with all traces empty. Outside this subset of annotated terms there are terms with weird/impossible annotations (for example annotation of functions that do not exist in the recursive environments) that we sometimes want to rule out from our statements.

Definition 6.2 (Reachable annotated term). An annotated term $\bar{t}$ is *reachable* if it occurs as the subterm of a term in a reduction sequence starting from an initial program of the form let rec D in $t[\emptyset]^{@\emptyset}$.

6.4 A Sketch of Correctness (Termination)

Due to space limitations, we moved our proofs of correctness and completeness for this annotated reduction algorithm to an appendix in our extended version. It proves the following results.

LEMMA. *If $\bar{t}$ is reachable and reduces, in the annotated system, to a β-normal form $\bar{v}$, then $\lfloor \bar{v} \rfloor$ is the β-normal form of $\lfloor \bar{t} \rfloor$.*

THEOREM (CORRECTNESS). *Annotated reduction is strongly normalizing: it either reduces to a β-normal form or reduces to a blocked redex in a finite number of reduction steps.*

THEOREM (COMPLETENESS). *If an annotated program $\bar{t}$ contains a blocked redex, then its unanno-tated erasure $\lfloor \bar{t} \rfloor$ admits an infinite reduction sequence.*

In this section we will merely sketch our termination argument.

The general approach is to use a termination measure. We first define a measure, that is, a function from our terms into a well-ordered set – a set with an order relation, such that there do not exist infinite strictly-decreasing sequences. Then we prove that the our annotated reduction strictly decreases the measure of terms.

We can first define a measure on our traces l. There is a finite set of type declarations in our system, and a trace can contain each type constructor at most one, so there is a largest possible trace L that contains all type constructors. An application annotated with this trace cannot be reduced. Any trace l can then be measured by the length difference $\text{length}(L) - \text{length}(l)$, a natural number.

Then the question is how to extend this measure on traces into a measure on annotated terms. One could think of using the (measure of) the trace of the head application, but this does not decrease during reduction if a subterm contains a larger trace and ends up in head position. (Note: we state correctness for any reduction strategy on annotated terms, not necessarily head reduction.)

A second idea is to measure a term $\bar{t}$ by the set of (measures of) the traces that occur inside it – technically the multiset of traces, using the [multiset ordering](#). But this is not decreasing either: when we reduce an application $f(\bar{t}_i)^i \ @ \ l$, we may duplicate its arguments t_i arbitrarily many times, and those may contain traces that are strictly larger than l, resulting in a larger overall measure for the reduced term.

The trick to make the proof work, which was suggested to us by Irène Waldspurger, is to use multisets of multisets: we measure each subterm in our annotated term by the *path* from this subterm to the root of the term, seen as a multiset of traces of applications. And then we measure our term by the multiset of measures of its subterms.

For example, for the term $f(g \ @ \ l_g, h \ @ \ l_h) \ @ \ l_f$, the measure of the root subterm is $\{\!\{l_f\}\!\}$ (we use double braces for multisets rather than sets), the measure of the g subterm is $\{\!\{l_f, l_g\}\!\}$, and the measure of the h subterm is $\{\!\{l_f, l_h\}\!\}$, so the measure of the whole term is $\{\!\{\{\!\{l_f\}\!\}, \{\!\{l_f, l_g\}\!\}, \{\!\{l_f, l_h\}\!\}\}\!\}$.

Consider now a subterm $\bar{u}$ of an argument of the redex $f(\bar{t}_i)^i \ @ \ l$. The measure of its head application may be larger than the measure of l, but its measure as a subterm is the *path* to the root, which contains l. When the application of f gets replaced by new subterms of a strictly smaller trace f, l, then the path from $\bar{u}$ to the root will change, f gets replaced by these new nodes in the path, so the path measure of $\bar{u}$ decreases strictly. This works even if this subterm $\bar{u}$ gets duplicated by expansion: we get several copies, but at a strictly smaller measure, so we are still decreasing for the multiset measure.

For example, if $f(g \ @ \ l_g, h \ @ \ l_h) \ @ \ l_f$ reduces into $f'(g \ @ \ l_g, g \ @ \ l_g) \ @ \ (l_f, f)$, the subterm h has been erased and the subterm g has been duplicated; its measure is now $\{\!\{(l_f, f), l_g\}\!\}$, which is strictly smaller than its previous measure $\{\!\{l_f, l_g\}\!\}$. The measure of the whole term changed from $\{\!\{\{\!\{l_f\}\!\}, \{\!\{l_f, l_g\}\!\}, \{\!\{l_f, l_h\}\!\}\}\!\}$ to the strictly smaller $\{\!\{\{\!\{(l_f, f)\}\!\}, \{\!\{(l_f, f), l_g\}\!\}, \{\!\{(l_f, f), l_g\}\!\}\}\!\}$.

7 ON CPP

Our on-the-fly termination checking algorithm is related to the macro expansion algorithm of the cpp preprocessor for C, as presented in the C11 standard. Quoting the standard:

> 6.10.3.4 (2) If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

In this context, the "replacement list" denotes the body of a function-like macro definition – here we only consider function-like macros, **#define** FOO(..) ... rather than **#define** FOO .... The standard explains that a macro must not be expanded from its own definition or from a "nested replacement". This corresponds to our idea of blocking redexes whose function name occur in their own trace. The idea that this non-replacement information remains active "in later contexts" corresponds to the idea of carrying annotations around in subterms as the computation proceeds.

The phrasing of the standard is not very clear! In the 1980s, Dave Prosser worked on a strategy to ensure that macro replacement always terminates by disallowing dangerous cyclic/recursive macros, and wrote a careful algorithm to allow as much replacement as would be possible without – hopefully – endangering non-termination. This algorithm was published as pseudo-code in a technical note Prosser [1986]. The C89 standard committee then translated Dave Prosser's pseudo-code into the obscure prose that became the standard text.

We know about David Prosser's pseudo-code today thanks to Diomidis Spinellis who spent "five years trying to implement a fully conforming C preprocessor"; Spinellis wrote an annotated version of Prosser's pseudo-code that explains the code: Spinellis [2008].

Prosser's algorithm is strongly related to our termination-monitoring algorithm – it was not an inspiration as we were unfortunately unaware of the connection when designing our own. We are not aware of any proof of correctness for Prosser's algorithm – that it does guarantee termination.

In this section, we will compare the two algorithms. We will only consider the "core" fragment of cpp macros, consisting of the function-like macros that use their formal parameters directly – no conditionals, no use of stringization or concatenation, etc.

7.1 First-Order and Closed-Higher-Order Function Macros

The cpp preprocessor performs β-reduction, but also parsing: it starts from a linear sequence of tokens instead of an abstract syntax tree. It is difficult to reason at the level of sequences of tokens, in particular about macros that generate unbalanced sequences of parentheses; we will not attempt to do so here. Let us only consider (core) macros whose terms are well-parenthesized. What is their expressivity in terms of a λ-calculus?

We define the *first-order fragment* of core macros as the fragment where all bound occurrences of a macro name foo are syntactically an application foo(...) – foo is immediately followed by well-bracketed parentheses.

This is the fragment that the vast majority of C programmers use. But it is possible to write (well-parenthesized) macros outside that fragment, whose reduction behavior is less clear. We will mention two examples, which we call the NIL example and the a(a) example:

```
#define NIL(xxx) xxx
#define G0(arg) NIL(G1)(arg)
#define G1(arg) NIL(arg)


G0(42) // ⤳ NIL(G1)(42) ⤳ G1(42)
       // ⤳ NIL(42) ⤳ 42
```

```
#define a(x) b
#define b(x) x


a(a)(a)(a) // ⤳ b(a)(a)
           //  ⤳ a(a) ⤳ b
```

In the NIL example, G1 is used in non-applied position in the definition of G0. In the a(a) example, the second occurrence of a in a(a) is not in application position.

We call this general case a *closed-higher-order* language: it is *higher-order* in the sense that functions (macro names) can be passed as parameters and returned as results, but those functions remain *closed*: functions cannot be declared locally and capture lexical variables. This language also corresponds to simply-typed *supercombinators* [Hughes 1982; Turner 1979]. (It is also similar to the use of function *pointers* in C, but we use macro names instead of runtime addresses.)

We can consider typed or untyped versions of this closed-higher-order language. The simply-typed version is a fragment of the simply-typed λ-calculus with recursion, so its halting problem is decidable. The untyped version is Turing-complete, just like most extensions with more constructors or richer type systems, so their halting problem is undecidable. C macro authors probably do not consider typing their macros, so they work in the untyped version, but an extension of ML with higher-kinded type definitions would correspond to the typed version. Our algorithm does not depend on types, so it can work on either version.

In the rest of this section we will detail the following claims:

FACT 1. *Our algorithm gives the same result as Dave Prosser's on the first-order macro fragment.*

FACT 2. *Our algorithm extends to the closed-higher-order macro fragment, and remains correct.*

FACT 3. *Neither our algorithm nor Prosser's are complete on the closed-higher-order macro fragment.*

7.2 The C and C++ Standard Bodies on Closed-Higher-Order Macros

Those two examples, NIL and a(a), come from the Defect Reports 017 of the C and C++ standard committee [The C standard committee, working group WG14 1992]. Since the C89 standard was published, programmers have asked for clarifications about the replacement behavior that would dictate how those examples should behave – we have indicated possible reduction sequences in comments, but implementations in the wild would behave differently and often stop before reaching the normal form.

In the first few years, the C standard committee refused to provide clarifications – we understand that those examples were perceived as unrealistic and absent from real-world C code. See the answers to questions 17 and 23 in the Defect report 017 [The C standard committee, working group WG14 1992]. It later became clear that some C or C++ programmers made real-world use of the non-first-order fragment, and lately standard bodies have moved towards actually specifying this behavior, choosing to honor Dave Prosser's original intent to reduce as much as possible. See in particular the discussion of the NIL example above in the document N3882 from the C++ standard body [The C++ standard committee, working group SG12 2014].

7.3 Relating our Algorithm to Dave Prosser's Pseudo-Code

In an appendix in our extended version, we show and explain Dave Prosser's algorithm, we relate it to our own termination-monitoring algorithm (they are different but related), we claim that they provide the same expansions in the first-order case, and finally we explain how Dave Prosser's algorithm works outside the first-order fragment.

We are not aware of any previous proof that Dave Prosser's algorithm terminates on all inputs; our comparison provides a proof in the first-order case for the core fragment, but the closed-higher-order case remains open.

7.4 Adapting our Algorithm to Closed-Higher-Order Macros

For the purposes of OCaml head shape computation, we have only formulated our termination-monitoring algorithm on the first-order λ-calculus with recursion. We now extend our first-order

calculus from Section 6.3 to the closed-higher-order fragment, and show that our termination-monitoring algorithm still enforces termination. In the ML world, this would correspond to extending ML declarations to higher-kinded type parameters – a feature present in Haskell.

$$
\begin{array}{llll}
p & ::= & \text{let rec } D \text{ in } t \\
D & ::= & \emptyset \mid D, f(x_i)^i = t \\
f & & \text{function name} \\
x & & \text{first-order variable} \\
t & ::= & x \mid \boxed{f} \mid \boxed{t}\,(t_i)^i
\end{array}
\qquad
\begin{array}{llll}
\bar{p} & ::= & \text{let rec } D \text{ in } \bar{t} \\
\bar{t} & ::= & x \mid \boxed{t}\,(\bar{t}_i)^i \ @ \ l \\
l & ::= & \emptyset \mid l, f \quad (f \notin l) \\
C[\Box] & ::= & \Box \mid \boxed{t}\,((\bar{t}_i)^i, C[\Box], (\bar{t}_j)^j) \ @ \ l
\end{array}
$$

We have highlighted above the changes compared to the first-order case: we add function names f as first-class terms, and relax our syntax of application from an application of a known function $f(t_i)^i \ @ \ l$ to the application of an arbitrary term $t(t_i)^i \ @ \ l$. Note that the annotation l here is the trace of the whole application node, not the trace of the function symbol – function symbols do not carry an annotation.[6]

The definition of reduction, repeated below for reference, is unchanged. But it now implicitly restricts redexes to the case where the left-hand-side of the function application has first been reduced to a function name f.

$$
\frac{(f(x_i)^i = t') \in D \qquad f \notin l}{\text{let rec } D \text{ in } C[f(\bar{t}_i)^i \ @ \ l] \rightsquigarrow \text{let rec } D \text{ in } C[t'\big[(x_i \leftarrow \bar{t}_i)^i\big]^{@l,f}]}
$$

This extension can trivially express the closed-higher-order examples that we discussed so far, as well as the famous non-terminating λ-term $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$.

```
let rec nil(x) = x
    and g0(arg) = nil(g1)(arg)
    and g1(arg) = nil(arg)
in g0(fortytwo)
```

```
let rec a(x) = b
    and b(x) = x
in a(a)(a)(a)
```

```
let rec delta(x) = x(x)
in delta(delta)
```

The NIL and a(a) examples reduce without getting stuck. The delta(delta) example gets stuck without looping.

```
   g0(fortytwo)[]
⇝ nil(g1)[g0](fortytwo)[g0]
⇝ g1(fortytwo)[g0]
⇝ nil(fortytwo)[g0]
⇝ fortywto
```

```
   a(a)[](a)[](a)[]
⇝ b(a)[](a)[]
⇝ a(a)[]
⇝ b
```

```
   delta(delta)[]
⇝ delta(delta)[delta]
```

THEOREM 7.1. *Our termination-monitoring algorithm remains correct for this closed-higher-order language: the annotated language is strongly normalizing.*

The proof of this result reuses the technical machinery of the correctness proof for the first-order fragment. It is detailed in an appendix in our extended version.

7.5 Both Algorithms are Incomplete Outside the First-Order Fragment

Here is a counterexample to completeness in the closed-higher-order fragment:

---

[6]We conjecture that our treatment corresponds to the intersection of the hide sets of the function name and closing parenthesis in Dave Prosser's algorithm. The function name and closing parenthesis may have been generated by different macro expansions, and the intersection computes a nearest common ancestor. Our application nodes make this nearest common ancestor explicit in the AST.

```
#define f(p,q) p(f(q,q))
#define id(x) x
#define stop(x) done
f(id,stop) // ↝ id(f(stop,stop)) ↝ f(stop,stop)
           // ↝ stop(f(stop,stop)) ↝ done
```

This example requires two nested expansions of f to reduce to its normal form **done**. Neither our algorithm nor Prosser's can reach this normal form; they block at f(stop,stop). Note that this term is an example of a term that is only weakly normalizing: it has a normal form, but also an infinite reduction sequence

```
f(stop,stop) ↝ stop(f(stop,stop)) ↝ stop(stop(f(stop,stop))) ↝ ...
```

8 RELATED WORK

There is a lot of work on value representations in programming languages, including questions of how to optimize data representation. We are not aware, however, of previous academic work on custom (datatype-specific) representation of sum/coproduct types, which requires checking that a candidate representation ensures disjointedness.

We will discuss the state of representation optimization in neighboring languages with sum types. In particular, the "niche-filling" optimizations of Rust are the most closely related work, as they involve a form of disjointedness. To our knowledge they have not previously been discussed in an academic context, except for a short abstract [Bartell-Mangel 2022].

8.1 Functional Programming Languages

Haskell. Haskell offers newtype for single-constructor unboxing. GHC supports unboxed sum types among its unboxed value types. Unboxed value types live in kinds different from the usual kind Type of types whose value representation is uniform. GHC also supports an UNPACK pragma [Marlow 2003] on constructor arguments to require that this argument be stored unboxed – generalizing OCaml's unboxing of float arrays and records. Haskell would still benefit from constructor unboxing. Note that lifted types (containing lazy thunks) would conflict with each other, limiting applicability – one has to use explicitly unlifted types, or Strict Haskell, etc.

MLton. MLton can eliminate some boxing due to aggressive specialization [Weeks 2006]; for example, (int * int) array will implicitly unbox the (int * int) tuple. Its relevant optimizations are SimplifyTypes, which performs unboxing for datatypes with a single constructor (after erasing constructors with uninhabited arguments) and DeepFlatten, RefFlatten which optimize combinations of product types and mutable fields. The representation of sum types with several (inhabited) constructors remains uniform.

Scala. In Scala, representation questions are constrained by the JVM but also by the high degree of dynamic introspectability expected. Even the question of single-constructor unboxing is delicate. A widespread AnyVal pattern has disappointing performance [Chan 2017; Compall 2017] and dotty introduced a specific *opaque type synonym* feature [Odersky and Moors 2018; Osheim, Cantero and Doeraene 2017] to work around this.

Specialization and representation optimizations. Both MLton and Rust create opportunities for datatype representation optimizations by performing aggressive monomorphization. In OCaml, statically specializing the representation of (int * int) array (as MLton does) or Option<Box<T>> (as Rust does) to be more compact would not be possible, as polymorphic functions working with 'a array or 'a option inputs are compiled once for all instances of these datatypes.

On the other hand, a non-monomorphizing language could perfectly implement and support representation optimizations for types that are only used in a specialized context. For example, at the cost of some code duplication, high-performance code could define its own `'a option_ref` code that looks similar to (`'a option`) **ref** but with a more compact representation. (In fact people do this today using foreign or unsafe code; `'a ref_option` can be safely expressed with inline records.) Finally, in Section 5.1 we described parameterized type definitions that support a quantification on the head shape of the type parameter, which could provide a compromise between genericity and unboxing opportunities.

We thus consider that monomorphization is a separate concern from representation optimizations such as unboxing. Monomorphization creates more opportunities for representation optimizations, with well-known tradeoff in terms of compilation time and code size. Languages could consider representation optimizations whether or not they perform aggressive specialization.

Unboxed sums and active/view patterns. Some languages, such as GHC Haskell and F♯, support both unboxed sums as "value types" [Ağacan 2016; Syme 2016] and active/view patterns [Peyton-Jones 2007; Syme, Neverov and Margetson 2007] to apply user-defined conversions during pattern matching. Combining those two features can get us close to unboxed constructors in some contexts, including our Zarith/gmp example. The idea is to have a very compact primitive/opaque value representation, and expose a "view" of these values in terms of unboxed sums – in F♯ they are called "struct-based discriminated unions".

This style can combine a compact in-memory representation, yet provide the usual convenience of pattern-matching, without extra allocations – even when crossing an FFI boundary. The overhead would typically be higher than native constructor unboxing, but only by a constant factor.

This approach is very flexible, it can be used to perform representation optimizations that are not covered by single-constructor unboxing alone. For example, it can be used to view native integers into an unboxed sum type of positive-or-negative-or-zero numbers. On the other hand, constructor unboxing extends the ranges of memory layouts that can be expressed directly in the language, as in our Zarith example; to use the "view" in this situation, we have to implement the representation datatype and the view function in unsafe foreign code.

8.2 Rust: Niche-Filling

Rust performs a form of constructor unboxing by applying so-called "niche-filling rules". The paradigmatic example of niche-filling is unboxing `Option<A>` for all types A whose representation contains a "niche" leaving one value available – typically, if A is a type of non-null pointers, the 0 word is never a valid A representation and can be used to represent the option's None.

Note that niche-filling in Rust is a qualitatively different optimization from constructor unboxing in OCaml, as it does not remove indirection through pointers. In Rust datatypes, sums (enums) are unboxed and pointer indirections are fully explicit through types such as `Box<T>`, and to our knowledge they are never optimized implicitly. Niche-filling shrinks the width of (value/unboxed) sum types by removing their tag byte; this is important when using (unboxed) arrays of enums, and because all values of an enum are padded to the width of its widest constructor. One could say that it is an *untagging* rather than *unboxing* optimization – but a tag is a sort of box.

The current form of niche-filling was discussed in Beingessner [2015] and implemented in two steps in Burtescu [2017] and Benfield [2022]. We understand that it works as follows: among the constructors whose arguments have maximal total width, try to find one with a "niche", a set of impossible values, that is large enough to store the tag of the sum. Then check that the arguments of all other constructors can be placed in the remaining space – outside the niche.

We can formulate niche-filling in our framework. Head shapes are abstractions of sets of binary words – all of the same size. In particular, the complement of the head shape of a type is its set of niches. Rust allows unboxing exactly when *all* constructors of a type can be "unboxed" in our sense – they can be placed in non-overlapping subsets of binary words. Rust will implicitly change the placement of some constructor arguments to obtain more non-overlapping cases, which is an instance of the "unboxing by transformation" approach that we discussed in Section 5.3.

Niche-filling has some limitations:

(1) Filling a niche with a type B is a direct inclusion of the tag and values of B, possibly placed at non-zero offsets within A; it is not possible to transform the representation of B along the way (for example, if B has two values, it is not possible to use the niches of two separate non-null pointer fields, there must be a full padding bit available somewhere.)
This limitation is shared with our work. (We point it out because some people expect magic from Rust optimizations.) One reason is that this guarantees that projecting the B out of the value is cheap, instead of requiring a possibly-expensive transformation.

(2) Today niche-filling is implementation-defined in Rust, purely a compiler heuristic, and there is no interface for the user to control niche-filling behavior. There seem to be some rare cases where niche-filling in fact degrades performance by making pattern-matching slightly more complex; the only recourse for programmers is to use the enum(C) attribute to ask for an ABI-specified layout that disables all representation optimizations. (There is an instance of this workaround in the PR that generalized niche-filling, 94075, out of concern for a potential performance regression in the Rust compiler itself.)

(3) Rust currently only allows niche-filling when all constructors can be unboxed at once. This is less expressive than our approach where some constructors can be unboxed selectively while others are not. In the context of Rust, removing the tag bytes from *all* constructors is the very point of niche-filling, so unboxing (untagging, really) only certain constructors selectively is not interesting. This suggests however that niche-filling as currently conceived in the Rust community is a specialized form of constructor unboxing, that is not necessarily suitable for other programming languages. Our presentation is more general.

8.3 Declarative Languages for Custom Datatype Representations

Hobbit. The experimental language Hobbit offered syntax to define algebraic "bit-datatypes" together with a mapping into a lower-level bit-level representation for FFI purposes, with support for constructing and matching the lower-level value directly. See in particular Diatchki, Jones and Leslie [2005], Section "2.2.4 Junk and Confusion?" on page 9, which refers to the coproduct property as a "no junk and confusion" guarantee, and mentions that their design in fact allows confusion: they are not forbidden by a static analysis, but a warning detects them in some cases.

One expressivity limitation is that "bit-datatypes" form a "closed universe": all types mentioned in a "bit-datatype" declaration must themselves be bit-data whose representation is available to the compiler, it does not seem possible to specify only a part of the variant value as bit-data, and have arbitrary values of the language (that may not have a bit-data encoding) for other arguments.

Cogent / Dargent. Dargent [Chen, Lafont, O'Connor, Keller, McLaughlin, Jackson and Rizkallah 2023] is a data layout language for Cogent, focused on providing flexible data representation choices for structure/product types. Users can define custom layout rules for record types, the system generates custom getters and setters for each field and verifies that each pair of a getter and setter satisfies the expected specification of being a lens. Cogent also has sum types, but Dargent does not allow unboxed layouts; it requires a bitset to store a unique tag for each case. Extending this formalization to richer representations of sum types would allow constructor unboxing. This could

be done by generating a pattern-matching function and a family of constructor functions, and checking that those satisfy the expected properties of a coproduct – disjointedness.

(There are many other data-description languages (DDLs), but they tend to not offer specific support for sum types / disjoint unions, especially not approaches to unbox them. The Dargent Related Work mentions the prominent previous works in this area.)

`Ribbit`: *Bit Stealing Made Legal.* Simultaneous to our work, Baudon, Radanne and Gonnord [2023] propose a new domain-specific language to express data layouts. It is designed for efficiency rather than for interoperability with foreign data, supports flexible representation optimizations for sum types, and formalizes (on paper) the "no-confusion" correctness condition for sums.

Having a full layout DSL is more expressive than our approach. The authors had both OCaml and Rust in mind when elaborating their proposed DSL, called `Ribbit`, which is thus expressive enough to express constructor unboxing (Zarith is mentioned as a use-case), as well as Rust niche-filling optimizations. The language currently suffers from the same "closed universe" limitation as Hobbit, and needs to be extended to deal with abstract types or non-specialized type parameters, but those are not fundamental limitations of the approach.

It would be nice to restructure our work as elaborating the user-facing feature into an intermediate representation of data layouts, and RIBBIT would be a good candidate for this.

The authors discuss efficient compilation of pattern-matching involving datatypes with a custom layout; they moved from matrix-based pattern-matching compilation (as used in the OCaml compiler) to decision-tree-based compilation to handle more advanced, "irregular" representations. (The RIBBIT prototype does not yet support real programs, so it is hard to draw performance conclusions; in contrast our prototype supports the full OCaml language.) So far our intuition is that constructor unboxing, and the various extension we considered, are regular enough that matrix-based compilation can give good results and no invasive changes are required in the OCaml compiler. But this question deserves further study.

8.4 Deciding Termination of Recursive Rewrite Rules

Our termination-monitoring algorithm provides an alternative proof of decidability of the halting problem for the first-order recursive calculus. This result was well-known (inside a different scientific sub-community than ours), but we want to point out that we provide a new (to our knowledge) decision *algorithm*, and that having a simple yet efficient algorithm is important to our application. By "termination-monitoring" we mean that our algorithm only adds some bookkeeping information to computations that we have to perform in any case to compute a normal form.

The simplest proof of decidability of termination for the first-order recursive calculus that we have found (thanks to Pablo Barenbaum) is Khasidashvili [2020]. Seen as a decision algorithm, this proof is terrible: it performs an exponential computation of all normal forms of all types definitions.

On the other hand, other termination proofs exist that also scale to the higher-order cases, and rely on evaluating the program in a semantic domain where base types are interpreted as the two-valued Scott domain $\{\bot, \top\}$, where $\bot$ represents potentially-non-terminating terms and $\top$ represent terminating terms. Terms at more complex types are interpreted as more complex domains, but (in absence of recursive types) they remain finite, so in particular recursive definitions, interpreted as fixpoints, can be computed by finite iteration. This approach in particular underlies the termination proofs in Plotkin [2022]; Salvati and Walukiewicz [2015]. This argument has a computational feel, but naively computing a fixpoint for each recursive definition in the type environment is still more work than we want to perform. It is possible that a clever on-demand computation strategy for those fixpoints would correspond to our algorithm.

REFERENCES

Ömer Sínan Ağacan. 2016. GHC unboxed sums. https://github.com/ghc/ghc/commit/714bebff44076061d0a719c4eda2cfd213b7ac3d

Noah Lev Bartell-Mangel. 2022. Filling a Niche: Using Spare Bits to Optimize Data Representations. https://www.noahlev.org/papers/popl22src-filling-a-niche.pdf POPL'22 student research presentation.

Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal. In *ICFP*. https://doi.org/10.1145/3607858

Aria Beingessner. 2015. Rust RFC 1230: More Exotic Enum Layout Optimizations. https://github.com/rust-lang/rfcs/issues/1230

Michael Benfield. 2022. rustc PR 94075: Use niche-filling optimization even when multiple variants have data. https://github.com/rust-lang/rust/pull/94075

Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. 2011. Full Reduction at Full Throttle. In *CPP*. https://inria.hal.science/hal-00650940

Eduard-Mihai Burtescu. 2017. rustc PR 45225: Refactor type memory layouts and ABIs, to be more general and easier to optimize. https://github.com/rust-lang/rust/pull/45225

Lloyd Chan. 2017. Scala Pre-SIP: Unboxed wrapper types. https://contributors.scala-lang.org/t/pre-sip-unboxed-wrapper-types/987

Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. 2023. Dargent: A Silver Bullet for Verified Data Layout Refinement. *PACMPL* 7, POPL, Article 47 (Jan 2023), 27 pages. https://doi.org/10.1145/3571240

Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. 2019. Unboxing Mutually Recursive Type Definitions in OCaml. In *JFLA 2019*. https://hal.inria.fr/hal-01929508

Stephen Compall. 2017. Blog post: the high cost of AnyVal classes. https://failex.blogspot.com/2017/04/the-high-cost-of-anyval-subclasses.html

Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. High-Level Views on Low-Level Representations. In *ICFP'05*. http://web.cecs.pdx.edu/~mpj/pubs/bitdata-icfp05.pdf

Torbjörn Granlund and contributors. 1991. GMP. https://gmplib.org/

John Hughes. 1982. Super-Combinators a New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming (LFP)*. https://doi.org/10.1145/800068.802129

Zurab Khasidashvil. 2020. A short proof of the decidability of normalization in recursive program schemes. In *Shalva Pkhakadze's Festschrift, AMIM Vol. 25 No. 2*. http://www.viam.science.tsu.ge/Ami/2020_2/5_zura.pdf

Simon Marlow. 2003. GHC's UNPACK pragma. https://github.com/ghc/ghc/commit/abbc5a0be1df84a33015470319062ed7a3aa3153

Antoine Miné and Xavier Leroy. 2012. Zarith. https://github.com/ocaml/Zarith/

Martin Odersky and Adriaan Moors. 2018. dotty PR 5300: Opaque types. https://github.com/lampepfl/dotty/pull/5300

Erik Osheim, Jorge Vicente Cantero, and Sébastien Doeraene. 2017. Scala SIP 35: Opaque types. https://contributors.scala-lang.org/t/pre-sip-unboxed-wrapper-types/987

Simon Peyton-Jones. 2007. GHC view patterns. https://gitlab.haskell.org/ghc/ghc/-/wikis/view-patterns

Gordon Plotkin. 2022. Recursion does not always help. (2022). https://arxiv.org/pdf/2206.08413.pdf

Dave Prosser. 1986. X3J11/86-196: Complete macro expansion algorithm. https://www.spinellis.gr/blog/20060626/x3J11-86-196.pdf

Sylvain Salvati and Igor Walukiewicz. 2015. Using models to model-check recursive schemes. *Logical Methods in Computer Science* Volume 11, Issue 2 (June 2015). https://doi.org/10.2168/LMCS-11(2:7)2015

Diomidis Spinellis. 2008. A corrected and annotated version of the X4J11/86-196 document. https://www.spinellis.gr/blog/20060626/

Don Syme. 2016. Fsharp PR 1395: struct discriminated unions. https://github.com/dotnet/fsharp/pull/1395

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible Pattern Matching via a Lightweight Language Extension. In *ICFP'07 (ICFP '07)*. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/p29-syme.pdf

The C++ standard committee, working group SG12. 2014. n3882; An update to the preprocessor specification. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3882.pdf

The C standard committee, working group WG14. 1992. Defect report 017. https://www.open-std.org/Jtc1/sc22/wg14/www/docs/dr_017.html

David A. Turner. 1979. A new implementation technique for applicative languages. In *Software - Practice and Experience*.

Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *ML Workshop 2006*. http://www.mlton.org/References.attachments/060916-mlton.pdf

Jeremy Yallop. 2020. OCaml RFC: constructor unboxing. https://github.com/ocaml/RFCs/pull/14