Abstract

Unmarshalling primitives in statically typed language require, in order to preserve type safety, to dynamically verify the compatibility between the incoming values and the statically expected type. In the context of programming languages based on parametric polymorphism and uniform data representation, we propose a relation of compatibility between (unmarshalled) memory graphs and types. It is defined as constraints over nodes of the memory graph. Then, we propose an algorithm to check the compatibility between a memory graph and a type. It is described as a constraint solver based on a rewriting system. We have shown that the proposed algorithm is sound and semi-complete in presence of algebraic data types, mutable data, polymorphic sharing, cycles, and functional values, however, in its general form, it may not terminate. We have implemented a prototype tailored for the OCaml compiler [17] that always terminates and still seems sufficiently complete in practice.

Categories and Subject Descriptors  D.3.3 [Programming Languages]: Language Constructs and Features—Input/Output;  F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms  Languages, Safety

Keywords  Type-safe marshalling, OCaml

1. Introduction

Marshalling is the process of transforming the memory representation of a value into a linear form, suitable for communication or storage. Unmarshalling is the reverse operation, importing external data as values in a running program. In the context of functional languages with parametric polymorphism, we can distinguish two main categories of verification mechanisms used to preserve type safety when unmarshalling data.

The first category relies on an ad hoc external representation of values and a couple of ad hoc functions for marshalling and unmarshalling: for a given type the unmarshalling function may either reconstruct a well-typed value or fail if the input is unexpected. This approach suits well the Haskell type-class mechanism which allows the multiple ad hoc functions to be hidden behind a unique name. Many Haskell libraries reuse the principles proposed in [14] to automatically derive the (un)marshalling functions from an algebraic type definition. Such libraries also exist in SML [7] or OCaml [21, 26] with less convenient interfaces.

The second category of verification mechanisms relies on the introduction of run-time type representations. In such mechanisms, marshalling primitives are generic functions that traverse the memory graph representing a value, and linearize it as a sequence of bytes. To ensure type safety, a value is usually marshalled together with its type and the unmarshalling primitives will accept to reconstruct the memory graph only when the marshalled type corresponds to the statically expected one. This category include mechanisms based on a dynamic type [1, 9, 16]. When the marshalling primitives are used to communicate values between different programs on different hosts, the main difficulty is to choose a common representation for the types [5].

We propose in this paper a third approach, also based on generic (un)marshalling functions but without adding more type information in the external representation than those already required to rebuild the corresponding memory graph. To ensure type safety, the initial idea is to check the compatibility of the reconstructed memory graph with the expected type. In other words, we check whether the memory graph may actually represent a value of that type. In a first approach, this can be achieved by recursively traversing the graph and the type together. In a more general way, we have defined an algorithm for checking the compatibility of a memory graph against classical algebraic data types (sums and records), but also against generalized algebraic data types or closures—that may introduce existential types—and against mutable record types or ar-
Uniform data representation in OCaml

Generics (un)marshalling primitives With the uniform tagged representation used by OCaml, a marshalling primitive could be implemented with a generic depth-first traversal of the memory graph. And a type-safe unmarshalling primitive could be conceptually split into two steps: a generic function that rebuilds a well-formed memory graph, with respect to the invariants assumed by the garbage collector, and a function that traverses the reconstructed graph to check its compatibility with the expected type.

A simple recursive algorithm Checking the compatibility of a well formed graph against primitive data types is simple—an integer should be an immediate value, a float should be an allocated value whose tag is the one specific to boxed float, etc.—and checking the compatibility against a classical ADT is easily achieved with a recursive algorithm that test if the size and the tag of the value correspond to a constructor in the type definition, then deduce the expected type for sub-values and recurse. For example, consider the following parametric definition of list:

```
type List(α) = Nil | Cons of α * List(α)
```

A memory graph is compatible with List(Int) if and only if:

- it is the immediate value 0 (representing Nil), or
- it is an allocated value of tag 0 and size 2 (repr. Cons), where:
  - the first element is compatible with Int, i.e. its an immediate value,
  - the 2nd element is recursively compatible with List(Int).

This recursive algorithm is extended in section 2.1 to efficiently handle sharing and cycles in the memory graph. In the presence of polymorphic sharing and polymorphic recursion, as introduced respectively by the generalizing let and let rec constructions[18], this will require to traverse the graph in topological order and to recompute, from all the types expected for a shared value, an approximation of its inferred polymorphic type in the source code. This is the main difference with a classical type inference algorithm: while checking the memory graph corresponding to a program let x = e₁ in e₂, our algorithm traverses e₂—in order to collect all the expected types for x—before traversing e₁.

Section 2.2 describes a simple extension that allows the verification of mutable data, and section 2.3 extends the algorithm to handle generalized algebraic data types (hereafter abbreviated GADT) and function types. This will require the introduction of existential types and the use of unification to instantiate them. This will also complicate the order in which the graph is traversed: the verification of a value against an existential type is postponed until the type is instantiated.

2.1 Topological traversal and anti-unification

In terms of soundness, checking the compatibility of a memory graph with neither cycles nor mutable data could be achieved by checking shared values multiple times, with possibly different expected types. However, given that compilers tend not to introduce more sharing in the memory graph than is explicitly introduced by the programmer with let constructions and functional abstractions, those multiple verifications seem redundant. In fact, to obtain a complete algorithm it should be sufficient to check shared values only once with the type of the corresponding variable in the original source code.

In the absence of polymorphic sharing or polymorphic recursion, a shared value in the source code has a monomorphic type. Hence, we can obtain an efficient algorithm by marking values with their expected types when traversing them for the first time; and, when checking a previously marked value, only test the equality between the new expected type and the memorized one.

Polymorphic sharing in the absence of cycle In the presence of polymorphic sharing, we can only assume that the original type of
a shared value is more general than all the expected types for that value. If we can definitely not recompute the original type, we can still try to recompute a type “minimally more general” than every expected type, hence compatible with this node of the memory graph. With a language based on parametric polymorphism like OCaml, a type that satisfies those conditions is the principal anti-unifier of the expected types [13, 19] (see section 4.2 for a formal definition). Then, in the absence of cycles in the memory graph, we obtain an efficient algorithm by using the following strategy:

- traverse the graph in a topological order, hence collecting all the types expected for a shared value before traversing it,
- check a shared value only once against the anti-unifier of all its expected types.

For example, consider the following values and the corresponding memory graph in figure 2.

```
let leaf : ∀α. Tree(α) = Leaf ".."
let trees : (Tree(Int) + Tree(Char)) =
  (Node (leaf, 3, leaf), Node (leaf, 'A', leaf))
```

The set of expected types collected for `leaf` while checking the compatibility of `trees` against the type `(Tree(Int) + Tree(Char))` is `{Tree(Int); Tree(Char)}`. Their anti-unifier is `∀α. Tree(α)`.

![Concrete tags: tuple = Leaf = 0; Node = 1; string = 252.](image)

**Figure 2.** Sharing in the memory graph

Cycles In an algorithm like ours, which recursively traverses the memory graph and the expected type together, it is not possible to collect all expected types for a value involved in a cycle without traversing that value. Hence, we have to elaborate a specific treatment for such cyclic values. One possibility is to mark them according to the following steps: test if the size and the tag of the value would still be compatible with the checked type.

(a) traverse the graph in topological order using a reference counter and anti-unify all the expected types before traversing a value; if some expected types could not be collected, then the graph contains cycles; proceed with (b).

(b) identify the SCCs and sort them in topological order; for each SCC proceed with (c).

(c) mark the values of the current SCC for which types were collected, anti-unify all the expected types and proceed recursively with (a) for all the marked values, considering the graph reduced to the current SCC.

(d) while checking a previously marked value, fail if the expected type is not equal to the memorized one.

We propose in section 4.4 an extension of that strategy that handles polymorphic recursion.

### 2.2 Mutatable fields

OCaml allows the definition of mutable fields in record types. However, mutability is known only at compile time: the runtime representation of records does not distinguish mutable fields from immutable ones. However, as we traverse a graph and its expected type in parallel, we can add a specific treatment when the expected type requires that a field of the corresponding value has to be immutable ones. However, as we traverse a graph and its expected type in parallel, we can add a specific treatment when the expected type requires that a field of the corresponding value has to be immutable. In that case, we check that the expected type for the mutable field is monomorphic, in order to avoid the well-known soundness problem of polymorphic references.

As the mutation of record fields is achieved in-place, the proposed verification for mutable records is only allowed if the checked memory graph could only be manipulated with the verified type. It is the case in the context of an unmarshalling function but it would not be the case in a cast function that does not duplicate the memory graph first. Otherwise, if a value with a mutable type is cast into an immutable type, nothing warrants that a mutated value would still be compatible with the checked type.

### 2.3 Existential types

The simple recursive algorithm described at the beginning of section 2 allows the verification of compatibility of a value against classical ADT. In this section we describe the different difficulties introduced by the existential types of GADT and how our algorithm handles them. In a second step, the mechanisms introduced to handle existential types will be reused for the verification of polymorphic closures.

**Unification** The version 4.0 of OCaml includes GADTs. Their memory representation is the same as the classical ADTs. In particular, they do not contain witness for existential types. To reconstruct the existential types, the recursive algorithm is extended with a unification mechanism. More concretely, the algorithm proceeds according to the following steps: test if the size and the tag of the value correspond to a constructor in the type definition; unify the expected type with the return type of the constructor; deduce the expected type for sub-values and recurce. For example, consider the following simple type definition:

```
type T1(_ : _) =
  | Int : Int → T1(Int)
  | Char : Char → T1(Char)
  | Bin : T1(α) + T1(α) → T1(α)
```

Checking the compatibility of an allocated value of tag 1 and size 1, representing the constructor `Char`, against type `T1(Int)` will fail while unifying `T1(Int)` with the return type of the constructor `Char`, i.e. `T1(Char)`. On the contrary, while checking the compatibility of an allocated value of tag 2 and size 2, representing `Bin`, against type `T1(Int)`, the unification allows the instantiation

1 at least as long as we do not consider recursive type expression.
of $\alpha$ and the recursive checking of the sub-values against the type $T_1(\text{Int})$.

In a more general way, the unification allows the instantiation of the existential types introduced by GADT. For example, consider the following type definition:

$$\text{type } T_2(\_ : \_ ) = \begin{cases} \text{Int : } \text{Int} \rightarrow T_2(\text{Int}) \\ \text{Char : } \text{Char} \rightarrow T_2(\text{Char}) \\ \text{Couple : } T_2(\alpha) \times T_2(\beta) \rightarrow T_2(\alpha \times \beta) \\ \text{Fst : } T_2(\alpha \times \beta) \rightarrow T_2(\alpha) \end{cases}$$

While checking the compatibility of an allocated value of tag 3 and size 1, representing Fst, against the type $T_2(\text{Int})$, the unification leaves the $\beta$ uninstantiated. Then, the recursive call has to verify that there exists a type $\beta$ such that the argument of Fst is compatible with $T_2(\text{Int} \times \beta)$. If this argument is the representation of Couple (Int 1, Char 'A'), it is decomposed in two independent checks:

- is the representation of Int 1 compatible with $T_2(\text{Int})$?
- is the representation of Char 'A' is compatible with $T_2(\beta)$?

The latter will instantiate $\beta$ as Char.

**Delayed verification**  In the presence of existential types, not every memory graph can be verified with a simple recursive algorithm. For example, consider the following type definition of dynamic types reduced to three constant constructors:

$$\text{type } Ty(\_ : \_ ) = \begin{cases} \text{TBool : } Ty(\text{Bool}) \\ \text{TInt : } Ty(\text{Int}) \\ \text{TChar : } Ty(\text{Char}) \end{cases}$$

The representation of Dyn (65, TInt) against the type Dyn with the recursive algorithm leads to the following question: is there a type $\beta$ such that:

- the immediate value 65 is compatible with $\beta$, and
- the representation of TInt is compatible with $Ty(\beta)$?

There are at least two possible instantiations of $\beta$ that could satisfy the former part of the question, that is: Int and Char; but only Int would also satisfy the latter part. Hence, while traversing this value, it is important to check the compatibility of TInt—and instantiate $\beta$ as Int by unification—before traversing 65. For this purpose, the algorithm must be able to delay the verification of some values until the expected type has been instantiated.

**Universal types**  Some existential types can not be instantiated by unification. Consider the following type definition, similar to $T_2(\_ : \_ )$ but where the constructors Int and Char have been replaced by a generic constructor Const:

$$\text{type } T_3(\_ : \_ ) = \begin{cases} \text{Const : } \alpha \rightarrow T_3(\alpha) \\ \text{Couple : } T_3(\alpha) \times T_3(\beta) \rightarrow T_3(\alpha \times \beta) \\ \text{Fst : } T_3(\alpha \times \beta) \rightarrow T_3(\alpha) \end{cases}$$

While checking the compatibility of a memory graph representing Fst (Couple (Const 3, Const 'A')), against $T_3(\text{Int})$, the test will be reduced to:

- is there a type $\beta$ such that the immediate value 65, representing the character 'A', is compatible with $\beta$?

While answering this question on the paper is trivial, our algorithm could not delay the verification anymore or instantiate $\beta$ by unification. However, in the same way Goldberg [10] shows that a garbage collector based on type reconstruction can collect values for which it can not reconstruct the type, we can ignore such values without breaking the soundness of the host language. For formalizing this situation in section 3, we will introduce an abstract type, named universal type, that is compatible with every memory graph. It will be used at the end of the traversal to instantiate the remaining type variables.

**Functional types**  We now extend the algorithm to allow the verification of closures. We suppose that the compiler is able to provide an association table that stores for each code pointer, the static type of the corresponding lambda-abstraction in the source code and the typing environment required to type the function. For example, consider the following piece of code:

```latex
defined \text{let delayed_apply} : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \text{Unit} \rightarrow \beta = \\
\lambda f. \lambda x. \\
defefined \text{let apply} : \text{Unit} \rightarrow \beta = \lambda(). (f x) \text{ in} $$
```

The static type associated to the three named lambda-abstractions $\text{succ}$, $\text{delayed_apply}$, and $\text{apply}$ are the following types, where the typing environment is written inside square brackets:

- $\sigma_{\text{succ}} = [\text{one} \mapsto \text{Int}] \rightarrow \text{Int} \rightarrow \text{Int}$
- $\sigma_{\text{delayed}} = \forall \alpha \beta [\_ ] \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \text{Unit} \rightarrow \beta$
- $\sigma_{\text{apply}} = \forall \alpha \beta. [f] \mapsto (\alpha \rightarrow \beta); x \mapsto \alpha \rightarrow \text{Unit} \rightarrow \beta$

Now consider the following partial application of $\text{delayed_apply}$ and the corresponding closure in figure 3 where dashed arrows represent code pointers and their associated static types:

```latex
defined \text{let dsucc} : \text{Unit} \rightarrow \text{Int} = \text{delayed_apply succ 52} $$
```

The code pointer of the closure representing $\text{dsucc}$ corresponds to the code of $\text{apply}$. Hence, the static type associated to $\text{dsucc}$ is $\sigma_{\text{apply}}$ and while checking the compatibility of $\text{dsucc}$ against the type $\text{Unit} \rightarrow \text{Int}$ the exact type of the environment has been lost and must be recomputed by our algorithm:

```latex
[f] \mapsto (\text{Int} \rightarrow \text{Int}); x \mapsto \text{Int} \rightarrow \text{Unit} \rightarrow \text{Int}$$
```

Given this static type information, checking the compatibility of a closure against a functional type could be achieved with a mechanism similar to those introduced for handling GADTs. In our algorithm this is achieved in three steps: instantiate the associated static type schema by replacing universally quantified type variable with existential types; unify the expected type with the instantiated static type; recursively check the environment.

```latex
\{\text{one} \mapsto \text{Int}\} \vdash \lambda x. x + \text{one} : \text{Int} \rightarrow \text{Int}$$
```

```latex
\{ f \mapsto (\alpha \rightarrow \beta) \} \vdash \lambda(). f x: \text{Unit} \rightarrow \beta$$
```

Concrete tags: fun = 249.

**Figure 3. Representation of the function $dsucc$**

**2.4 Existential types in presence of sharing**

When we try to implement in the presence of existential types the topological traversal proposed in section 2.1, two facets still require attention: how to anti-unify existential types and how to avoid “deadlocks” between the topological sort and the delayed verification.
Parallel propagation of types  When the set of collected types for a shared value contains existential types, it is not always possible to compute their anti-unifier. For example, if the set of expected types contains \( \alpha \rightarrow \text{Int} \) and \( \text{Bool} \rightarrow \beta \) where \( \alpha \) and \( \beta \) are existential types, the anti-unifier depends on the future instantiation of \( \alpha \) and \( \beta \): if \( \alpha \) is later instantiated in \( \text{Int} \) and \( \beta \) in \( \text{Bool} \), the anti-unifier is \( \forall \gamma. \gamma \rightarrow \gamma \); but if \( \alpha \) is instantiated in \( \text{Int} \) and \( \beta \) in \( \text{Int} \), it is \( \text{Int} \rightarrow \text{Int} \); and for any other substitutions, it is \( \forall \gamma \delta. \gamma \rightarrow \delta \). There is no other choice than to check the compatibility of the shared values against every collected type.

For this situation, we have extended our algorithm to check the compatibility of a value against a set of types and stated that it will accept the value if and only if it is compatible with the anti-unifier of the set. This allows two optimizations. Firstly, given a set, all types must have the same head constructors, otherwise their anti-unifier is the empty type and no value could be accepted; this allows partial instantiation of some existential types. Secondly, the set of types that is computed for sub-values of an allocated value may not contain existential types and it could be simplified by anti-unification.

The same difficulties arise in the presence of cycles: the equality test between the expected types for a marked value and all the possible anti-unifiers of the memorized set of types is not complete. For example, if the expected type is \( \forall \alpha. \alpha \rightarrow \alpha \) and the set of memorized types contains \( \alpha \rightarrow \text{Int} \) and \( \text{Bool} \rightarrow \beta \), the test is valid only if \( \alpha \) is later instantiated as \( \text{Int} \) and \( \beta \) in \( \text{Bool} \) but is invalid otherwise. In this situation, the test may be postponed until the end of the traversal. At that stage, remaining type variables can be instantiated with the universal type and the set of expected types can be anti-unified.

Explicit decomposition  For some specific cases, delaying the verification of a value when the expected type is an existential type does not always allow the collection of all the expected types for shared values. Consider for example the following piece of code and the corresponding memory graph in figure 4:

\[
\begin{align*}
\text{type T} & = \text{None} \mid \text{Some of (T \rightarrow \text{Int})} \\
\text{let f : T \rightarrow \text{Int} =} & \text{\lambda x. match x with} \\
& \text{\mid None \rightarrow 0} \\
& \text{\mid Some g \rightarrow g Some} \\
\text{let h : Unit \rightarrow \text{Int} =} & \text{delayed_apply f (Some f)}
\end{align*}
\]

Verifying the compatibility of the closure \( h \) against the type \( \text{Unit} \rightarrow \text{Int} \) will be recursively decomposed into the question of knowing if there is an \( \alpha \) such that:

- the representation of \( f \) is compatible with \( \alpha \rightarrow \text{Int} \), and
- the representation of \( \text{Some f} \) is compatible with \( \alpha \)?

If we try to follow the proposed strategy for graph traversals then the verification is stuck at this step: we can not traverse \( f \) to solve the first part of the question until the second expected type for \( f \) is collected; and we can not traverse \( \text{Some f} \) to solve the second part until \( \alpha \) is instantiated. In such situation, our algorithm forces the verification of \( \text{Some f} \) by introducing an existential type for every sub-value of the allocated block representing \( \text{Some f} \)—here it will introduce an existential type \( \beta \) for \( f \)—and it memorizes an explicit constraint to be solved when \( \alpha \) is instantiated: is an allocated value with the tag of \( \text{Some} \) and with a single sub-value of type \( \beta \) compatible with \( \alpha \)?

3. A type system for heap values

Before formally describing our algorithm in section 4, we define in this section the syntax of values and the compatibility relation, between the representation of a value in memory and a type, on which the algorithm is based. The syntax of values relies on the notion of heap to allow the explicit description of sharing and cycle.

Then, the compatibility relation is defined as a classical type system for heap values. We do not prove in this paper the soundness of this type system with respect to the operational semantics of a programming language. However, our language of values and its type system are very similar to the OCaml ones and we assume their soundness.

Syntax of types and values  In order to simplify the formalization of the proposed algorithm while still covering all the situations described in section 2, we restrict ourselves to the following type algebra:

\[
\tau ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid (\tau \ast \tau) \mid \text{List(\tau)} \mid \tau \rightarrow \tau
\]

The universal type is a type compatible with every immediate value and allocated block. It is required to instantiate the existential variables that could not be instantiated by unification (see section 2.4). We write \( \nu(\tau) \) the set of variables of \( \tau \) and we write \( \tau_1 \sqsubseteq \tau_2 \) when \( \tau_2 \) is an instance of \( \tau_1 \), meaning there exists a substitution \( \theta \) such that \( \theta(\tau_1) = \tau_2 \).

A value is represented by a couple \((w, \mu)\) where \( w \) is a word and \( \mu \) is a heap. A word \( w \) is either an integer or a pointer:

\[
w ::= i \mid \ell
\]

A heap \( \mu \) is a partial function from pointers to allocated values. An allocated value \( h \) is either a block containing two words or a closure:

\[
h ::= \text{blk}(w, w) \mid (\tau, w)
\]

To simplify the formal presentation of the algorithm, references are represented by blocks of size 2 whose second element is the immediate value 0.

A closure is represented by the static type of the original code (see section 2.3) and a single word \( w \) for the environment. We will write \( \nu(p(\text{img}() \cup \{\ell\})) \) for the set of pointers occurring in the image of \( \mu \). A value \((\ell, \mu)\) is well defined when \( \nu(p(\text{img}()) \cup \{\ell\}) \) is included in the domain of \( \mu \).

Typing rules  A typing environment \( \Psi \) is a partial function from pointer to types. Typing judgments for words \( w \) and allocated values \( h \) are respectively written \( \Psi \vdash w : \tau \) and \( \Psi \vdash h : \tau \). We will say that a pointer \( \ell \) is compatible with a type \( \tau \) in an environment \( \Psi \) if \( \tau \) is an instance of \( \Psi(\ell) \).

\[
\begin{align*}
\Psi(\ell) & \approx \tau \\
\Psi \vdash \ell : \tau & \quad \text{[W-LABEL]}
\end{align*}
\]
The compatibility of an integer with a type is abstracted by a function \( \text{Imm}() \) that returns the set of immediate values compatible with \( \tau \). It is defined by case analysis on the head type constructor:

\[
\begin{align*}
\text{Imm}(\text{Unit}) &= \{0\} & \text{[IMM-UNIT]} \\
\text{Imm}(\text{List}(\tau)) &= \{0\} & \text{[IMM-EMPTYLIST]} \\
\text{Imm}(\text{Bool}) &= \{0; 1\} & \text{[IMM-BOOL]} \\
\text{Imm}(\text{Int}) &= \{\ldots; -1; 0; 1; \ldots\} & \text{[IMM-INT]} \\
\text{Imm}(\ast) &= \{\ldots; -1; 0; 1; \ldots\} & \text{[IMM-UNIV]} \\
\text{Imm}(\tau) &= \emptyset \quad \text{otherwise}
\end{align*}
\]

\( i \in \text{Imm}(\tau) \quad \frac{}{\Psi \vdash 1 : \tau} \quad \text{[W-INT]} \)

The compatibility of a block \( \text{Blk}(w_1, w_2) \) with a type uses a function \( \text{Arg}(\tau) \) that returns the types expected for the components of a block of type \( \tau \). It is defined by case analysis of the head type constructor\(^2\):

\[
\begin{align*}
\text{Arg}(\text{List}(\tau)) &= [\tau; \text{List}(\tau)] & \text{[ARG-LIST]} \\
\text{Arg}(\text{((} \tau_1 \text{ + } \tau_2 \text{))}) &= [\tau_1; \tau_2] & \text{[ARG-COUPLE]} \\
\text{Arg}(\text{Ref}(\tau)) &= [\tau; \text{Unit}] & \text{[ARG-REF]} \\
\text{Arg}(\tau) &= [\tau_1; \tau_2] \quad \frac{}{\Psi \vdash \text{Blk}(w_1, w_2) : \tau} \quad \text{[H-BLK]} \\
\Psi \vdash w_1 : \tau_1 \quad \Psi \vdash w_2 : \tau_2 \\
\end{align*}
\]

A block is compatible with the universal type only if, for each of its components, there exists a type that is compatible with it\(^3\):

\[
\Psi \vdash w_1 : \tau_1 \quad \Psi \vdash w_2 : \tau_2 \\
\frac{}{\Psi \vdash \text{Blk}(w_1, w_2) : \tau} \quad \text{[H-UNIV]}
\]

A closure \(<\tau, w>\) is compatible with a type \( \tau_1 \rightarrow \tau_2 \) if there exists a type \( \tau_{env} \) compatible with the captured environment and such that \( \tau_{env} \rightarrow \tau_1 \rightarrow \tau_2 \) is an instance of the static type of the original code.

\[
\tau \triangleq \tau_{env} \rightarrow \tau_1 \rightarrow \tau_2 \\
\Psi \vdash (\tau, w) : \tau_{env} \\
\frac{}{\Psi \vdash \text{Clos}(\tau, w) : \tau_1 \rightarrow \tau_2} \quad \text{[H-CLOS]}
\]

To prove some intermediate steps of the type verification algorithm, we will have to manipulate partially typed heaps and to distinguish the type hypothesis for the pointers of a heap from the actual type of those pointers. In that case, we will say that a heap \( \mu \) is partially compatible with type environment \( \Psi \) under the hypothesis \( \Psi' \), and we will write \( \Psi' \vdash \mu : \Psi \), if and only if:

- \( \text{dom}(\Psi) \subseteq \text{dom}(\mu) \)
- \( \text{fp}(\text{img}(\mu)) \subseteq \text{dom}(\Psi') \)
- for all pointers \( \ell \in \text{dom}(\Psi) \) then \( \Psi' \vdash \mu(\ell) : \Psi(\ell) \)
- for all pointers \( \ell \in \text{dom}(\Psi') \) such that \( \Psi'(\ell) = \text{Ref}(\tau) \), then \( \tau \) is closed.

We will say that a heap \( \mu \) is compatible with an environment \( \Psi \) and we will write \( \mu : \Psi \) if and only if \( \Psi' \vdash \mu : \Psi \). We will say that a value \( (w, \mu) \) is compatible with a type \( \tau \) if there exists an environment \( \Psi \) such that \( \mu : \Psi \) and \( \Psi \vdash w : \tau \).

4. Rewriting system

The order in which the algorithm described in section 2 traverses a memory graph depends on the structure of the graph but also on the expected types. To formally describe this complex traversal, we used a rewriting system (sections 4.1 and 4.2) inspired from a constraint-based algorithm for ML type inference\(^2\). In a first step, the set of rewriting rules allows us to prove the soundness and the semi-completeness of the algorithm independently of the traversal order (section 4.3). In a second step, a rewriting strategy allows us to precise the traversal order (section 4.4). In its general form, the algorithm does not terminate, but we show its termination in the absence of cycle, as well as in the absence of existential types. To conclude we propose a variant of the algorithm that always terminates and still seems sufficiently complete in concrete situation.

4.1 Syntax of constraints

Terms of the rewriting system are called constraints. They are defined by the following grammar detailed afterwards:

\[
C ::= \text{True} \quad \text{trivial constraint} \\
\text{False} \quad \text{failure} \\
C \land C \quad \text{conjunction} \\
(\mu, \Phi), C \quad \text{heap fragment and memorized types} \\
w : [\tau; \ldots] \quad \text{type constraint} \\
\exists \alpha, C \quad \text{existential quantification} \\
\tau = \tau \quad \text{unification} \\
\text{arg}(\tau) = [\tau; \tau] \quad \text{ADT decomposition} \\
\{\tau; \ldots\} \quad \text{head type constructors homogeneity}
\]

where \( \Phi \) is an environment that associates a set of types to a pointer: it allows the memorization of the already checked types for an allocated value. In the rewriting rules, we will write \( \Phi \cup \{\ell : \{\tau_1, \ldots\}\} \) to denote the environment obtained by replacing in \( \Phi \) the set associated to \( \ell \) by the extended one \( \Phi(\ell) \cup \{\tau_1, \ldots\} \), i.e. the environment \( \Phi \cup \{\ell : \Phi(\ell) \cup \{\tau_1, \ldots\}\} \).

A type constraint \( w : \{\tau; \ldots\} \) represents the set of yet unchecked types for the value \( w \). Hence, the initial constraint for checking a value \( (w, \mu) \) against a type \( \tau \) is:

\[
(\mu, \emptyset), w : \{\tau\}
\]

The normal forms will be the constraints True and False.

The existential quantification \( \exists \alpha, C \) binds a type variable in \( C \). The existentially qualified type variables are syntactically distinguished with an upper dot. The other type variables—that appear in a constraint without being explicitly bound by an existential quantifier—represent empty types. We will call them universal type variables. We will write \( \text{fev}(\tau) \) (respectively \( \text{fv}(\tau) \)) the set of existential (resp. universal) variables of a type.

Equalities \( \tau_1 = \tau_2 \) will be resolved by a unification procedure, that will consider universal variables as fixed types and produce a substitution for existential variables. We will write such a substitution \( \theta \).

The ADT decomposition \( \text{arg}(\tau) = [\tau; \tau] \) and the head type constructor homogeneity \( \{\tau; \ldots\} \) are required when forcing the verification of a block against an existential type variable.

The construction \( (\mu, \Phi), C \) binds the pointer from \( \text{dom}(\mu) \) into \( C \) and into the image of \( \mu \). This explicit introduction of the heap in the syntax of constraints will allow in section 4.4 the split of a heap into fragments corresponding to SCCs and the expression of a rewriting strategy based on a topological sort.

4.2 Rewriting rules

Rewriting rules are written \( C_1 \Rightarrow C_2 \), meaning that a constraint \( C_1 \) could be rewritten into \( C_2 \) in any context. We present progressively the set of rewriting rules, starting with those for handling sharing and cycles. We then present the set of rules allowing the verification of immediate values and allocated blocks, and finally those required for closures and existential types.
Allocated values

\[(\mu, \Phi'). (S : \{\tau_1 \ldots \} \land C) \Rightarrow \exists \alpha_{i} \ldots \beta_{i}. \{ \o \{ \tau_1 \ldots \} \land \omega : \{ \beta_{i} \ldots \} \land C \}
\]

\[\text{L}^{\alpha_{i}}_{\Phi'} (S : \{\tau_1 \ldots \} \land C) \Rightarrow \exists \alpha_{i} \ldots \beta_{i}. \{ \o \{ \tau_1 \ldots \} \land \omega : \{ \beta_{i} \ldots \} \land C \}
\]

\[\text{Parametericity}
\]

\[\exists \alpha_{i} \ldots \beta_{i}. \{ \o \{ \tau_1 \ldots \} \land \arg(\beta) = [\tau ; \tau'] \land \nu : \{ \gamma \} \Rightarrow \text{True}
\]

\[\text{Immediate values}
\]

\[\exists \alpha_{i} \ldots \beta_{i}. \{ \o \{ \tau_1 \ldots \} \land \arg(\beta) = [\tau ; \tau'] \land \nu : \{ \gamma \} \Rightarrow \text{True}
\]

\[\text{Blocks}
\]

\[\exists \alpha_{i} \ldots \beta_{i}. \{ \o \{ \tau_1 \ldots \} \land \arg(\beta) = [\tau ; \tau'] \land \nu : \{ \gamma \} \Rightarrow \text{True}
\]

\[\text{Rewriting rules}
\]

We let implicit some trivial rules such as associativity and commutativity of the conjunction and commutativity and hoisting of existential quantification. In general, we write constraints in the following form, where overlines denote sets:

\[\exists \alpha. \tau = \tau \land \o \{ \tau \} \land \arg(\beta) = [\tau ; \tau'] \land \nu : \{ \gamma \} \Rightarrow \text{True}
\]

\[\text{Anti-unification}
\]

The formalization of the algorithm uses the definition of principal anti-unifiers proposed by Huet [13]. It supposes the existence of a bijection \(au(\tau, \tau')\) from the set of distinct couple of types with existential variable to a subset of universal type variables. We write \(\tau \land \tau'\) for the anti-unifier of \(\tau\) and \(\tau'\) defined by the following rules of congruence:

1. \(\text{Int} \land \text{Int} = \text{Int}\)
2. \(\text{Unit} \land \text{Unit} = \text{Unit}\)
3. \(\text{List}(\tau) \land \text{List}(\tau') = \text{List}(\tau \land \tau')\)
4. \(\text{Ref}(\tau) \land \text{Ref}(\tau') = \text{Ref}(\tau \land \tau')\)
5. \((\tau_1 + \tau_2) \land (\tau_1' + \tau_2') = ((\tau_1 \land \tau_1') + (\tau_2 \land \tau_2'))\)
6. \((\tau_1 \rightarrow \tau_2) \land (\tau_1' \rightarrow \tau_2') = ((\tau_1 \land \tau_1') \rightarrow (\tau_2 \land \tau_2'))\)
7. \(\tau \land \tau' = au(\tau, \tau')\) otherwise

Modulo a renaming of the produced type variables, the definition of anti-unification is associative and commutative. Given a set of types \(\tau_1 \ldots \), we write \(\lambda(\tau_1 \ldots)\) for its principal anti-unifier.

Property 4.1. We have \((\tau_1 \land \tau_2) \equiv \tau_1\) and \((\tau_1 \land \tau_2) \equiv \tau_2\).

Property 4.2. If \(\tau \equiv \tau_1\) and \(\tau \equiv \tau_2\), then \(\tau \equiv (\tau_1 \land \tau_2)\).

Sharing The rewriting rule [R-MERGE] allows the merge of two sets of expected types for the same allocated value, while the [R-AUNIFI] allows the simplification of a set of expected types by replacing two of its types by their principal anti-unifier. The latest applies only if the anti-unifier is independent of the future instantiation of existential type variables.

\[\ell : \{\tau_1 \ldots \} \land \ell : \{\tau_1 \ldots \} \Rightarrow \ell : \{\tau_1 \ldots \}
\]

\[w : \{\tau_1 \ldots \} \Rightarrow w : \{\tau_1 \ldots \}
\]

The rewriting rule [R-AUNIFI] allows the simplification of the set of memozied types for an allocated value by replacing two of the types by their principal anti-unifier. As for the rule [R-AUNIFI], it applies only if the anti-unifier is independent of the future instantiation of existential type variables.

\[\ell : \{\tau_1 \ldots \} \land \ell : \{\tau_1 \ldots \} \Rightarrow \ell : \{\tau_1 \ldots \}
\]

\[w : \{\tau_1 \ldots \} \Rightarrow w : \{\tau_1 \ldots \}
\]

The rule [R-REMOVE] allows the removal of type constraints that are instances of previously checked types. More precisely, this rule applies only if, for any substitution \(\theta\), the anti-unifier of the instatiation of the expected types by \(\theta\) is an instance of the anti-unifier of the instatiation of the memozied types by \(\theta\).

\[\ell : \{\tau_1 \ldots \} \land \ell : \{\tau_1 \ldots \} \Rightarrow \ell : \{\tau_1 \ldots \}
\]

\[w : \{\tau_1 \ldots \} \Rightarrow w : \{\tau_1 \ldots \}
\]
Consider the following heap containing two closures and two blocks—it corresponds to the memory graph in figure 4 where the closure’s environment has been boxed in an allocated block:

\[
\begin{align*}
\mu &= (l_1 \mapsto \langle (\alpha + (\alpha \rightarrow \beta)) \rightarrow \text{Unit} \rightarrow \beta, l_2 \rangle \\
&\quad \quad \quad \quad \text{ } l_2 \mapsto \text{Blk}(l_3, l_4) \\
&\quad \quad \quad \quad \text{ } l_3 \mapsto \text{Blk}(l_4, 0) \\
&\quad \quad \quad \quad \text{ } l_4 \mapsto \langle \text{Unit} \rightarrow T \rightarrow \text{Int}, 0 \rangle
\end{align*}
\]

And consider the following type environments that will be used as aliases for memorig types in the rewriting sequence:

\[
\begin{align*}
\Phi_1 &= \{ l_1 \mapsto \{ \text{Unit} \rightarrow \text{Int} \} \} \\
\Phi_2 &= \Phi_1 \cup \{ l_2 \mapsto \{ (\alpha + (\alpha \rightarrow \text{Int})) \} \} \\
\Phi_3 &= \Phi_2 \cup \{ l_3 \mapsto \{ \alpha \} \} \\
\Phi_4 &= \Phi_3 \cup \{ l_4 \mapsto \{ \beta, \gamma \rightarrow \text{Int} \} \} \\
\Phi'_4 &= \Phi_1 \cup \{ l_2 \mapsto \{ (T * (T \rightarrow \text{Int})) \} \\
&\quad \quad \quad \quad \text{ } l_3 \mapsto \{ T \} \\
&\quad \quad \quad \quad \text{ } l_4 \mapsto \{ T \rightarrow \text{Int}; T \rightarrow \text{Int} \}
\end{align*}
\]

To allow the verification of value of type T, the functions \(\text{Imm(\_)}\) and \(\text{Arg(\_)}\) are extended with the following cases:

\[
\begin{align*}
\text{Imm(T)} &= \{0\} \\
\text{Arg(T)} &= \{1 \rightarrow \text{Unit} \}; \text{Unit}
\end{align*}
\]

A possible rewriting sequence for checking the compatibility of the value \((l_1, \mu)\) against the type \(\text{Unit} \rightarrow \text{Int}\) begins by checking the closure \(\mu(l_1) = ((\alpha + (\alpha \rightarrow \beta)) \rightarrow \text{Unit} \rightarrow \beta, l_2)\):

\[
\begin{align*}
(\mu, \emptyset). (l_1 : \{ \text{Unit} \rightarrow \text{Int} \})
\end{align*}
\]

\[\text{[R-CLOS]} \quad \exists \alpha \beta. \Big\{ \text{Unit} \rightarrow \text{Int} \rightarrow \text{Unit} \rightarrow \beta \Big\}
\]

\[\text{[R-UNIF]} \quad \exists \alpha. (\mu, \Phi_1). (l_2 : \{(\alpha + (\alpha \rightarrow \text{Int}))\})
\]

Continue by checking the block \(\mu(l_2) = \text{Blk}(l_3, l_4)\):

\[
\begin{align*}
\{ \text{arg}((\alpha + (\alpha \rightarrow \text{Int})) \rightarrow \beta; \gamma \} \\
(\mu, \Phi_2). (l_3 : \{ \beta; \gamma \rightarrow \text{Int} \})
\end{align*}
\]

\[\text{[R-BLK]} \quad \exists \alpha \beta. \gamma. \Big\{ \text{arg}((\alpha + (\alpha \rightarrow \text{Int})) \rightarrow \beta; \gamma \} \\
\end{align*}
\]

\[\text{[R-REF]} \supseteq \exists \beta. \gamma. \langle \text{arg}((\alpha + (\alpha \rightarrow \text{Int})) \rightarrow \beta; \gamma \}
\]

\[\text{[R-REF-SET]} \quad \langle \text{Ref(\_)} \rangle \land \land \text{Ref(\_)}
\]

\[\text{Closures} \quad \text{The rewriting rule [R-CLOS], described in figure 5, allows the verification of a closure } (\tau^\text{sw} \rightarrow \tau^\prime, w) \text{. To test the existence for every expected type } \tau_1 \text{ of a type } \tau^\text{sw}\text{ such that } \tau^\text{sw} \rightarrow \tau_1 \text{ is an instance of the closure static type } \tau^\text{sw} \rightarrow \tau^\prime \rightarrow \tau^\prime, \text{the rewriting rule introduces a unification constraint between } \tau_1 \text{ and an instance of } \tau^\prime \rightarrow \tau^\prime \text{ where every type variable has been replaced by an existential type variable. This instance is written}
\]

\[\tau_i^\prime \rightarrow \tau_i^\prime; \text{the corresponding instance of } \tau^\text{sw} \rightarrow \tau_i^\prime \text{ is written } \tau_i^\text{sw} \rightarrow \tau_i^\prime; \text{and the set of existential variable of } \tau^\text{sw} \rightarrow \tau_i^\prime \rightarrow \tau_i^\prime \text{ is written } \tau_i^\text{sw}. \text{The unification constraints are resolved using the following two rules.}
\]

\[
\begin{align*}
\exists \alpha. \tau_1 = \tau_2 \land C \supseteq \theta(C) \\
\text{[R-UNIF]} \quad \text{if } \theta \text{ is the m.g.u of } \tau_1 \text{ and } \tau_2, \text{and if } \text{dom}(\theta) = \bar{\alpha} \\
\exists \alpha. \tau_1 = \tau_2 \land C \supseteq \text{False} \\
\text{[R-UNIF]} \quad \text{if } \tau_1 \text{ and } \tau_2 \text{ do not unify}
\end{align*}
\]

As for the rule [R-BLK], when a closure is checked for the second time (or more), the rule [R-CLOS] checks the closure against the new expected types and the previously checked types.

\[\text{Administrative rules} \quad \text{To complete the set of rewriting rules there are also four administrative rules that respectively allow the propagation of a failure [R-FALSE], the removal of the trivial constraint [R-TRUE], the removal of heap fragment when there is no more type constraints over the bound pointers [R-HEAP], and the instantiation of the remaining existential variables with the universal type [R-UNIV] (fig. 5),}
\]

\[
\begin{align*}
C \land \text{False} &\supseteq \text{False} \\
C \land \text{True} &\supseteq C \\
(\mu, \Phi), C &\supseteq C \\
\text{if } \text{fp}(C) \neq \text{dom}(\mu) &\supseteq \text{[R-HEAP]}
\end{align*}
\]

\[\text{Example} \quad \text{A complete example is given in figure 6.}
\]
4.3 Rewriting system properties

We have shown that the rewriting system is sound and semi-complete with respect to the relation of compatibility between a value and a type that is defined in section 3.

Theorem 4.1 (Soundness). If \( (\mu, \emptyset), (\ell : \{\tau\}) \Rightarrow \text{True} \) then there exists a typing environment \( \Psi \) such that \( \mu : \Psi \) and \( \Psi(\ell) \Leftrightarrow \tau \).

Theorem 4.2 (Semi-completeness). If \( (\mu, \emptyset), (\ell : \{\tau\}) \Rightarrow \text{False} \) then there exists no typing environment \( \Psi \) such that \( \mu : \Psi \) and \( \Psi(\ell) \Leftrightarrow \tau \).

To prove these theorems, we will use two distinct invariant expressed with a notion of constraint satisfiability.

Constraint satisfiability The first notion of constraint satisfiability allows the proof of soundness of the rewriting system. It is parametrized by a substitution \( \theta \) for the free existential variables of the constraint and a typing environment \( \Psi \) for the free pointers. We write \( \theta, \Psi \vdash C \) when a constraint \( C \) is satisfied by \( \theta \) and \( \Psi \).

The constraint \( \text{True} \) is always satisfied and a conjunction \( C_1 \land C_2 \) is satisfied whenever both \( C_1 \) and \( C_2 \) are satisfied. The constraint \( \text{False} \) can not be satisfied.

\[
\begin{align*}
\text{[C-TRUE]} & : \quad \theta, \Psi \vdash \text{True} & \text{[C-AND]} & : \quad \theta, \Psi \vdash C_1, \theta, \Psi \vdash C_2 \\
\text{[C-EXISTS]} & : \quad \exists \alpha. C \quad \text{[C-QUESTION]} & : \quad \Psi \vdash w : \{\tau_1, \ldots, \tau_n\}
\end{align*}
\]

An existential constraint \( \exists \alpha. C \) is satisfied by \( \theta \) and \( \Psi \) if and only if there exists a type \( \tau \) such that \( C \) is satisfied by \( \theta \) extended with a substitution \( \alpha \mapsto \tau \) and \( \Psi \).

\[
\text{[C-EQUAL]} : \quad \theta(\tau_1) = \theta(\tau_2), \theta, \Psi \vdash \tau_1 = \tau_2
\]

An equality constraint \( \tau_1 = \tau_2 \) is satisfied by \( \theta \) if and only if after instantiation by \( \theta \) the types are syntactically equal. In the same way, an ADT decomposition constraint \( \text{arg}(\tau) = \{\tau_1 : \tau_2\} \) is satisfied by \( \theta \) if and only if after instantiation by \( \theta \) it corresponds to a case of the function \( \text{Arg}(\_\_\_) \); and a head type constructor \( \hat{\tau}_{\_\_\_} \) is satisfied by a substitution \( \theta \) if and only if after instantiation by \( \theta \) all the type share the same head type constructor.

\[
\begin{align*}
\text{[C-ARG]} & : \quad \text{Arg}(\theta(\tau)) = [\theta(\tau) ; \theta(\tau'')] \\
\text{[C-0]} & : \quad \forall i \leq n, \hat{\theta}(\tau_i) = 1 \rightarrow \tau_{i+1} \quad \theta, \Psi \vdash \hat{\theta}(\tau_i) = 1 \rightarrow \tau_i \\
\text{[C-\theta\text{-REF}]} & : \quad \forall i \leq n, \hat{\theta}(\tau_i) = \text{Ref}(\tau'_i) \quad \theta, \Psi \vdash \text{fv}(\tau'_i) = \emptyset
\end{align*}
\]

And the satisfiability of a constraint \( (\mu, \Psi, C) \), \( C \) is defined by the following rule, where \( \Psi' = \lambda(\hat{\theta}(\Phi')) \land \Psi'' \) is a type environment such that for all \( \ell \in \text{dom}(\Psi'') \):

\[
\Psi'(\ell) = \begin{cases} 
\Psi''(\ell) & \text{if } \ell \notin \text{dom}(\Psi') \\
\big( \lambda(\hat{\theta}(\tau)) \land \forall \tau \in \Psi'(\ell) \big) \land \Psi''(\ell) & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
\text{[C-HEAP]} & : \quad \text{dom}(\Psi') = \text{dom}(\mu) \quad \Psi' = \lambda(\hat{\theta}(\Phi')) \land \Psi'' \\
\theta, \Psi \vdash \mu : \Psi' & \Rightarrow \theta, \Psi \vdash \Psi' \models \mu : \Phi' \\
\theta, \Psi \vdash \Psi'' \models C & \Rightarrow \theta, \Psi \vdash (\mu, \Phi', C)
\end{align*}
\]

In other words, a constraint \( (\mu, \Phi', C) \), \( C \) is satisfied when:

- there exists a typing environment \( \Psi' \) more general than the memorized types, that allows the satisfaction of \( C \), and
- there exists a typing environment \( \Psi'' \) compatible with \( \mu \) under the hypothesis \( \Psi' \), such that:
  - \( \Psi''(\ell) \) is an instance of \( \Psi'(\ell) \) for all pointers \( \ell \) of \( \text{dom}(\mu) \) for which there are memorized types, and
  - \( \Psi''(\ell) \) is equal to \( \Psi'(\ell) \) for all the other pointers of \( \text{dom}(\mu) \).

Soundness The satisfiability of the initial constraint correspond to the relation of compatibility between a value and a type (lemma 4.1) and for each rewriting rule, the satisfiability of the right constraint implies the satisfiability of the left constraint (lemma 4.2). These two lemmata allow the proof of soundness of the rewriting system (theorem 4.1).

Lemma 4.1. \( \emptyset, \emptyset \vdash (\mu, \emptyset), (\ell : \{\tau\}) \Rightarrow \text{True} \) and \( \emptyset, \emptyset \vdash (\mu, \emptyset), (\ell : \{\tau\}) \Rightarrow \text{False} \).

Lemma 4.2. If \( C_1 \Rightarrow C_2 \) and \( \theta, \Psi \vdash C_2 \) then \( \theta, \Psi \vdash C_1 \).

To prove the soundness of the rule \([R-\text{BLK}]\), we have to check that the function \( \text{Arg}(\_\_\_) \) is stable by anti-unification (lemma 4.3).

Lemma 4.3. If, given a set of types \( \tau_{1..n} \), sharing all the same head constructor, there exists \( \tau'_1 \) and \( \tau''_1 \) such that \( \text{Arg}(\tau_1) = [\tau'_1; \tau''_1] \), then \( \text{Arg}(\lambda(\tau_{1..n})) = \lambda(\tau'_{1..n}) \).

Semi-completeness The first definition of constraint satisfiability allows the proof of semi-completeness of the rewriting system only if we add an extra application condition for the rules \([R-\text{BLK}]\) and \([R-\text{CLOS}]\) stating that the types to be checked are not instances of previously checked types. However, in presence of existential types such a condition is not the exact negation of the application condition of the rule \([R-\text{REMOVE}]\) and some constraint may remain stuck in a non-trivial form. For example, consider the following constraint:

\[
C_{\text{stack}} = (\mu, \ell \rightarrow (\text{Int} \rightarrow \text{Int})), (\ell : \{\hat{\alpha} \rightarrow \hat{\alpha}\})
\]

We can not decide, without instantiating the variable \( \alpha \), whether the expected type \( \hat{\alpha} \rightarrow \hat{\alpha} \) is an instance or not of the memorized types \( \text{Int} \rightarrow \text{Int} \). We choose not to add this extra condition, allowing the introduction in section 4.4 of rewriting strategies that “unstick” constrained by applying the rule \([R-\text{CLOS}]\).

To prove the semi-completeness without adding the extra condition to the rules \([R-\text{BLK}]\) and \([R-\text{CLOS}]\), we introduce a second definition of constraint satisfiability, called strict satisfiability and written \( \theta, \Psi \models^\text{\text{\&\&}} C \). The set of strictly satisfied constraints is the subset of satisfied constraints for which the memorized types is actually compatible with the heap. More precisely, it can be defined by replacing the rule \([C-\text{HEAP}]\) by the following rule:

\[
\begin{align*}
\text{[C-\text{HEAP}]} & : \quad \text{dom}(\Psi') = \text{dom}(\mu) \quad \Psi' = \lambda(\hat{\theta}(\Phi')) \land \Psi'' \\
\theta, \Psi \vdash \mu : \Psi' & \Rightarrow \theta, \Psi \vdash (\mu, \Phi', C)
\end{align*}
\]
The strict satisfiability of the initial constraint still corresponds to the relation of compatibility between a value and a type (lemma 4.4) and for every rewriting rule, the satisfiability of the left constraint implies the satisfiability of the right constraint (lemma 4.5). Those two lemmata allow the proof of semi-completeness of the rewriting system (theorem 4.2).

Lemma 4.4. \( \emptyset, \emptyset \vdash (\mu, \emptyset), (\ell : \{\tau\}) \) if and only if there exists a typing environment \( \Psi \) such that \( \mu : \Psi \) and \( \Psi(\ell) \approx \tau \).

Lemma 4.5. If \( C_1 \gg C_2 \) and \( \emptyset, \Psi \vdash C_1 \) then \( \emptyset, \Psi \vdash C_2 \).

The second notion of satisfiability does not allow the soundness proof of the rule \([\text{R-HEAP}]\).

**Termination** In the presence of cycles in the memory graph, the rewriting system described in section 4.2 may not terminate. One trivial way to create an infinite rewriting sequence is to apply the rule \([\text{R-BLK}]\) or \([\text{R-CLOS}]\) whenever the rule \([\text{R-REMOVE}]\) is applicable. Such infinite sequences are easily avoided by using rewriting strategies that prefer the latter rule over the former; however, it is not sufficient to avoid all infinite sequences as shown by the following example. Consider, the heap:\[\mu = \{\ell_0 \mapsto \{\text{apply}, \{f : \ell_0 ; x : 0\}\}\}\]

Checking the compatibility of \( \mu \) against the type \( \text{Unit} \rightarrow \text{Int} \) will lead to following infinite sequence:

\[
\begin{align*}
[R-CLOS] & \gg \exists \alpha. (\mu, \Phi_1), \{\ell_0 : \{\alpha \rightarrow \text{Int}\} \rightarrow 0 : \{\alpha\}\} \\
[R-UNIF] & \gg \exists \alpha. (\mu, \Phi_2), \{\ell_0 : \{\beta \rightarrow \text{Int}; \gamma \rightarrow \text{Int}\} \rightarrow 0 : \{\beta; \gamma\}\}
\end{align*}
\]

where:

\[
\begin{align*}
\Phi_1 &= \{\ell_0 \mapsto \{\text{Unit} \rightarrow \text{Int}\}\} \\
\Phi_2 &= \{\ell_0 \mapsto \{\text{Unit} \rightarrow \alpha \rightarrow \text{Int}\}\}
\end{align*}
\]

When applying the rule \([R-CLOS]\) for the second time, and like the constraint \(C_{\text{null}}\), it is not possible to decide without instantiating \(\alpha\) whether \(\alpha \rightarrow \text{Int}\) is an instance of \(\text{Unit} \rightarrow \text{Int}\) or not.

If we cannot prove the termination of the rewriting system in the general case, we have proved its termination in two specific situations: in the absence of cycle (theorem 4.3) and in the absence of existential types (theorem 4.4) as introduced by GADTs or polymorphic closures. We propose in section 4.4 an extension of the rewriting system that terminates but may not be complete in the presence of cycles and existential types.

**Theorem 4.3.** In a heap \( \mu \) that contains no cycle, a constraint \( (\mu, \emptyset), (\ell : \{\tau\}) \) is rewritten in \text{True} or \text{False} in a finite number of steps.

**Theorem 4.4.** In a heap \( \mu \) that contains no polymorphic closure that may introduce existential types, a constraint \( (\mu, \emptyset), (\ell : \{\tau\}) \) is rewritten in \text{True} or \text{False} in a finite number of step.

To prove the termination in presence of cycles, but in the absence of existential types, we defined the size of a constraint and showed that, for every rewriting rule, the right constraint is smaller than the left constraint. More precisely, the size of a constraint is a 6-uple, sorted in lexicographic order, and composed of the number of pointers without memoized type, the number of type constructors in the anti-unifiers of memoized type’s sets, the number of type constraints, the number of types in the sets of memoized types and in the sets of types constraints, the number of homogeneity and decomposition constraints, and the number of unification constraints.

### 4.4 Rewriting strategy

In order to express precisely the topological traversal of the graph and to enable termination in cycles and existential types, we now add one rewriting rule in order to topologically sort the heap, and two rules dedicated to ensure termination. The two latter rules imply loosing completeness.

**Topological sort** To express the topological traversal proposed in section 2.1 as a rewriting strategy, the following rule allows the split of the heap in fragments corresponding to SCCs.\[ (\mu, \Phi), C \gg (\mu_1, \Phi_1), (\mu_2, \Phi_2), C' \quad [\text{R-SORT}] \]

When a heap is decomposed in SCCs, the rules \([R-BLK]\) and \([R-CLOS]\) syntactically forbid to check a value that does not belong to the current SCC. Once a SCC has been checked, it can be removed from the constraints with the rule \([R-HEAP]\).

**Monomorphic recursion** As shown in section 2.1, when we restrict ourselves to monomorphic recursion, it is not necessary to check an allocated cyclic value twice when the memory graph is traversed in a topological order: it is sufficient to check the equality between the expected type and the memoized one. Hence, we never apply the rules \([R-BLK]\) or \([R-CLOS]\) for previously checked pointers, and we replace the rule \([R-REMOVE]\) by the following checked rule, that allows the unification of the expected type with the memoized one:

\[
\begin{align*}
\mu, \Phi', (\ell : \{\tau'\} \land C) & \gg [\text{R-FORCEUNIF}] \\
& \tau' = \tau' \land (\mu, \Phi'), C' \\
& \text{if } \Phi'(\ell) = \{\tau\}
\end{align*}
\]

This rule is sound (as for lemma 4.2) but not semi-complete (as for lemma 4.5).

If a set of expected types or a set a memoized types could not be simplified by anti-unification, the following rule allows the instantiation of an existential variables with the universal type.\[ \exists \alpha. C \gg [\text{R-FORCEINST}] \]

This rule is obviously not semi-complete, hence it is applied only when no other rules apply.

**Polymorphic recursion** If we want to accept polymorphic recursion, we may need to check cyclic values several times, and accept that the types expected from inside the SCC be instances of the memoized types. In order to do so, we allow the usage of the rule \([R-REMOVE]\). Guaranteeing termination may then be obtained by allowing multiple checks of cyclic values (rules \([R-BLK]\) and \([R-CLOS]\)) only when we may anti-unify\(^5\) the expected types with memoized types.

### 4.5 Type-safe (un)marshalling primitives

We have implemented a prototype of this algorithm for the OCaml compiler. This provides a type-safe unmarshalling function based

\[^5\text{Remember that the anti-unifier may not always be computed in the presence of existential types.}\]

---

\[^4\text{This heap may represent } \text{A.loop in the OCaml program:}\]

```ocaml
module rec A : sig
  val loop : Unit \rightarrow \alpha
end = struct
  let delay f x = fun () \rightarrow f x
  let loop = delay A.loop ()
end
```

---

Submitted to ICFP '12

2012/8/12
on the unsafe marshalling mechanism available in the OCaml standard library without changing the external representation of data. The current unsafe (un)marshalling functions of the standard library have the following type signatures, where the type String is used as a sequence of bytes:

\[
\begin{align*}
\text{val } & \text{marshal} : \alpha \rightarrow \text{String} \\
\text{val } & \text{unmarshal} : \text{String} \rightarrow \alpha
\end{align*}
\]

The type-safe unmarshalling function has the following prototype:

\[
\text{val safe\_unmarshall} : \text{Ty}(\alpha) \rightarrow \text{String} \rightarrow \alpha
\]

where \(\text{Ty}(\tau)\) is the classical singleton type whose unique value is a runtime representation of the type \(\tau\) [6, 12].

Adapting our algorithm to other runtimes The compatibility algorithm works on a memory graph where pointers and immediate values can be distinguished. In situations where the runtime does not make such a distinction but still uses an exact garbage collection mechanism that traverse precisely the memory graph, our algorithm remains applicable. We would ask the generic marshalling function to add the necessary information to the external representation of data. Our algorithm would then be applied on an intermediate form of the memory graph.

For checking the compatibility of closures, our algorithm asks the running program to provide the static type of the code pointers being unmarshalled. This has been implemented in our prototype.

Note that type-safe unmarshalling of closures allows marshalling unevaluated lazy values.

The OCaml external representation of closures is composed of a raw code pointer and the representation of the environment. Hence, the unmarshalling of a closure can only take place in different instances of the same compiled program. In situations where the runtime uses dynamic loading of code and relinking when unmarshalling closures, we would again delegate this task to the generic unmarshaller, and check type compatibility on the resulting memory graph.

5. Related work

Type-safe (un)marshalling in OCaml When compared to existing OCaml libraries that provide type-safe marshalling mechanisms, our proposal is the first to provide both the ability to communicate safely with untrusted peers—such as a web browser for an HTTP server—and to handle GADTs and closures with the help of existential type variables. The Deriving library[26] use the Camlp4 preprocessor to generate \(\text{ad hoc}\) functions from data type definitions. As other mechanisms based on \(\text{ad hoc}\) functions we know of, it allows the communication with untrusted peers but do not handle closures and existential types.

Our proposal does not allow the (un)marshalling of abstract data types introduced by the OCaml module system, whereas the Hash- Caml [5] compiler or the Quicksilver library [21] do. The Hash-Caml compiler is an extension of the OCaml compiler, that adds runtime type representations. It contains a safe (un)marshalling mechanism that keeps the type of a value in its external representation. For representing abstract types, it computes a hash of the type and of the source code of the associated function. Quicksilver is another Camlp4 based marshalling library that generates \(\text{ad hoc}\) (un)marshalling functions. It allows the user to associate a cryptographic key to each abstract type and to encrypt the external representation of an abstract value.

Polytypic programming Another approach to type-safe serialization is polytypic programming [24]. To motivate a programming language that allows the explicit manipulation and analysis of type expressions in a typed manner, Weirich [25] uses a generic marshalling function that recursively analyzes the type of the value. This typed approach has the obvious advantage of not requiring an external proof of soundness. However, it may not be possible to handle polymorphic sharing with anti-unification of the expected types.

Type reconstruction The idea of traversing a memory graph in parallel with its type has been previously used to build tag-free garbage collectors [3, 4, 10, 23] or to implement debuggers [2]. In these contexts, the memory graph is known to be compatible with the type. Our algorithm may also be used to pretty-print values in a debugger. Our approach has the advantage of being less intrusive and does not require to keep runtime type information in the stack or inside the values.

Interoperability The TypedRacket programming language [22] is a statically typed version of Racket. Both languages share the same runtime and they use dynamic type compatibility checking to make sure that untyped data can be safely imported in typed parts of a program. The main differences with our algorithm is that compatibility checking is based on contracts [8] and checking of functional values is delayed until their application.

6. Conclusion

We have presented a type compatibility checking algorithm for ML-like programming languages, that handles algebraic data types, mutable data, cyclic data, GADTs and closures. We gave a formal description of the problem, and presented the verification as a rewriting system, the algorithm being essentially a particular rewriting strategy.

This systematic approach greatly simplifies the proof of soundness: the soundness of each rewriting rule is proved independently. The naive strategy being incomplete, the presentation as a rewriting system enables the precise identification of where completeness is lost and what step to follow in order to minimize the chances of rejecting correct data in a finite time.

The obtained algorithm is efficient and has been implemented in a prototype version of the OCaml compiler. It needs no type information in the data representation and can be used to import, in a statically typed program, external data built by untyped or differently-typed programs.

References


