Typing Unmarshalling without Marshalling Types

Grégoire Henry
CNRS, PPS, UMR 7126
Univ Paris Diderot
Sorbonne Paris Cité
F-75205 Paris, France
henry@pps.univ-paris-diderot.fr

Michel Mauny
ENSTA-ParisTech
32, boulevard Victor,
F-75739 Paris Cedex 15, France
Michel.Mauny@ensta.fr

Emmanuel Chailloux
LIP6 - UMR 7606
Université Pierre et Marie Curie
Sorbonne Universités
75005 Paris, France
Emmanuel.Chailloux@lip6.fr

Pascal Manoury
Université Pierre et Marie Curie
PPS, UMR 7126 CNRS
Univ Paris Diderot
Sorbonne Paris Cité
F-75205 Paris, France
Pascal.Manoury@pps.univ-paris-diderot.fr

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 Program*]: 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-

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

rays. This algorithm is efficient: it is linear in the size of the graph in presence of monomorphic or polymorphic sharing and quasi-linear in presence of cycles. The proposed algorithm is sound and semi-complete, however, in the most general case, it may not terminates. In practice, we have implemented a prototype tailored for the OCaml compiler that always terminates and seems sufficiently complete.

In section 2, we informally describe the algorithm in terms of graph traversal. In section 3 we define the compatibility relation between a memory graph and a type, then in section 4 we describe the algorithm as a rewriting system and study its soundness and completeness with respect to the previous compatibility relation. In section 5 we compare our proposal with other type-safe (un)marshalling primitives.

2. Type-checking a memory graph: discussion

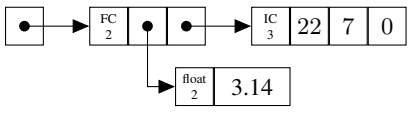
Our algorithm has been initially conceived for the OCaml runtime which uses a tagged, uniform representation for values. The description of the algorithm will be done for such a representation but in section 4.5 we will discuss how it may be reused in a runtime with different representations for values.

Uniform representation The uniform representation of OCaml runtime values uses a bit tag that allows distinction between an immediate value and a pointer to an allocated value (a memory block). Furthermore, every allocated value has a generic header containing its size and a tag. The tag is a small integer mainly used to discriminate between cases of an algebraic data type (hereafter abbreviated ADT) but there are also some tags reserved for specific allocated values such as character strings, closures or boxed floats[15]. The latter tags specify that the content of the value does not follow the uniform representation and hence should not be traversed by the garbage collector.

More precisely, integers are represented by immediate values and characters are represented by immediate values lower than 256. Records and tuples are represented by allocated values of tag 0. A constant constructor is represented as an immediate value that is the position of the constructor in the corresponding type definition. A non-constant constructor is an allocated value whose tag also depends on the position of the constructor in the corresponding type definition. For example, consider the following list of rational numbers:

```
type Rlist =
| Nil
| IC of Int * Int * Rlist
| FC of Float * Rlist
let l : Rlist = FC (3.14, IC (22, 7, Nil))
```

It is represented by the memory graph in figure 1 where the tags of allocated blocks are represented with a symbolic name instead of an integer.



Concrete tags: IC = 0; FC = 1; float = 253.

Figure 1. Uniform data representation in OCaml

Generic (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:

$\texttt{type List}(\alpha) = \texttt{Nil} \mid \texttt{Cons of}\ \alpha * \texttt{List}(\alpha)$

A memory graph is compatible with $\texttt{List}(\texttt{Int})$ 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 $\texttt{List}(\texttt{Int})$.

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 $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, our algorithm traverses e_2—in order to collect all the expected types for *x*—before traversing e_1.

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 complexify 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. Then, in such a situation, we can assume while checking a memory graph that all the expected types for that shared value are equal to that 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:

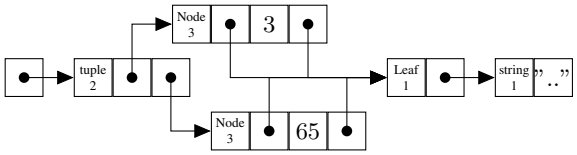
- traverse the graph in a topological order, hence collecting all the types expected for a shared value before traversing it,
- check a shared value only once against the anti-unifier of all its expected types.

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

$\texttt{type}\ \texttt{Tree}(\alpha)\ =$
$|\ \texttt{Leaf of String}$
$|\ \texttt{Node of Tree}(\alpha) * \alpha * \texttt{Tree}(\alpha)$

$\texttt{let}\ \textit{leaf} : \forall\alpha.\, \texttt{Tree}(\alpha) = \texttt{Leaf}\ \text{"..."}$
$\texttt{let}\ \textit{trees} : (\texttt{Tree(Int)} * \texttt{Tree(Char)}) =$
$(\ \texttt{Node}\ (\textit{leaf}, 3, \textit{leaf}),\ \texttt{Node}\ (\textit{leaf},\ \texttt{'A'},\ \textit{leaf})\ )$

The set of expected types collected for *leaf* while checking the compatibility of *trees* against the type $(\texttt{Tree(Int)} * \texttt{Tree(Char)})$ is $\{\texttt{Tree(Int)}; \texttt{Tree(Char)}\}$. Their anti-unifier is $\forall\alpha.\, \texttt{Tree}(\alpha)$.



Concrete tags: tuple = Leaf = 0; Node = 1; string = 252.

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 with their expected type when traversing them for the first time; and then, when checking a previously marked value against a type that is not an instance of the memorized one, to recheck the value against the anti-unifier of the two types, while marking the value with the anti-unifier. That process will reach a fixed point: each anti-unification reduces the size of the memorized type[1] and the smallest type is the empty type: $\forall\alpha.\, \alpha$. Hence, the number of iterations for a value is limited by the size of the first expected type.

The proposed process is complete even in the presence of polymorphic recursion but is not easy to implement efficiently. Furthermore, in the absence of polymorphic recursion in the source language, it should not be necessary to check values involved in a cycle twice: the types collected from outside the strongly connected components (hereafter abbreviated SCC) should allow the algorithm to recompute a sufficiently general type for the *roots* of the SCC. Hence, we initially restrict ourselves to monomorphic recursion and in our prototype implementation we apply the following strategy, assuming that the graph has been previously annotated with the number of references towards a shared value:

(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 Mutable 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 considered as mutable. In that case, we check that the expected type for the mutable field is monomorphic, in order to avoid the well-know 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 recurse. For example, consider the following simple type definition:

$\texttt{type}\ \texttt{T}_1(\_)\ =$
$|\ \texttt{Int} : \texttt{Int} \to \texttt{T}_1(\texttt{Int})$
$|\ \texttt{Char} : \texttt{Char} \to \texttt{T}_1(\texttt{Char})$
$|\ \texttt{Bin} : \texttt{T}_1(\alpha) * \texttt{T}_1(\alpha) \to \texttt{T}_1(\alpha)$

Checking the compatibility of an allocated value of tag 1 and size 1, representing the constructor `Char`, against type $\texttt{T}_1(\texttt{Int})$ will fail while unifying $\texttt{T}_1(\texttt{Int})$ with the return type of the constructor `Char`, i. e. $\texttt{T}_1(\texttt{Char})$. On the contrary, while checking the compatibility of an allocated value of tag 2 and size 2, representing `Bin`, against type $\texttt{T}_1(\texttt{Int})$, the unification allows the instantiation

[1] at least as long as we do not consider recursive type expression.

of α and the recursive checking of the sub-values against the type $\texttt{T}_1(\texttt{Int})$.

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

$$\begin{array}{l}\texttt{type T}_2(\_) \ = \\ \mid \ \texttt{Int} : \texttt{Int} \to \texttt{T}_2(\texttt{Int}) \\ \mid \ \texttt{Char} : \texttt{Char} \to \texttt{T}_2(\texttt{Char}) \\ \mid \ \texttt{Couple} : \texttt{T}_2(\alpha) * \texttt{T}_2(\beta) \to \texttt{T}_2(\alpha * \beta) \\ \mid \ \texttt{Fst} : \texttt{T}_2(\alpha * \beta) \to \texttt{T}_2(\alpha)\end{array}$$

While checking the compatibility of an allocated value of tag 3 and size 1, representing `Fst`, against the type $\texttt{T}_2(\texttt{Int})$, the unification leaves the β uninstantiated. Then, the recursive call has to verify that there exists a type β such that the argument of `Fst` is compatible with $\texttt{T}_2(\texttt{Int} * \beta)$. 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 $\texttt{T}_2(\texttt{Int})$?
- is the representation of `Char 'A'` is compatible with $\texttt{T}_2(\beta)$?

The latter will instantiate β 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:

$$\begin{array}{l}\texttt{type Ty}(\_) \ = \\ \mid \ \texttt{TBool} : \texttt{Ty}(\texttt{Bool}) \\ \mid \ \texttt{TInt} : \texttt{Ty}(\texttt{Int}) \\ \mid \ \texttt{TChar} : \texttt{Ty}(\texttt{Char}) \\ \texttt{type Dyn} \ = \ \texttt{Dyn} : \alpha * \texttt{Ty}(\alpha) \to \texttt{Dyn}\end{array}$$

Testing the representation of $\texttt{Dyn}\ (65,\ \texttt{TInt})$ against the type `Dyn` with the recursive algorithm leads to the following question: is there a type β such that:

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

There are at least two possible instantiations of β 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 β 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 $\texttt{T}_2(\_)$ but where the constructors `Int` and `Char` have been replaced by a generic constructor `Const`:

$$\begin{array}{l}\texttt{type T}_3(\_) \ = \\ \mid \ \texttt{Const} : \alpha \to \texttt{T}_3(\alpha) \\ \mid \ \texttt{Couple} : \texttt{T}_3(\alpha) * \texttt{T}_3(\beta) \to \texttt{T}_3(\alpha * \beta) \\ \mid \ \texttt{Fst} : \texttt{T}_3(\alpha * \beta) \to \texttt{T}_3(\alpha)\end{array}$$

While checking the compatibility of a memory graph representing $\texttt{Fst}\ (\texttt{Couple}\ (\texttt{Const}\ 3, \texttt{Const 'A'})\ )$ against $\texttt{T}_3(\texttt{Int})$, the test will be reduced to:

- is there a type β such that the immediate value 65, representing the character `'A'`, is compatible with β?

While answering this question on the paper is trivial, our algorithm could not delay the verification anymore nor instantiate β 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:

$$\begin{array}{l}\texttt{let } \textit{one} : \texttt{Int} = 1 \\ \texttt{let } \textit{succ} : \texttt{Int} \to \texttt{Int} = \lambda x.\ x + \textit{one} \\ \texttt{let } \textit{delayed\_apply} : (\alpha \to \beta) \to \alpha \to \texttt{Unit} \to \beta = \\ \ \lambda f.\ \lambda x. \\ \quad \texttt{let } \textit{apply} : \texttt{Unit} \to \beta = \lambda().\ (f\ x)\ \texttt{in} \\ \quad \textit{apply}\end{array}$$

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

$$\begin{array}{rcl}\sigma_{\textit{succ}} & = & [\textit{one} \mapsto \texttt{Int}] \to \texttt{Int} \to \texttt{Int} \\ \sigma_{\textit{delayed\_apply}} & = & \forall\alpha\beta.\,[\,] \to (\alpha \to \beta) \to \alpha \to \texttt{Unit} \to \beta \\ \sigma_{\textit{apply}} & = & \forall\alpha\beta.\,[f \mapsto (\alpha \to \beta); x \mapsto \alpha] \to \texttt{Unit} \to \beta\end{array}$$

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

$$\texttt{let } \textit{dsucc} : \texttt{Unit} \to \texttt{Int} = \textit{delayed\_apply succ 52}$$

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

$$[f \mapsto (\texttt{Int} \to \texttt{Int}); x \mapsto \texttt{Int}] \to \texttt{Unit} \to \texttt{Int}$$

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.

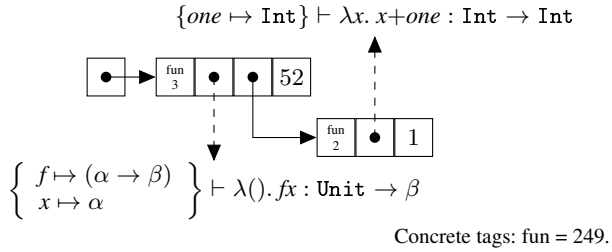


Figure 3. Representation of the function $dsucc$

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 $\alpha \to \texttt{Int}$ and $\texttt{Bool} \to \beta$ where α and β are existential types, the anti-unifier depends on the future instantiation of α and β: if α is later instantiated in `Int` and β in `Bool`, the anti-unifier is $\forall\gamma.\gamma \to \gamma$; but if α in instantiated in `Bool` and β in `Int`, it is $\texttt{Bool} \to \texttt{Int}$; and for any other substitutions, it is $\forall\gamma\delta.\gamma \to \delta$. 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 $\forall\alpha.\,\alpha \to \alpha$ and the set of memorized types contains $\alpha \to \texttt{Int}$ and $\texttt{Bool} \to \beta$, the test is valid only if α is later instantiated as `Int` and β in `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:

$\texttt{type T} \;=\; \texttt{None} \mid \texttt{Some of}\ (\texttt{T} \to \texttt{Int})$
$\texttt{let}\ f : \texttt{T} \to \texttt{Int} = \lambda x.\ \texttt{match}\ x\ \texttt{with}$
$\quad \mid \texttt{None} \to 0$
$\quad \mid \texttt{Some}\ g \to g\ \texttt{None}$
$\texttt{let}\ h : \texttt{Unit} \to \texttt{Int} = \textit{delayed\_apply}\ f\ (\texttt{Some}\ f)$

Verifying the compatibility of the closure h against the type $\texttt{Unit} \to \texttt{Int}$ will be recursively decomposed into the question of knowing if there is an α such that:

- the representation of f is compatible with $\alpha \to \texttt{Int}$, and
- the representation of $\texttt{Some}\ f$ is compatible with α?

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 $\texttt{Some}\ f$ to solve the second part until α is instantiated. In such situation, our algorithm *forces* the verification of $\texttt{Some}\ f$ by introducing an existential type for every sub-value of the allocated block representing $\texttt{Some}\ f$—here it will introduce an existential type β for f—and it memorizes an explicit *constraint* to be solved when α is instantiated: is an allocated value with the tag of `Some` and with a single sub-value of type β compatible with α?

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.

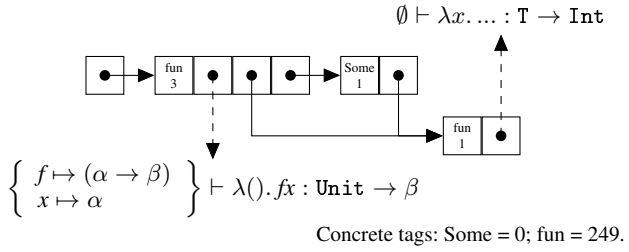


Figure 4. Representation of the function h

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:

$$\begin{array}{rll} \tau & ::= & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Unit} \mid (\tau * \tau) \mid \texttt{List}(\tau) \mid \tau \to \tau \\ & \mid & \texttt{Ref}(\tau) \qquad\qquad \textit{reference} \\ & \mid & \alpha \qquad\qquad \textit{type variable} \\ & \mid & \star \qquad\qquad \textit{universal type} \end{array}$$

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 $\mathit{fv}(\tau)$ the set of variables of τ and we write $\tau_1 \preccurlyeq \tau_2$ when τ_2 is an instance of τ_1, meaning there exists a substitution θ such that $\theta(\tau_1) = \tau_2$.

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

$$\begin{array}{rll} w & ::= & i \qquad\qquad \textit{integer} \\ & \mid & \ell \qquad\qquad \textit{pointer} \end{array}$$

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

$$\begin{array}{rll} h & ::= & \texttt{Blk}(w,w) \qquad\qquad \textit{block} \\ & \mid & \langle \tau, w \rangle \qquad\qquad \textit{closure} \end{array}$$

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 $\mathit{fp}(\mathit{img}(\mu))$ for the set of pointers occurring in the image of μ. A value (ℓ, μ) is well defined when $\mathit{fp}(\mathit{img}(\mu)) \cup \{\ell\}$ is included in the domain of μ.

Typing rules A typing environment Ψ 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 ℓ is compatible with a type τ in an environment Ψ if τ is an instance of $\Psi(\ell)$.

$$\frac{\Psi(\ell) \preccurlyeq \tau}{\Psi \vdash \ell : \tau} \quad [\text{W-LABEL}]$$

The compatibility of an integer with a type is abstracted by a function $Imm(\tau)$ that returns the set of immediate values compatible with τ. It is defined by case analysis on the head type constructor:

$$\begin{array}{rcll} Imm(\texttt{Unit}) & = & \{0\} & \text{[IMM-UNIT]} \\ Imm(\texttt{List}(\tau)) & = & \{0\} & \text{[IMM-EMPTYLIST]} \\ Imm(\texttt{Bool}) & = & \{0;1\} & \text{[IMM-BOOL]} \\ Imm(\texttt{Int}) & = & \{\ldots;-1;0;1;\ldots\} & \text{[IMM-INT]} \\ Imm(\star) & = & \{\ldots;-1;0;1;\ldots\} & \text{[IMM-UNIV]} \\ Imm(\tau) & = & \emptyset & \textit{otherwise} \end{array}$$

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

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

$$\begin{array}{rcll} Arg(\texttt{List}(\tau)) & = & [\tau\,;\,\texttt{List}(\tau)] & \text{[ARG-LIST]} \\ Arg((\tau_1 * \tau_2)) & = & [\tau_1\,;\,\tau_2] & \text{[ARG-COUPLE]} \\ Arg(\texttt{Ref}(\tau)) & = & [\tau\,;\,\texttt{Unit}] & \text{[ARG-REF]} \end{array}$$

$$\frac{Arg(\tau) = [\tau_1\,;\,\tau_2] \quad \Psi \vdash w_1 : \tau_1 \quad \Psi \vdash w_2 : \tau_2}{\Psi \vdash \texttt{Blk}(w_1, w_2) : \tau} \quad \text{[H-BLK]}$$

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].

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

A closure $\langle \tau, w \rangle$ is compatible with a type $\tau_1 \to \tau_2$ if there exists a type τ_{env} compatible with the captured environment and such that $\tau_{env} \to \tau_1 \to \tau_2$ is an instance of the static type of the original code.

$$\frac{\tau \preccurlyeq \tau_{env} \to \tau_1 \to \tau_2 \qquad \Psi \vdash w : \tau_{env}}{\Psi \vdash \langle \tau, w \rangle : \tau_1 \to \tau_2} \quad \text{[H-CLOS]}$$

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 μ is partially compatible with type environment Ψ under the hypothesis Ψ', and we will write $\Psi' \vdash \mu : \Psi$, if and only if:

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

We will say that a heap μ is compatible with an environment Ψ and we will write $\mu : \Psi$ if and only if $\Psi \vdash \mu : \Psi$. We will say that a value (w, μ) is compatible with a type τ if there exists an environment Ψ 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[20]. 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:

$$\begin{array}{rcll} C & ::= & \texttt{True} & \textit{trivial constraint} \\ & | & \texttt{False} & \textit{failure} \\ & | & C \wedge C & \textit{conjunction} \\ & | & (\mu, \Phi).\, C & \textit{heap fragment and memorized types} \\ & | & w : \{\tau\,;\ldots\} & \textit{type constraint} \\ & | & \exists\dot{\alpha}.\, C & \textit{existential quantification} \\ & | & \tau = \tau & \textit{unification} \\ & | & \texttt{arg}(\tau) = [\tau\,;\,\tau] & \textit{ADT decomposition} \\ & | & \Diamond\{\tau\,;\ldots\} & \textit{head type constructors homogeneity} \end{array}$$

where Φ 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 $\Phi \cup \{\ell : \{\tau;\ldots\}\}$ to denote the environment obtained by replacing in Φ the set associated to ℓ by the extended one $\Phi(\ell) \cup \{\tau;\ldots\}$, i.e. the environment $\Phi \oplus \{\ell : \Phi(\ell) \cup \{\tau;\ldots\}\}$.

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, μ) against a type τ is:

$$(\mu, \emptyset).\, w : \{\tau\}$$

The normal forms will be the constraints `True` and `False`.

The existential quantification $\exists\dot{\alpha}.\, C$ 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 $fev(\tau)$ (respectively $fuv(\tau)$) 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 $\dot{\theta}$.

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

The construction $(\mu, \Phi).\, C$ binds the pointer from $dom(\mu)$ into C and into the image of μ. 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 $C_1 \gg C_2$, 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.

[2] On a concrete setting, the function $Arg(\_)$ receives also the tag of the block being checked and returns the types of the components of the corresponding data constructor.

[3] Requiring the existence of types compatible with the components slightly restricts the set of memory graphs compatible with a type. However, it simplifies the algorithm in the situation described in section 2.4 where it is required to force the verification of a block against existential types.

Allocated values

$$(\mu,\Phi').(\ell:\{\tau_{1..n}\}\wedge C)\gg\exists\dot\alpha_{1..m}.\dot\beta_{1..m}.\begin{cases}\Diamond\{\tau_{1..m}\}\\ \bigwedge_{i=1..m}\mathtt{arg}(\tau_i)=[\dot\alpha_i\,;\,\dot\beta_i]\\ (\mu,\Phi'\cup\{\ell:\{\tau_{1..n}\}\}).(w':\{\dot\alpha_{1..m}\}\wedge w'':\{\dot\beta_{1..m}\}\wedge C)\end{cases}\quad\text{[R-BLK]}$$
$$\text{if } \mu(\ell)=\mathtt{Blk}(w',w'') \text{ and } \Phi'(\ell)=\{\tau_{n+1..m}\}$$
$$\text{and } \{\dot\alpha_{1..m}\,;\,\dot\beta_{1..m}\}\#fev(\{\tau_{1..m}\})\cup fev(\Phi')\cup fev(C)$$

$$(\mu,\Phi').(\ell:\{\tau_{1..n}\}\wedge C)\gg\exists\bar\dot\alpha_{1..m}.\begin{cases}\bigwedge_{i=1..m}\tau_i=\dot\tau_i'\to\dot\tau_i''\\ (\mu,\Phi'\cup\{\ell:\{\tau_{1..n}\}\}).(C\wedge w:\{\dot\tau^{env}_{1..m}\})\end{cases}\quad\text{[R-CLOS]}$$
$$\text{if } \mu(\ell)=\langle\tau^{env}\to\tau'\to\tau'',w\rangle \text{ and } fv(\tau^{env}\to\tau'\to\tau'')=\bar\alpha$$
$$\text{and } \Phi'(\ell)=\{\tau_{n+1..m}\} \text{ and } \bar\dot\alpha_{1..m}\#fev(\tau_{1..m})\cup fev(\Phi')\cup fev(C)$$

Parametricity

$$\exists\bar\dot\alpha_i\bar\dot\beta\bar\dot\gamma.\,\overline{\Diamond\{\dot\alpha_{1..n}\}}\wedge\overline{\Diamond\{\mathtt{Ref}(\tau)\}}\wedge\overline{\mathtt{arg}(\dot\beta)=[\tau'\,;\,\tau'']}\wedge\overline{i:\{\dot\gamma\}}\gg\mathtt{True}\qquad \text{if } \overline{fuv(\tau)=\emptyset}\quad\text{[R-UNIV]}$$

Figure 5. 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:

$$\exists\bar\dot\alpha.\,\overline{\tau=\tau}\wedge\overline{\Diamond\{\bar\tau\}}\wedge\overline{\mathtt{arg}(\tau)=[\tau\,;\,\tau]}\wedge\overline{(\mu,\Phi)}.\ \left(\overline{\ell:\{\bar\tau\}}\right)$$

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 without existential variables to a subset of universal type variables. We write $\tau\curlywedge\tau'$ for the anti-unifier of τ and τ' defined by the following rules of congruence:

$$\begin{array}{rcll}\mathtt{Int}&\curlywedge&\mathtt{Int}&=\mathtt{Int}\\ \mathtt{Unit}&\curlywedge&\mathtt{Unit}&=\mathtt{Unit}\\ \mathtt{List}(\tau)&\curlywedge&\mathtt{List}(\tau')&=\mathtt{List}(\tau\curlywedge\tau')\\ \mathtt{Ref}(\tau)&\curlywedge&\mathtt{Ref}(\tau')&=\mathtt{Ref}(\tau\curlywedge\tau')\\ (\tau_1*\tau_2)&\curlywedge&(\tau_1'*\tau_2')&=((\tau_1\curlywedge\tau_1')*(\tau_2\curlywedge\tau_2'))\\ (\tau_1\to\tau_2)&\curlywedge&(\tau_1'\to\tau_2')&=(\tau_1\curlywedge\tau_1')\to(\tau_2\curlywedge\tau_2')\\ \tau&\curlywedge&\tau'&=au(\tau,\tau')\quad otherwise\end{array}$$

Modulo a renaming of the produced type variables, the definition of anti-unification is associative and commutative. Given a set of types $\tau_{1..n}$, we write $\curlywedge\{\tau_{1..n}\}$ for its principal anti-unifier.

Property 4.1. *We have* $(\tau_1\curlywedge\tau_2)\preccurlyeq\tau_1$ *and* $(\tau_1\curlywedge\tau_2)\preccurlyeq\tau_2$.

Property 4.2. *If* $\tau\preccurlyeq\tau_1$ *and* $\tau\preccurlyeq\tau_2$, *then* $\tau\preccurlyeq(\tau_1\curlywedge\tau_2)$.

Sharing The rewriting rule [R-MERGE] allows the merge of two sets of expected types for the same allocated value, while the [R-AUNIF1] 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.

$$\ell:\{\tau_{1..n}\}\wedge\ell:\{\tau_{n+1..m}\}\gg\ell:\{\tau_{1..m}\}\quad\text{[R-MERGE]}$$
$$w:\{\tau_1\,;\,\tau_2\,;\,\tau_{3..n}\}\gg w:\{\tau\,;\,\tau_{3..n}\}\quad\text{[R-AUNIF1]}$$
$$\text{if } \forall\dot\theta.\,\dot\theta(\tau)=\dot\theta(\tau_1)\curlywedge\dot\theta(\tau_2)$$

The rewriting rule [R-AUNIF2] allows the simplification of the set of memorized types for an allocated value by replacing two of the types by their principal anti-unifier. As for the rule [R-AUNIF1], it applies only if the anti-unifier is independent of the future instantiation of existential type variables.

$$(\mu,\Phi'\cup\{\ell:\{\tau_1\,;\,\tau_2\}\}).C\gg(\mu,\Phi'\cup\{\ell:\{\tau\}\}).C\quad\text{[R-AUNIF2]}$$
$$\text{if } \forall\dot\theta.\,\dot\theta(\tau)=\dot\theta(\tau_1)\curlywedge\dot\theta(\tau_2)$$

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 $\dot\theta$, the anti-unifier of the instantiations of the expected types by $\dot\theta$ is an instance of the anti-unifier of the instantiations of the memorized types by $\dot\theta$.

$$(\mu,\Phi').(\ell:\{\tau_{1..n}\}\wedge C)\gg(\mu,\Phi').C\quad\text{[R-REMOVE]}$$
$$\text{if } \ell\in dom(\Phi')$$
$$\text{and } \forall\dot\theta.\ \curlywedge\{\dot\theta(\Phi'(\ell))\}\preccurlyeq\curlywedge\{\dot\theta(\tau_{1..n})\}$$

Immediate values The rewriting rules for immediate values simply reflect the function $Imm(\tau)$ and state that the verification against an existential type is delayed.

$$i:\{\tau\}\gg\mathtt{True}\qquad \text{if } i\in Imm(\tau)\quad\text{[R-INT]}$$
$$i:\{\tau\}\gg\mathtt{False}\qquad \text{if } \tau\not\equiv\dot\alpha \text{ and } i\notin Imm(\tau)\quad\text{[R-INT-FAIL]}$$

While checking an immediate value against a set of expected types that can not be anti-unified because of existential type variable, it is sufficient to ensure its compatibility against one of the types and to ensure that all the types share the same head type constructor. As the function $Imm(\_)$ is defined by case analysis of the head constructor of its type argument, this will ensure that the value is compatible with the anti-unifier of the expected types.

$$i:\{\tau_{1..n}\}\gg\Diamond\{\tau_{1..n}\}\wedge i:\{\tau_1\}\qquad \text{if } n\geq 2\quad\text{[R-INT-SET]}$$

Blocks The rewriting rule [R-BLK] described in figure 5, allows the verification of an allocated block $\mathtt{Blk}(w',w'')$. It introduces two existential variables $\dot\alpha_i$ and $\dot\beta_i$ for each expected type τ_i and ensures that they correspond to the types expected for the arguments of a block of type τ_i, as computed by the function $Arg(\tau_i)$. It also generates types constraints $w':\{\dot\alpha_1;\ldots;\dot\alpha_n\}$ and $w'':\{\dot\beta_1;\ldots;\dot\beta_n\}$.

The explicit introduction of the constraint $\mathtt{arg}(\tau_i)=[\dot\alpha_i\,;\,\dot\beta_i]$ allows the application of the rule [R-BLK] even when the set of expected types only contains existential types variables. This constraint is resolved by the rules [R-ARG] and [R-ARG-FAIL] that simply reflect the function $Arg(\tau)$ and state that the verification against an existential type is delayed.

$$\mathtt{arg}(\tau)=[\tau'\,;\,\tau'']\gg\tau'=\tau_1\wedge\tau''=\tau_2\quad\text{[R-ARG]}$$
$$\text{if } Arg(\tau)=[\tau_1\,;\,\tau_2]$$
$$\mathtt{arg}(\tau)=[\tau'\,;\,\tau'']\gg\mathtt{False}\quad\text{[R-ARG-FAIL]}$$
$$\text{if } \tau\not\equiv\dot\alpha \text{ and } \tau\notin dom(Arg)$$

The rule [R-BLK] also adds the set of expected types τ_i in the environment of memorized types Φ. Without specific rewriting strategy, it is possible to start the verification of a block without collecting all the expected types. Then, to handle the situation where a block is checked for the second time (or more), the rule [R-BLK] checks the block against the new expected types *and* the previously checked types. This is required in the presence of existential types to verify that the block is compatible with the anti-unifier of all those types.

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:

$$\begin{aligned}\mu = \{\ & \ell_1 \mapsto \langle(\alpha * (\alpha \to \beta)) \to \texttt{Unit} \to \beta, \ell_2\rangle \\ & \ell_2 \mapsto \texttt{Blk}(\ell_3,\ell_4) \\ & \ell_3 \mapsto \texttt{Blk}(\ell_4,0) \\ & \ell_4 \mapsto \langle\texttt{Unit} \to \texttt{T} \to \texttt{Int}, 0\rangle \qquad\qquad \}\end{aligned}$$

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

$$\begin{aligned}\Phi_1 &= \{\ \ell_1 \mapsto \{\ \texttt{Unit} \to \texttt{Int}\ \}\ \} \\ \Phi_2 &= \Phi_1 \cup \{\ \ell_2 \mapsto \{\ (\dot\alpha * (\dot\alpha \to \texttt{Int}))\ \}\ \} \\ \Phi_3 &= \Phi_2 \cup \{\ \ell_3 \mapsto \{\ \dot\alpha\ \}\ \} \\ \Phi_4 &= \Phi_3 \cup \{\ \ell_4 \mapsto \{\ \dot\beta\,;\ \dot\alpha \to \texttt{Int}\ \}\ \} \\ \Phi'_4 &= \Phi_1 \cup \{\ \ell_2 \mapsto \{\ (\texttt{T} * (\texttt{T} \to \texttt{Int}))\ \}\ ; \\ &\qquad\quad \ell_3 \mapsto \{\ \texttt{T}\ \}\ ; \\ &\qquad\quad \ell_4 \mapsto \{\ \texttt{T} \to \texttt{Int}; \texttt{T} \to \texttt{Int}\ \}\ \}\end{aligned}$$

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

$$\begin{aligned}\mathit{Imm}(\texttt{T}) &= \{0\} \\ \mathit{Arg}(\texttt{T}) &= [(\texttt{T} \to \texttt{Unit})\,;\,\texttt{Unit}]\end{aligned}$$

A possible rewriting sequence for checking the compatibility of the value (ℓ_1, μ) against the type $\texttt{Unit} \to \texttt{Int}$ begins by checking the closure $\mu(\ell_1) = \langle(\alpha * (\alpha \to \beta)) \to \texttt{Unit} \to \beta, \ell_2\rangle$:

$$\begin{aligned}& (\mu, \emptyset).\,(\ell_1 : \{\texttt{Unit} \to \texttt{Int}\}) \\ \text{[R-CLOS]} \gg\ & \exists\dot\alpha\dot\beta.\ \begin{cases}\texttt{Unit} \to \texttt{Int} = \texttt{Unit} \to \dot\beta \\ (\mu, \Phi_1).\,(\ell_2 : \{(\dot\alpha * (\dot\alpha \to \dot\beta))\})\end{cases} \\ \text{[R-UNIF]} \gg\ & \exists\dot\alpha.\,(\mu, \Phi_1).\,(\ell_2 : \{(\dot\alpha * (\dot\alpha \to \texttt{Int}))\})\end{aligned}$$

Continue by checking the block $\mu(\ell_2) = \texttt{Blk}(\ell_3,\ell_4)$.

$$\text{[R-BLK]} \gg\ \exists\dot\alpha\dot\beta\dot\gamma.\ \begin{cases}\Diamond\{(\dot\alpha * (\dot\alpha \to \texttt{Int}))\} \\ \texttt{arg}(\dot\alpha * (\dot\alpha \to \texttt{Int})) = [\dot\beta\,;\,\dot\gamma] \\ (\mu, \Phi_2).\,(\ell_3 : \{\dot\beta\} \wedge \ell_4 : \{\dot\gamma\})\end{cases}$$

$$\begin{aligned}\text{[R-}\Diamond\text{]} \gg\ & \exists\dot\alpha\dot\beta\dot\gamma.\ \begin{cases}\texttt{arg}(\dot\alpha * (\dot\alpha \to \texttt{Int})) = [\dot\beta\,;\,\dot\gamma] \\ (\mu, \Phi_2).\,(\ell_3 : \{\dot\beta\} \wedge \ell_4 : \{\dot\gamma\})\end{cases} \\ \text{[R-ARG]} \gg\ & \exists\dot\alpha\dot\beta\dot\gamma.\ \begin{cases}\dot\beta = \dot\alpha \wedge \dot\gamma = \dot\alpha \to \texttt{Int} \\ (\mu, \Phi_2).\,(\ell_3 : \{\dot\beta\} \wedge \ell_4 : \{\dot\gamma\})\end{cases} \\ \text{[R-UNIF]} \gg\ & \exists\dot\alpha.\,(\mu, \Phi_2).\,(\ell_3 : \{\dot\alpha\} \wedge \ell_4 : \{\dot\alpha \to \texttt{Int}\})\end{aligned}$$

Start the verification of block $\mu(\ell_3) = \texttt{Blk}(\ell_4,0)$.

$$\begin{aligned}\text{[R-BLK]} \gg\ & \exists\dot\alpha\dot\beta\dot\gamma.\ \begin{cases}\Diamond\{\dot\alpha\} \wedge \texttt{arg}(\dot\alpha) = [\dot\beta\,;\,\dot\gamma] \\ (\mu, \Phi_3).\ \begin{cases}\ell_4 : \{\dot\beta\} \\ \ell_4 : \{\dot\alpha \to \texttt{Int}\} \\ 0 : \{\dot\gamma\}\end{cases}\end{cases} \\ \text{[R-MERGE]} \gg\ & \exists\dot\alpha\dot\beta\dot\gamma.\ \begin{cases}\Diamond\{\dot\alpha\} \wedge \texttt{arg}(\dot\alpha) = [\dot\beta\,;\,\dot\gamma] \\ (\mu, \Phi_3).\ \begin{cases}\ell_4 : \{\dot\beta; \dot\alpha \to \texttt{Int}\} \\ 0 : \{\dot\gamma\}\end{cases}\end{cases}\end{aligned}$$

Postpone the verification of constraints $\Diamond$ and $\texttt{arg}(\_)$ and continue by checking the closure $\mu(\ell_4) = \langle\texttt{Unit} \to \texttt{T} \to \texttt{Int}, 0\rangle$.

$$\begin{aligned}\text{[R-CLOS]} \gg\ & \exists\dot\alpha\dot\beta\dot\gamma.\ \begin{cases}\Diamond\{\dot\alpha\} \wedge \texttt{arg}(\dot\alpha) = [\dot\beta\,;\,\dot\gamma] \\ \dot\beta = \texttt{T} \to \texttt{Int} \\ \dot\alpha \to \texttt{Int} = \texttt{T} \to \texttt{Int} \\ (\mu, \Phi_4).\ \begin{cases}0 : \{\texttt{Unit}; \texttt{Unit}\} \\ 0 : \{\dot\gamma\}\end{cases}\end{cases} \\ \text{[R-UNIF]} \gg\ & \exists\dot\gamma.\ \begin{cases}\Diamond\{\texttt{T}\} \wedge \texttt{arg}(\texttt{T}) = [\texttt{T} \to \texttt{Int}\,;\,\dot\gamma] \\ (\mu, \Phi'_4).\ \begin{cases}0 : \{\texttt{Unit}; \texttt{Unit}\} \\ 0 : \{\dot\gamma\}\end{cases}\end{cases}\end{aligned}$$

Conclude by checking the postponed constraints.

$$\begin{aligned}\text{[R-}\Diamond\text{]} &\gg \exists\dot\gamma.\,(\dots) \\ \text{[R-ARG]} &\gg \exists\dot\gamma.\,(\dot\gamma = \texttt{Unit} \wedge \dots) \\ \text{[R-UNIF]} &\gg (\mu, \Phi_4).\,(0 : \{\texttt{Unit}; \texttt{Unit}\} \wedge 0 : \{\texttt{Unit}\}) \\ \text{[R-HEAP]} &\gg 0 : \{\texttt{Unit}; \texttt{Unit}\} \wedge 0 : \{\texttt{Unit}\} \\ \text{[R-AUNIF1]} &\gg 0 : \{\texttt{Unit}\} \wedge 0 : \{\texttt{Unit}\} \\ \text{[R-INT]} &\gg 0 : \{\texttt{Unit}\} \\ \text{[R-INT]} &\gg \texttt{True}\end{aligned}$$

Figure 6. An example of rewriting sequence

To ensure the compatibility of a block with the anti-unifier of the expected types, it is required to check that all the expected types share the same head type constructor (lemma 4.3). For that purpose, the rule [R-BLK] introduces an homogeneity constraint $\Diamond\{\tau_1\,;\,\dots\,;\,\tau_n\}$. This constraint is resolved by the rewriting rule [R-$\Diamond$] where T is a type constructor distinct from Ref.

$$\begin{aligned}&\Diamond\{\texttt{T}(\tau_1, \dots, \tau_n)\,;\,\tau'_{1..m}\} \gg \qquad\qquad \text{[R-}\Diamond\text{]} \\ &\quad \exists\dot\alpha_{(1,1)..(n,m)}.\ \textstyle\bigwedge_{i=1..m}(\tau'_i = \texttt{T}(\dot\alpha_{(1,i)}, \dots, \dot\alpha_{(n,i)})) \\ &\qquad \textit{if } \dot\alpha_{(1,1)..(n,m)} \,\#\, \mathit{fv}(\{\tau'_{1..m}\}) \textit{ and } \texttt{T} \not\equiv \texttt{Ref}\end{aligned}$$

The homogeneity constraint is also used to ensure that the expected type for a mutable field does not contain any universal variable. The rule [R-$\Diamond$-REF] could not be applied if the type τ contains existential variables.

$$\begin{aligned}&\Diamond\{\texttt{Ref}(\tau)\} \gg \texttt{True} && \textit{if } \mathit{fv}(\tau) = \emptyset && \text{[R-}\Diamond\text{-REF]} \\ &\Diamond\{\texttt{Ref}(\tau)\} \gg \texttt{False} && \textit{if } \mathit{fuv}(\tau) \neq \emptyset && \text{[R-}\Diamond\text{-REF-FAIL]} \\ &\Diamond\{\texttt{Ref}(\tau)\,;\,\tau_{1..n}\} \gg && && \text{[R-}\Diamond\text{-REF-SET]} \\ &\qquad \Diamond\{\texttt{Ref}(\tau)\} \wedge \textstyle\bigwedge_{i=1..n}(\tau_i = \texttt{Ref}(\tau))\end{aligned}$$

Closures The rewriting rule [R-CLOS], described in figure 5, allows the verification of a closure $\langle\tau^{env} \to \tau' \to \tau'', w\rangle$. To test the existence for every expected type τ_i of a type τ_i^{env} such that $\tau_i^{env} \to \tau_i$ is an instance of the closure static type $\tau^{env} \to \tau' \to \tau''$, the rewriting rule introduces a unification constraint between τ_i and an instance of $\tau' \to \tau''$ where every type variable has been replaced by an existential type variable. This instance is written $\dot\tau'_i \to \dot\tau''_i$; the corresponding instance of τ^{env} is written $\dot\tau_i^{env}$ and the set of existential variable of $\dot\tau_i^{env} \to \dot\tau'_i \to \dot\tau''_i$ is written $\bar{\dot\alpha}_i$. The unification constraints are resolved using the two following rules.

$$\begin{aligned}&\exists\bar{\dot\alpha}.\,\tau_1 = \tau_2 \wedge C \gg \dot\theta(C) && \text{[R-UNIF]} \\ &\quad \textit{if } \dot\theta \textit{ is the m.g.u of } \tau_1 \textit{ and } \tau_2\textit{, and if } \mathit{dom}(\dot\theta) = \bar{\dot\alpha} \\ &\exists\bar{\dot\alpha}.\,\tau_1 = \tau_2 \wedge C \gg \texttt{False} && \text{[R-NUNIF]} \\ &\qquad\qquad \textit{if } \tau_1 \textit{ and } \tau_2 \textit{ do not unify}\end{aligned}$$

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.

Administrative rules 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).

$$\begin{aligned}&C \wedge \texttt{False} \gg \texttt{False} && && \text{[R-FALSE]} \\ &C \wedge \texttt{True} \gg C && && \text{[R-TRUE]} \\ &(\mu, \Phi).\,C \gg C && \textit{if } \mathit{fp}(C) \,\#\, \mathit{dom}(\mu) && \text{[R-HEAP]}\end{aligned}$$

Example 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* $(\mu, \emptyset).\,(\ell : \{\tau\}) \gg \texttt{True}$ *then there exists a typing environment* Ψ *such that* $\mu : \Psi$ *and* $\Psi(\ell) \preccurlyeq \tau$.

Theorem 4.2 (Semi-completeness). *If* $(\mu, \emptyset).\,(\ell : \{\tau\}) \gg \texttt{False}$ *then there exists no typing environment* Ψ *such that* $\mu : \Psi$ *and* $\Psi(\ell) \preccurlyeq \tau$.

To prove those 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 $\dot\theta$ for the free existential variables of the constraint and a typing environment Ψ for the free pointers. We write $\dot\theta, \Psi \models C$ when a constraint C is satisfied by $\dot\theta$ and Ψ.

The constraint `True` is always satisfied and a conjunction $C_1 \wedge C_2$ is satisfied whenever both C_1 and C_2 are satisfied. The constraint `False` can not be satisfied.

$$\text{[C-True]}\quad \dot\theta, \Psi \models \texttt{True} \qquad\qquad \text{[C-And]}\quad \frac{\dot\theta, \Psi \models C_1 \qquad \dot\theta, \Psi \models C_2}{\dot\theta, \Psi \models C_1 \wedge C_2}$$

An existential constraint $\exists\dot\alpha.\,C$ is satisfied by $\dot\theta$ and Ψ if and only if there exists a type τ such that C is satisfied by $\dot\theta$ extended with a substitution $\dot\alpha \mapsto \tau$ and Ψ.

$$\text{[C-Exists]}\quad \frac{\dot\theta \oplus \{\dot\alpha \mapsto \tau\}, \Psi \models C}{\dot\theta, \Psi \models \exists\dot\alpha.\,C}$$

A type constraint $w : \{\tau_{1..n}\}$ is satisfied by $\dot\theta$ and Ψ if and only if under the environment Ψ the word w is compatible with the anti-unifier of the instantiations of $\tau_{1..n}$ by $\dot\theta$.

$$\text{[C-Question]}\quad \frac{\Psi \vdash w : \curlywedge\{\dot\theta(\tau_{1..n})\}}{\dot\theta, \Psi \models w : \{\tau_{1..n}\}}$$

An equality constraint $\tau_1 = \tau_2$ is satisfied by $\dot\theta$ if and only if after instantiation by $\dot\theta$ the types are syntactically equal. In the same way, an ADT decomposition constraint $\texttt{arg}(\tau) = [\tau_1\,;\,\tau_2]$ is satisfied by $\dot\theta$ if and only if after instantiation by θ it corresponds to a case of the function $Arg(\_)$; and a head type constructor constraint $\Diamond\{\tau_{1..n}\}$ is satisfied by a substitution $\dot\theta$ if and only if after instantiation by θ all the type share the same head type constructor.

$$\text{[C-Equal]}\quad \frac{\dot\theta(\tau_1) = \dot\theta(\tau_2)}{\dot\theta, \Psi \models \tau_1 = \tau_2} \qquad \text{[C-Arg]}\quad \frac{Arg(\dot\theta(\tau)) = [\dot\theta(\tau')\,;\,\dot\theta(\tau'')]}{\dot\theta, \Psi \models \texttt{arg}(\tau) = [\tau'\,;\,\tau'']}$$

$$\text{[C-}\Diamond\text{]}\quad \frac{\forall i = 1..n,\ \ \dot\theta(\tau_i) = \texttt{T}_1(\tau_i^1, \ldots, \tau_i^m)}{\dot\theta, \Psi \models \Diamond\{\tau_{1..n}\}}$$

$$\text{[C-}\Diamond\text{-Ref]}\quad \frac{\forall i = 1..n,\ \ \dot\theta(\tau_i) = \texttt{Ref}(\tau_i') \qquad fv(\tau_i') = \emptyset}{\dot\theta, \Psi \models \Diamond\{\tau_{1..n}\}}$$

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

$$\Psi'(\ell) = \begin{cases} \Psi''(\ell) & \text{if } \ell \notin dom(\Phi') \\ (\curlywedge\{\dot\theta(\tau) \mid \forall \tau \in \Phi'(\ell)\}) \curlywedge \Psi''(\ell) & \text{otherwise} \end{cases}$$

$$\text{[C-Heap]}\quad \frac{dom(\Psi'') = dom(\mu) \qquad \Psi' = \curlywedge\{\dot\theta(\Phi')\} \curlywedge \Psi'' \qquad \Psi \oplus \Psi' \vdash \mu : \Psi'' \qquad \dot\theta, \Psi \oplus \Psi' \models C}{\dot\theta, \Psi \models (\mu, \Phi').\,C}$$

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

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

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. $\emptyset, \emptyset \models (\mu, \emptyset).\,(\ell : \{\tau\})$ *if and only if there exists a typing environment* Ψ *such that* $\mu : \Psi$ *and* $\Psi(\ell) \preccurlyeq \tau$.

Lemma 4.2. *If* $C_1 \gg C_2$ *and* $\dot\theta, \Psi \models C_2$ *then* $\dot\theta, \Psi \models C_1$.

To prove the soundness of the rule [R-Blk], we have to check that the function $Arg(\tau)$ 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* τ_i' *and* τ_i'' *such that* $Arg(\tau_i) = [\tau_i'\,;\,\tau_i'']$, *then* $Arg(\curlywedge\{\tau_{1..n}\}) = [\curlywedge\{\tau'_{1..n}\}\,;\,\curlywedge\{\tau''_{1..n}\}]$.

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-Blk] and [R-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-Remove] and some constraint may remain stuck in a non-trivial form. For example, consider the following constraint:

$$C_{stuck} = (\mu, \{\ell \mapsto \{\texttt{Int} \to \texttt{Int}\}\}).(\ell : \{\dot\alpha \to \dot\alpha\})$$

We can not decide, without instantiating the variable $\dot\alpha$, whether the expected type $\dot\alpha \to \dot\alpha$ is an instance or not of the memorized types $\texttt{Int} \to \texttt{Int}$. We choose not to add this extra condition, allowing the introduction in section 4.4 of rewriting strategies that "unstuck" this constraint by applying the rule [R-Clos].

To prove the semi-completeness without adding the extra condition to the rules [R-Blk] and [R-Clos], we introduce a second definition of constraint satisfiability, called strict satisfiability and written $\dot\theta, \Psi \models^{s} C$. 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-Heap] by the following rule:

$$\text{[C'-Heap]}\quad \frac{dom(\Psi'') = dom(\mu) \qquad \Psi' = \curlywedge\{\dot\theta(\Phi')\} \curlywedge \Psi'' \qquad \Psi \oplus \Psi' \vdash \mu : \Psi' \qquad \dot\theta, \Psi \oplus \Psi' \models^{s} C}{\dot\theta, \Psi \models^{s} (\mu, \Phi').\,C}$$

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. *$\emptyset, \emptyset \models^{s} (\mu, \emptyset).(\ell : \{\tau\})$ if and only if there exists a typing environment Ψ such that $\mu : \Psi$ and $\Psi(\ell) \preccurlyeq \tau$.*

Lemma 4.5. *If $C_1 \gg C_2$ and $\dot{\theta}, \Psi \models^{s} C_1$ then $\dot{\theta}, \Psi \models^{s} C_2$.*

The second notion of satisfiability does not allow the soundness proof of the rule [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 [R-BLK] or [R-CLOS] whenever the rule [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[4]:

$$\mu = \{\ell_0 \mapsto \langle \sigma_{apply}, \{f : \ell_0 \,;\, x : 0\}\rangle\}$$

Checking the compatibility of μ against the type $\texttt{Unit} \to \texttt{Int}$ will lead to following infinite sequence:

$$\begin{array}{lll} & (\mu, \emptyset).(\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}) & \\ \text{[R-CLOS]}\ \text{[R-UNIF]} & \gg & \exists\dot{\alpha}.(\mu, \Phi_1).\left\{\begin{array}{l} \ell_0 : \{\dot{\alpha} \to \texttt{Int}\} \\ 0 : \{\dot{\alpha}\} \end{array}\right. \\ \text{[R-CLOS]}\ \text{[R-UNIF]} & \gg & \exists\dot{\alpha}\dot{\beta}\dot{\gamma}.(\mu, \Phi_2).\left\{\begin{array}{l} \ell_0 : \{\dot{\beta} \to \texttt{Int}; \dot{\gamma} \to \texttt{Int}\} \\ 0 : \{\dot{\alpha}\} \wedge 0 : \{\dot{\beta}\,;\,\dot{\gamma}\} \end{array}\right. \\ \text{[R-CLOS]} & \gg & \ldots \end{array}$$

where:

$$\begin{array}{rcl} \Phi_1 & = & \{\ell_0 \mapsto \{\texttt{Unit} \to \texttt{Int}\}\} \\ \Phi_2 & = & \{\ell_0 \mapsto \{\texttt{Unit} \to \texttt{Int}\,;\,\dot{\alpha} \to \texttt{Int}\}\} \end{array}$$

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

If we can not 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 μ that contains no cycle, a constraint $(\mu, \emptyset).(\ell : \{\tau\})$ is rewritten in $\texttt{True}$ or $\texttt{False}$ in a finite number of steps.*

Theorem 4.4. *In a heap μ that contains no polymorphic closure that may introduce existential types, a constraint $(\mu, \emptyset).(\ell : \{\tau\})$ is rewritten in $\texttt{True}$ or $\texttt{False}$ 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 memorized type, the number of type constructors in the anti-unifiers of memorized type's sets, the number of type constraints, the number of types in the sets of memorized 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 presence of 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.

$$(\mu, \Phi).C \gg (\mu_1, \Phi_1).(\mu_2, \Phi_2).C \quad \begin{array}{r} \text{if } \mu_1 \uplus \mu_2 = \mu \\ \text{and } fp(img(\mu_1)) \# dom(\mu_2) \\ \text{and } \Phi_i = \Phi_{|dom(\mu_i)} \end{array} \quad \text{[R-SORT]}$$

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 memorized 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 rule, that allows the unification of the expected type with the memorized one:

$$(\mu, \Phi').(\ell : \{\tau'\} \wedge C) \gg \tau = \tau' \wedge (\mu, \Phi').C \quad \text{if } \Phi'(\ell) = \{\tau\} \quad \text{[R-FORCEUNIF]}$$

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 memorized types could not be simplified by anti-unification, the following rule allows the instantiation of an existential variables with the universal type.

$$\exists\dot{\alpha}.C \gg \exists\dot{\alpha}.(C \wedge \dot{\alpha} = \star) \quad \text{[R-FORCEINST]}$$

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 memorized 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 memorized 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

[4] This heap may represent *A.loop* in the OCaml program:

```
module rec A : sig
  val loop : Unit → α
end = struct
  let delay f x = fun () → f x
  let loop = delay A.loop ()
end
```

[5] Remember that the anti-unifier may not always be computed in the presence of existential type variables.

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:

`val` *marshall* : $\alpha \to$ `String`
`val` *unmarshall* : `String` $\to \alpha$

The type-safe unmarshalling function has the following prototype:

`val` *safe_unmarshall* : $\texttt{Ty}(\alpha) \to \texttt{String} \to \alpha$

where $\texttt{Ty}(\tau)$ is the classical singleton type whose unique value is a runtime representation of the type τ [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 implented 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 un-trusted 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 *ad hoc* functions from data type definitions. As other mechanisms based on *ad hoc* functions we know of, it allows the communication with un-trusted 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 HashCaml [5] compiler or the Quicksilver library [21] do. The HashCaml 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 definition and of the source code of the associated function. Quicksilver is another Camlp4 based marshalling library that generates *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

[1] M. Abadi, L. Cardelli, B. C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1): 111–130, Jan. 1995.

[2] S. Aditya and A. Caro. Compiler-directed type reconstruction for polymorphic languages. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 74–82, June 1993.

[3] S. Aditya, C. H. Flood, and J. E. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 12–23, June 1994.

[4] A. W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(2):153–162, July 1989.

[5] J. Billings, P. Sewell, M. R. Shinwell, and R. Strnisa. Type-safe distributed programming for OCaml. In *ML'06: Proceedings of the ACM Workshop on ML*, pages 20–31, Sept. 2006.

[6] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP'98: Proceedings of the 3rd ACM International Conference on Functional Programming*, volume 34(1) of *SIGPLAN Not.*, pages 301–312, Sept. 1998.

[7] M. Elsman. Type-specialized serialization with sharing. In *TFP'05: Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming*, pages 47–62, Sept. 2005.

[8] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP'02: Proceedings of the 7th ACM International Conference on Functional Programming*, volume 37(9) of *SIGPLAN Not.*, pages 48–59, Sept. 2002.

[9] J. Furuse and P. Weis. Input/output of caml values (in french). In *JFLA'00: Journées Francophones des Langages Applicatifs*. INRIA, Jan. 2000.

[10] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. *SIGPLAN Lisp Pointers*, V(1):53–65, 1992.

[11] G. Henry. *Typing unmarshalling without marshalling types (in french)*. PhD thesis, Univ Paris Diderot, 2011. URL `http://tel.archives-ouvertes.fr/tel-00624156/PDF/these.pdf`.

[12] M. Hicks, S. Weirich, and K. Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer-Verlag, 2000.

[13] G. Huet. *Résolution d'équations dans des langages d'ordre 1,2,..., ω*. PhD thesis, Université Paris 7, 1976.

[14] A. J. Kennedy. Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[15] X. Leroy. Efficient data representation in polymorphic languages. In *PLILP'90: 2nd International Workshop on Programming Language Implementation and Logic Programming*, volume 456 of *Lecture Notes in Computer Science*, pages 255–276. Springer, Aug. 1990.

[16] X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[17] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.12*, June 2011.

[18] A. Mycroft. Polymorphic type schemes and recursive definitions. In *ISP'84: Proceedings of the 6th International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984.

[19] G. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[20] F. Pottier and D. Rémy. The Essence of ML Type Inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[21] H. Sutou and E. Sumii. Quicksilver/OCaml: A poor man's type-safe and abstraction-secure communication library. Unpublished, 2007.

[22] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL'08: Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, pages 395–406, Jan. 2008.

[23] A. P. Tolmach. Tag-free garbage collection using explicit type parameters. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 1–11, June 1994.

[24] S. Weirich. Higher-order intensional type analysis. In *ESOP'02: Proceedings of the 11th European Symposium on Programming*, pages 98–114, 2002.

[25] S. Weirich. Type-safe run-time polytypic programming. *Journal of Functional Programming*, 16(10):681–710, Nov. 2006.

[26] J. Yallop. Practical generic programming in OCaml. In *ML'07: Proceedings of the ACM Workshop on ML*, pages 83–94, Sept. 2007.