Numbering Matters: First-Order Canonical Forms for Second-Order Recursive Types

Nadji Gauthier
INRIA
Nadji.Gauthier@inria.fr

François Pottier
INRIA
Francois.Pottier@inria.fr

Abstract

We study a type system equipped with universal types and equirecursive types, which we refer to as F_μ. We show that type equality may be decided in time $O(n \log n)$, an improvement over the previous known bound of $O(n^2)$. In fact, we show that two more general problems, namely entailment of type equations and type unification, may be decided in time $O(n \log n)$, a new result. To achieve this bound, we associate, with every F_μ type, a *first-order canonical form*, which may be computed in time $O(n \log n)$. By exploiting this notion, we reduce all three problems to equality and unification of *first-order* recursive terms, for which efficient algorithms are known.

1 Introduction

During the last decade, the programming language community spent a great deal of effort studying object-oriented programming languages and devising *object encodings* [3, 7, 15, 12, 16]. A typical object encoding is a type-preserving translation of a surface object-oriented language into a typed λ-calculus. Such an encoding may serve two purposes. First, it explains object-oriented programming in terms of standard type-theoretic concepts. Second, it may be put to effective use as the front-end of a type-preserving compiler, whose back-end is then purely concerned with typed λ-calculus. This requires, however, the target language of the encoding to have *decidable* typechecking and, if possible, to admit an *efficient* typechecking procedure.

Because object orientation is complex, the target languages of most object encodings are rich λ-calculi. They typically incorporate some or all of the following features: first-class *universal and existential types*; *recursive types*; type operators; subtyping and bounded quantification. In the present paper, we focus on the combination of the first two: the object of our study is F_μ, an extension of Girard and Reynolds' system F with recursive types. The question we are interested in is, *does F_μ have decidable and efficient typechecking?*

Before addressing such a question, we must state it more precisely, because F_μ comes in two flavors, whose typechecking problems are quite different: one extends F with *isorecursive* types, while the other extends it with *equirecursive* types [1, 10].

In an extension of F with isorecursive types, two new typing rules are added to the type system, which direct the typechecker to fold or unfold a recursive type. (These rules usually allow folding and unfolding only at the root of the type. Allowing them to take place under a context requires either adding *coercions*—special constructs that generate no code—to the programming language, as proposed by Abadi and Fiore [1], or defining more complex typing rules for folding and unfolding, perhaps along the lines suggested by Collins and Shao [5].) The definition of type equality is the same as in F: that is, no new axioms are added to deal with recursive types. Thus, typechecking isorecursive F_μ is no more difficult than typechecking F.

In an extension of F with equirecursive types, on the other hand, there are no new typing rules. Instead, type equality is extended so that comparing two types amounts to comparing their infinite unfoldings. Thus, typing derivations are less verbose. Folding and unfolding naturally take place not only at the root of a type, but also under a context. However, it is now more difficult to determine whether two types are equal.

Thus, a more precise statement of the question is: does *equirecursive* F_μ have decidable and efficient typechecking? Perhaps surprisingly, the problem has received little attention in the literature. As suggested above, the key issue is to decide whether two types are equal. It appears to have been only recently studied. Colazzo and Ghelli [4] show that the more general problem of deciding subtyping in Kernel Fun is decidable. Glew [13] studies type equality in F_μ and proves that its complexity is bounded by $O(n^2)$, where n is the size of the types at hand. In the present paper, we improve upon Glew's result by giving a decision algorithm whose complexity is $O(n \log n)$.

We are in fact able to settle a more general question: does an extension of equirecursive F_μ with *guarded algebraic data types*, in the style of Xi *et al.* [22], have decidable typechecking? Such a type system is not of purely theoretical interest: for instance, it could be a component of a type-preserving compiler whose front-end implements a typical object encoding, requiring universal types and recursive types, and whose back-end performs defunctionalization in the style of [18], requiring guarded algebraic data types. The key issue is then to decide whether two F_μ types are equal *under a number of equality hypotheses*, that is, to decide whether a conjunction of type equations *entails* another type equation. To the best of our knowledge, this issue has never been studied before. In the present paper, we show that it can be decided in time $O(n \log n)$, where n is the size of the input problem.

Our solution to the entailment problem is via a reduction to the *unification* problem. That is, we are able to determine whether two F_μ types are *unifiable* in time $O(n \log n)$. This

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \textit{(variable)} \\
& | & \mathrm{a} & \textit{(atom)} \\
& | & \mu\alpha.T\,\vec{\tau} & \textit{(type constructor application)} \\
& | & \mu\alpha.\forall \mathrm{a}.\tau & \textit{(universal type)}
\end{array}
$$

Figure 1: Types in F_μ

$$
\frac{}{\alpha =_{\mu a} \alpha} \qquad \frac{}{\mathrm{a} =_{\mu a} \mathrm{a}}
$$

$$
\frac{\{\alpha \mapsto \mu\alpha.T\,\vec{\tau}\}\vec{\tau} =_{\mu a} \{\alpha' \mapsto \mu\alpha'.T\,\vec{\tau}'\}\vec{\tau}'}{\mu\alpha.T\,\vec{\tau} =_{\mu a} \mu\alpha'.T\,\vec{\tau}'}
$$

$$
\frac{\{\alpha \mapsto \mu\alpha.\forall \mathrm{a}.\tau\}\tau =_{\mu a} \{\alpha' \mapsto \mu\alpha'.\forall \mathrm{a}.\tau'\}\tau'}{\mu\alpha.\forall \mathrm{a}.\tau =_{\mu a} \mu\alpha'.\forall \mathrm{a}.\tau'}
$$

Figure 2: Type equality in F_μ

result could have implications in the area of (partial) type inference for F_μ. It may also be used to implement *hash-consing* of second-order recursive types, a technique that so far has been studied for first-order recursive types only [6].

In fact, our algorithm for unifying F_μ types has already found an initially unexpected application. We discovered, during a conversation with Jacques Garrigue, that OCaml's type inferencer requires such an algorithm, because object types may be recursive and may contain polymorphic methods. Upon close examination, the unification algorithm that has been employed by the OCaml compiler for several years was found to be unsound. It should soon be replaced with a version of the one described in this paper [Garrigue, personal communication].

2 Types and type equality in F_μ

In this section, we define the problem and highlight some of its subtleties. We explain how the decision problems for type equality in F and F_μ have been dealt with in the literature, and give an outline of our solution.

2.1 Definition

The syntax of types in our version of F_μ appears in Figure 1. For the sake of clarity, we distinguish *variables* $\alpha, \beta, \gamma, \ldots$, which are bound by μ, and *atoms* $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \ldots$, which are bound by $\forall$. Variables and atoms are drawn from two disjoint, denumerable sets. The *free variables* $\mathrm{fv}(\tau)$ and the *free atoms* $\mathrm{fa}(\tau)$ of a type τ are defined in the usual way. We identify types modulo α-equivalence of variables and atoms. A type is *atom-closed* if and only if it has no free atoms.

We let T range over an arbitrary set of *type constructors*, each of which is equipped with a nonnegative integer arity. In the notation $T\,\vec{\tau}$, the length of the vector of types $\vec{\tau}$ is implicitly assumed to match the arity of T. In several examples, we employ the type constructor $\rightarrow$, of arity 2, whose applications are written infix.

Following common practice, we combine the μ quantifier (which forms recursive types) with type constructor applications and with the $\forall$ quantifier. By not making $\tau ::= \mu\alpha.\tau$ a production of the grammar, we disallow meaningless types such as $\mu\alpha.\alpha$. For the sake of readability, we write $T\,\vec{\tau}$ for $\mu\alpha.T\,\vec{\tau}$ when α does not appear free in $\vec{\tau}$, and $\forall \mathrm{a}.\tau$ for $\mu\alpha.\forall \mathrm{a}.\tau$ when α does not appear free in τ.

In standard presentations of F_μ, the distinction between variables and atoms is not made. As a result, a standard F_μ type must undergo a simple translation step in order to fit our formalism. The translation is straightforward: universally bound type variables become atoms, while μ-bound and free type variables remain variables. For instance, the standard F_μ type $\mu\alpha.\beta \rightarrow \forall\beta.\alpha \rightarrow \beta \rightarrow \gamma$ is written $\mu\alpha.\beta \rightarrow \forall \mathrm{b}.\alpha \rightarrow \mathrm{b} \rightarrow \gamma$ in this paper.

It is important to remark that the image of a standard F_μ type under this translation is atom-closed by construction. For this reason, the input of the decision problems studied in this paper, such as equality and unifiability, is restricted to consist of *atom-closed* types. Also, two types are considered unifiable if and only if they admit an *atom-closed* unifier. Note, however, that the subterms of an atom-closed type are not in general atom-closed.

It is also worth noting that, under this translation, the images of the standard types τ and $\forall\alpha.\tau$ may differ at arbitrarily deep locations. For instance, the image of $\alpha \rightarrow \alpha$ is $\alpha \rightarrow \alpha$, while the image of $\forall\alpha.\alpha \rightarrow \alpha$ is $\forall \mathrm{a}.\mathrm{a} \rightarrow \mathrm{a}$. Thus, in standard F or F_μ, constructing $\forall\alpha.\tau$ given τ requires constant time, whereas, in our formalism, constructing a representation of $\forall\alpha.\tau$ given a representation of τ is *not* a constant time operation.

A *substitution* is a total mapping of variables to types. The *domain* of θ is the set of variables α where α and $\theta(\alpha)$ differ. We write $\{\alpha \mapsto \tau\}$ for the substitution that maps α to τ and is the identity elsewhere. A substitution may be viewed as a total mapping of *types* to types, in the usual, *capture-avoiding*, manner.

Types are finite terms with binders. As a result, mathematical equality of types, which we write $=$, incorporates α-equivalence of variables and atoms, but does not treat μ binders in a special way. In order to obtain an *equirecursive* flavor of F_μ, one must define a more permissive notion of type equality, incorporating folding and unfolding of recursive types. This new equivalence relation, which we write $=_{\mu a}$, is coinductively defined by the rules in Figure 2.

The definition of $=_{\mu a}$ is entirely standard. (For background reading on recursive types and coinduction, we refer the reader to [1, 2, 10].) Relations are extended to vectors in a pointwise manner, so that $\vec{\tau} =_{\mu a} \vec{\tau}'$ means that, for every index i, the i-th components of the vectors $\vec{\tau}$ and $\vec{\tau}'$ are in the relation $=_{\mu a}$. The effect of the last rule is to unfold the outermost μ binders, exposing a pair of universal types, whose bodies are then compared. For the sake of simplicity, the rule requires the universal quantifiers on either side of the equality to share a common naming convention, that is, to bind the *same* atom a. Because types are identified modulo α-equivalence of atoms, this does not incur any loss of generality: it is possible to formulate an equivalent rule, where this requirement is removed, and where the premise incorporates an explicit renaming of atoms.

It is straightforward to establish the following facts: substitution preserves equality; equality preserves free atoms; substitution preserves or increases free atoms.

Lemma 2.1 *$\tau =_{\mu a} \tau'$ implies $\theta\tau =_{\mu a} \theta\tau'$.* ◇

Lemma 2.2 *$\tau =_{\mu a} \tau'$ implies* $\mathrm{fa}(\tau) = \mathrm{fa}(\tau')$. ◇

Lemma 2.3 $\mathrm{fa}(\tau) \subseteq \mathrm{fa}(\theta\tau)$. ◇

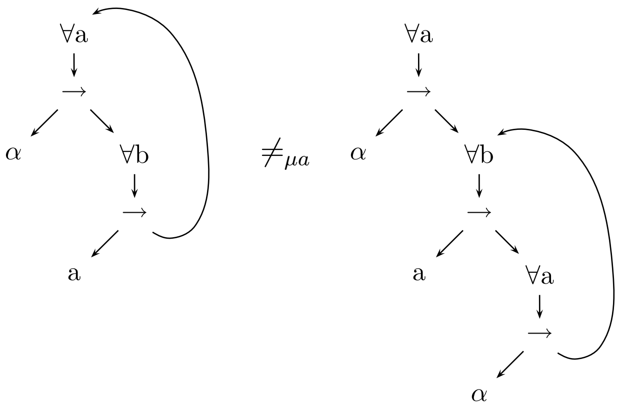

Figure 3: Subtleties of type equality

2.2 Some subtleties of type equality

Although the definition of $=_{\mu a}$ is simple, one must proceed with caution: it is easy to form misleading intuitions about it. Part of its subtlety is illustrated in Figure 3, which contains graphical representations of the types $\tau_1 = \mu\beta.\forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \beta$ and $\tau_2 = \forall \mathrm{a}.\alpha \to \mu\beta.\forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{a}.\alpha \to \beta$. (This example is adapted from [13].) These types are *not* in the relation $=_{\mu a}$, even though one might believe, at first sight, that their infinite unfoldings coincide.

Let us have a closer look. An unfolding of τ_2 is

$$\forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{c}.\alpha \to \mu\beta.\forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{a}.\alpha \to \beta.$$

Starting from the left, examine the third universal quantifier: is this what you expected? Here is what happened. Because the atom a appears free in the term $\mu\beta.\forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{a}.\alpha \to \beta$, and because β appears inside the scope of a $\forall$a quantifier in the term $\forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{a}.\alpha \to \beta$, computing a correct unfolding requires an α-conversion step, so as to avoid capture. Here, the innermost $\forall$a quantifier in τ_2 was changed into $\forall$c, which explains the result.

Computing an unfolding of τ_1 is more straightforward. Indeed, since τ_1 is atom-closed, there is no danger of capture. We find

$$\forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \mu\beta.\forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \beta,$$

which, by α-equivalence, may be written

$$\forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \mu\beta.\forall \mathrm{c}.\alpha \to \forall \mathrm{b}.\mathrm{c} \to \beta.$$

Let us now place the unfoldings of τ_1 and τ_2 next to each other:

$$\begin{array}{l} \forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \mu\beta.\forall \mathrm{c}.\alpha \to \forall \mathrm{b}.\mathrm{c} \to \beta \\ \forall \mathrm{a}.\alpha \to \forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{c}.\alpha \to \mu\beta.\forall \mathrm{b}.\mathrm{a} \to \forall \mathrm{a}.\alpha \to \beta \end{array}$$

It is now clear that these types are *not* in the relation $=_{\mu a}$. Indeed, starting from the left and until the fourth universal quantifier, these types offer a common structure. However, at that point, the former exhibits an occurrence of the atom c, whereas the latter exhibits an occurrence of a.

In short, the (incorrect) intuition that τ_1 and τ_2 are related by $=_{\mu a}$ stems from the mental use of a *capturing* substitution. By naïvely unrolling the loop in τ_2, we bring an occurrence of the atom a into the scope of the innermost $\forall$a quantifier, within which it initially did *not* lie: indeed, *the scope of a $\forall$ quantifier does not extend through a reverse edge*. This fact is obvious when examining syntactic representations of types—for instance, in $\mu\beta.\mathrm{a} \to \forall \mathrm{a}.\beta$, the scope of $\forall$a is β alone, and does not include the occurrence of a to its left, which is free—but is perhaps less so when thinking in terms of graphs.

2.3 Deciding type equality: the state of the art

The above example illustrates some of the difficulties that arise when comparing two types for equality. First, one must really compare the *infinite unfoldings* of the types at hand. Second, *renamings of atoms* are involved, for two reasons: (i) unfolding recursive types involves capture-avoiding substitutions, and (ii) comparing two universal types requires ensuring that the bound atoms match.

The decision problem for type equality has been investigated by Glew [13]. He encodes types as ad hoc automata, which may also be viewed as graphs somewhat analogous to those found in Figure 3, and gives an algorithm that decides type equality. Roughly speaking, Glew's algorithm checks for the existence of a bisimulation relating two automata. In terms of graphs, this process could be described as follows. The two graphs are traversed synchronously. When reaching two nodes labeled with universal quantifiers, say $\forall$a and $\forall$b, one keeps track of the correspondence between the atoms a and b, so that, when later reaching two leaf nodes labeled with the atoms a and b, they are (correctly) viewed as related. Glew uses *partial bijections* to keep track of this correspondence. Because both the number of partial bijections that may be constructed and the number of pairs of nodes that may be visited are finite, the algorithm terminates. However, the number of partial bijections is in fact exponential in n, where n is the size of the input problem. Fortunately, thanks to a more clever abrupt termination criterion, Glew is able to achieve time complexity $O(n^2)$.

It is worth recalling that, in system F (that is, in the absence of recursive types), types can be compared in time $O(n)$, provided they are represented using a De Bruijn encoding. The cost of converting a nameful representation into a De Bruijn encoding is $O(n \log n)$, assuming some flavor of balanced trees is used to map atoms to integer indices. (The expected cost can be brought down to $O(n)$ by using hash tables instead of balanced trees.) This approach is used in many typecheckers for F; see, for instance, [17, Chapter 25]. In the presence of equirecursive types, however, De Bruijn indices become more difficult to manipulate. For instance, successive unfoldings of a type may cause an ever-growing sequence of indices to appear, leading to an infinite, *irregular* first-order term: see [13, Section 3.1]. To the best of our knowledge, the practical use of a De Bruijn encoding in such a setting has never been investigated. Glew does consider infinite trees that contain De Bruijn indices, but only as a mathematical model, as opposed to an implementation scheme.

To sum up, the current state of the art is as follows: although type equality has worst-case time complexity $O(n \log n)$ in F, the best known algorithm for F_μ runs in time $O(n^2)$. Why such a gap? Should equirecursive types really be so expensive? In the following, we answer in the negative.

2.4 Our approach

The strength of the classic De Bruijn encoding lies in the fact that it provides *first-order canonical forms* of types: two F types are equal, up to α-equivalence of atoms, if and only if their De Bruijn encodings, which are first-order terms, are syntactically equal.

We propose to proceed in a similar manner: to every F_μ type, we associate a first-order recursive term, where atoms are replaced with suitable natural integers. The structure of the input type, including its μ binders, is preserved, so that the encoding's output may in fact be viewed as an *infinite*, but *regular*, first-order tree. The key trick is to choose the numbering of atoms in such a way that the encoding is *canonical*: we prove that two F_μ types are related by $=_{\mu a}$ if and only if their encodings, viewed as regular first-order trees, are equal. The manner in which we number atoms appears to be original, and is unrelated to De Bruijn's scheme.

We prove that, by using appropriate data structures, the time complexity of computing a type's encoding is $O(n \log n)$. Furthermore, a standard first-order unification algorithm such as Huet's [14] allows testing two recursive first-order terms for equality in time $O(n\alpha(n))$. There follows that type equality in F_μ has time complexity $O(n \log n)$.

The problem of determining whether two F_μ types are unifiable is addressed in the same manner: it is reduced, via the encoding, to unification of first-order recursive terms.

2.5 Related work

Colazzo and Ghelli [4] study the decision problem for the subtyping relationship in an extension of Kernel Fun with equirecursive types, and find it to be decidable. This implies that type equality in F_μ is decidable as well. The time complexity of their algorithm appears to be unknown.

Glew's work [13] was mentioned above. He studies type equality in F_μ and gives an algorithm whose time complexity is quadratic.

The problem of determining whether two F_μ types are unifiable may be turned, in a very simple manner, into a *nominal unification* problem [21], provided nominal unification is extended with support for recursive terms, which appears straightforward. However, neither we nor Urban [personal communication] are currently able to formulate a nominal unification algorithm whose time complexity is less than $O(n^2)$.

We solve the unification problem for F_μ types, which we refer to as *second-order recursive types* and which Glew refers to as *second-order trees*. Yet, the present paper has nothing to do with *second-order unification* [8]. Here, we are interested in unification modulo $=_{\mu a}$, that is, modulo α-equivalence of atoms and folding and unfolding of recursive types. Second-order (or higher-order) unification consists in unifying simply-typed λ-terms modulo $\beta\eta$-equivalence, and is undecidable.

3 A first-order encoding of F_μ types

In this section, we encode second-order recursive types (*types* for short) into a particular class of first-order recursive terms (*terms* for short).

$$
\begin{array}{rcll}
\sigma & := & \alpha & \textit{(variable)} \\
 & \mid & \mathrm{a} & \textit{(atom)} \\
 & \mid & \mu\alpha.(\mathrm{a})\, T\, \vec{\sigma} & \textit{(term constructor application)} \\
 & \mid & \mu\alpha.(\mathrm{a})\, \forall\, \sigma & \textit{(idem)}
\end{array}
$$

Figure 4: First-order recursive terms

$$
\frac{}{\alpha =_\mu \alpha} \qquad \frac{}{\mathrm{a} =_\mu \mathrm{a}}
$$

$$
\frac{\{\alpha \mapsto \mu\alpha.(\mathrm{a})\, T\, \vec{\sigma}\}\vec{\sigma} =_\mu \{\alpha' \mapsto \mu\alpha'.(\mathrm{a})\, T\, \vec{\sigma}'\}\vec{\sigma}'}{\mu\alpha.(\mathrm{a})\, T\, \vec{\sigma} =_\mu \mu\alpha'.(\mathrm{a})\, T\, \vec{\sigma}'}
$$

$$
\frac{\{\alpha \mapsto \mu\alpha.(\mathrm{a})\, \forall\, \sigma\}\sigma =_\mu \{\alpha' \mapsto \mu\alpha'.(\mathrm{a})\, \forall\, \sigma'\}\sigma'}{\mu\alpha.(\mathrm{a})\, \forall\, \sigma =_\mu \mu\alpha'.(\mathrm{a})\, \forall\, \sigma'}
$$

Figure 5: Equality of first-order recursive terms

3.1 First-order recursive terms

We first define the target space of the encoding, that is, the syntax of the first-order terms σ that we use to encode types. It appears in Figure 4. As before, terms include variables and atoms, and variables may be μ-bound at a constructor application node. The essential difference with respect to the syntax of types, which was given in Figure 1, lies in the treatment of atoms. Here, applications of the constructors T and $\forall$ are annotated with an atom (a), but do *not* bind it: that is, a occurs *free* in both $\mu\alpha.(\mathrm{a})\, T\, \vec{\sigma}$ and $\mu\alpha.(\mathrm{a})\, \forall\, \sigma$. As a result, atoms are *never* bound: all of the atoms that occur in a first-order term σ occur *free* in σ. The constructor $\forall$ no longer plays a special role: it is simply a unary term constructor.

We equip terms with a notion of equality, written $=_\mu$, whose coinductive definition appears in Figure 5. It is the standard notion of equality for first-order recursive terms: it only accounts for α-equivalence of variables and for folding and unfolding of μ binders. In other words, two terms are related by $=_\mu$ if and only if their infinite unfoldings, which are regular trees, coincide. In the third and fourth rules in Figure 5, the *same* atom (a) must appear on either side of the equality: since atoms are never bound, no implicit α-conversion step is allowed.

To complete the definition of terms, we must be more specific about the nature of atoms. Here is why. In the type $\forall$a.a, the atom a is bound: this type may also be written $\forall$b.b. However, at the level of terms, atoms are free, so they are observable: if a and b are distinct atoms, then the terms $(\mathrm{a})\,\forall\,\mathrm{a}$ and $(\mathrm{b})\,\forall\,\mathrm{b}$ are distinct. As a result, our encoding, whose purpose is to produce *canonical* forms, must be able to perform a deterministic choice between the two. If atoms were interchangeable for all purposes, as is usually the case, such a choice would be impossible [9, Remark 4.6]. Thus, we must impose some more structure on the set of atoms.

It is convenient to identify atoms with *natural integers*, so that atoms are totally ordered and have a successor function. From here on, we adopt this convention. At the level of types, this decision has no impact: because types are identified modulo α-equivalence of atoms, and because, at the end of the day, we are only interested in atom-closed

$$
\begin{array}{lcl}
N(\theta, \alpha) & = & \alpha \\
N(\theta, \mathrm{a}) & = & \mathrm{a} \\
N(\theta, \mu\alpha.T\,\vec{\tau}) & = & \mu\alpha.(\mathrm{a})\,T\,N(\theta \circ \{\alpha \mapsto \mathrm{a}\}, \vec{\tau}) \\
 & & \quad \text{if } a = \max \mathrm{fa}(\theta(\mu\alpha.T\,\vec{\tau})) \\
N(\theta, \mu\alpha.\forall(\mathrm{a}+1).\tau) & = & \mu\alpha.(\mathrm{a})\,\boldsymbol{\forall}\,N(\theta \circ \{\alpha \mapsto \mathrm{a}\}, \tau) \\
 & & \text{if } a = \max \mathrm{fa}(\theta(\mu\alpha.\forall(\mathrm{a}+1).\tau)) \\
 & & \\
N(\tau) & = & N(id, \tau)
\end{array}
$$

Figure 6: The encoding

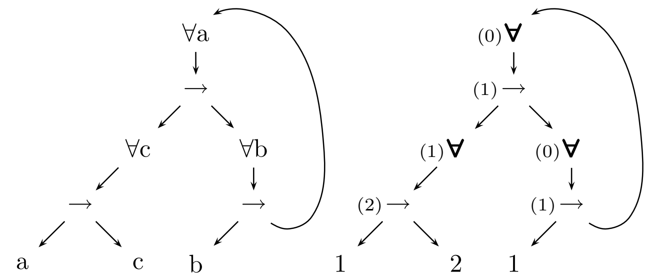

Figure 7: A type and its encoding

types, atoms are still used as interchangeable *names*. At the level of terms, atoms are never bound, so their identity is observable, and they really are *numbers*. In other words, the purpose of our encoding is to map *names* to *numbers*.

3.2 The encoding

We are now ready to present the encoding. Let us recall that it is a function N of types to terms, and that we intend it to define canonical forms, that is, we intend $\tau =_{\mu a} \tau'$ to be *equivalent* to $N(\tau) =_\mu N(\tau')$.

The definition of the encoding appears in Figure 6. We first define a function N of two parameters, namely, a substitution θ and a type τ. The substitution θ is used to associate information with μ-bound variables. It is initially empty: we define $N(\tau)$ as a shorthand for $N(id, \tau)$, where id is the identity substitution. When τ is nonrecursive, the parameter θ is irrelevant and may be ignored. We recommend doing so upon first reading of the equations in Figure 6.

To begin, note that the encoding is *structure-preserving*: every variable is mapped to itself, every atom is mapped to an atom, and every constructor application is mapped to an application of the same constructor. In other words, the sole effect of the encoding is to fix the numbering of atoms.

One might wonder how it is possible for the encoding to impose a numbering of atoms, since the second equation in Figure 6 seems to state that every atom is mapped to itself. The truth is, it only states that an atom is mapped to itself *if it appears at the root of the type.* More generally, it is possible to check that every atom that occurs *free* in the original type is mapped to itself by the encoding. Such a fact is, however, of little value, because, in the end, we are interested in *atom-closed* types, which have no free atoms. So, the key question is, how does the encoding deal with *bound* atoms?

To answer this question, let us examine the encoding of universal types, which bind atoms. Because the encoding must be canonical, $=_{\mu a}$-equivalent types must be mapped to $=_\mu$-equivalent terms. For instance, the types $\tau_1 = \mu\alpha.\forall \mathrm{a.a} \to \mathrm{c} \to \alpha$ and $\tau_2 = \forall \mathrm{b.b} \to \mathrm{c} \to \mu\alpha.\forall \mathrm{a.a} \to \mathrm{c} \to \alpha$, which are $=_{\mu a}$-equivalent, must receive $=_\mu$-equivalent encodings. This requires agreeing on a common name d for the atom that is bound at their root. By α-conversion, τ_1 and τ_2 may be written $\mu\alpha.\forall \mathrm{d.d} \to \mathrm{c} \to \alpha$ and $\forall \mathrm{d.d} \to \mathrm{c} \to \mu\alpha.\forall \mathrm{a.a} \to \mathrm{c} \to \alpha$, respectively, *where d is any atom other than c,* since c occurs free in τ_1 and τ_2 and must not be captured. In order to choose d in a deterministic manner, *we let d be the successor of c.* Because atoms are natural integers, this definition makes sense.

In the general case, when encoding a universal type $\forall \mathrm{a}'.\tau$, we require the bound atom a' to be the *successor of the greatest atom that occurs free in* $\forall a'.\tau$. In other words, we require a' to be $\mathrm{a}+1$, where a is $\max \mathrm{fa}(\forall \mathrm{a}'.\tau)$. (By convention, $\max \varnothing$ is 0.) If a' does not meet this requirement, then an α-conversion step must be performed. Because, by construction, $\mathrm{a}+1$ does not occur free in $\forall \mathrm{a}'.\tau$, such a step must be possible, which means that, in spite of this requirement, the encoding remains a total function. Also, it is important to keep in mind that, if two types τ and τ' are related by $=_{\mu a}$, then their sets of free atoms must coincide, so the atoms $\max \mathrm{fa}(\tau)$ and $\max \mathrm{fa}(\tau')$ must coincide as well. This is key to proving that the encoding maps $=_{\mu a}$-equivalent types to $=_\mu$-equivalent terms.

To sum up the idea exposed in the previous paragraph, here is a simplified version of the fourth equation in Figure 6, which makes sense when types are nonrecursive. Then, μ binders disappear, and the substitution θ is suppressed:

$$
\begin{array}{lcl}
N(\forall(\mathrm{a}+1).\tau) & = & (\mathrm{a})\,\boldsymbol{\forall}\,N(\tau) \\
 & & \text{if } a = \max \mathrm{fa}(\forall(\mathrm{a}+1).\tau)
\end{array}
$$

The effect of the side condition is to determine the value of a. For readers who find its apparently circular formulation mysterious, here is an equivalent version where the required α-conversion step is made explicit:

$$
\begin{array}{lcl}
N(\forall \mathrm{b}.\tau) & = & (\mathrm{a})\,\boldsymbol{\forall}\,N(\{\mathrm{b} \mapsto \mathrm{a}+1\}\tau) \\
 & & \text{if } a = \max \mathrm{fa}(\forall \mathrm{b}.\tau)
\end{array}
$$

In short, to encode $\forall \mathrm{b}.\tau$, one computes the greatest atom a that occurs free in $\forall \mathrm{b}.\tau$, renames b to $\mathrm{a}+1$ in τ, and proceeds with the encoding of (the renamed version of) τ.

To complete our explanation of the fourth equation in Figure 6, we must describe the machinery that deals with recursive types. As pointed out earlier, the encoding is structure-preserving: every μ binder and every variable is kept unchanged. There is only one subtlety: when computing the set of free atoms at a certain node in the input type, one must account for the free atoms contributed by the reverse edges that point back *above* that node. Consider, for instance, the type $\forall \mathrm{a}.\tau$, where τ stands for $\mu\alpha.\mathrm{a} \to \forall \mathrm{b.b} \to \alpha$. (A graphic representation appears in Figure 8.) Strictly speaking, we have $\mathrm{fa}(\forall \mathrm{b.b} \to \alpha) = \varnothing$, so b can safely be renamed to any atom, including a. However, an unfolding of τ is $\mathrm{a} \to \forall \mathrm{b.b} \to \tau$, where b *cannot* be renamed to a, because $\mathrm{fa}(\forall \mathrm{b.b} \to \tau)$ is $\{\mathrm{a}\}$. We claim that it is necessary to rename b in a manner that is correct not only with respect to τ, but also with respect to all of its unfoldings. (We come back to this point in §5.) For this reason, when computing the free atoms of $\forall \mathrm{b.b} \to \alpha$, one should not view α as a leaf that has no free atoms. Instead,

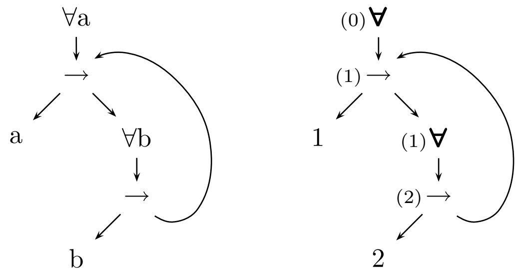

Figure 8: A type and its encoding

one should follow the reverse edge from α to τ, and, since $\mathrm{fa}(\tau)$ is $\{\mathrm{a}\}$, consider that α contributes the free atom a. By proceeding in such a manner, one is lead to renaming b to the successor of a, a choice that is safe with respect to all unfoldings of τ.

Technically, this idea is implemented as follows. When examining the node $\tau = \mu\alpha.\ldots$, we evaluate $\max \mathrm{fa}(\tau)$, yielding a. Then, we create the substitution $\theta = \{\alpha \mapsto \mathrm{a}\}$, so as to record the fact that every occurrence of α stands for a type whose greatest free atom is a. (One could equivalently define θ as $\{\alpha \mapsto \tau\}$; see Lemma 4.1.) Upon reaching the node $\forall\mathrm{b}.\ldots$, we compute $\max \mathrm{fa}(\theta(\forall\mathrm{b}.\mathrm{b} \to \alpha))$, which due to the presence of θ is a, and conclude that b should be renamed to the successor of a. This explains the role of θ in the definition of N.

The third equation in Figure 6 is analogous to the fourth one. Because nodes of the form $\mu\alpha.T\,\vec{\tau}$ do not bind atoms, no α-conversion takes place. We simply update θ as above.

Example Figure 7 depicts the type $\tau = \mu\alpha.\forall\mathrm{a}.(\forall\mathrm{c}.\mathrm{a} \to \mathrm{c}) \to \forall\mathrm{b}.\mathrm{b} \to \alpha$ and its image through N. Here is how the latter is computed. Because τ has no free atoms, its root node is annotated with (0), and the atom a is renamed to 1. (In particular, observe that the right-hand term's leftmost leaf is 1.) Then, one moves down to the next node, an arrow constructor. Its only free atom is a, that is, 1, so it is annotated with (1), and one moves down to its children, which are respectively labelled $\forall$c and $\forall$b. As for the former, the greatest free atom of $\forall\mathrm{c}.\mathrm{a} \to \mathrm{c}$ is a, that is, 1, so the $\boldsymbol{\forall}$ node is annotated with (1), and c is renamed to 2. As for the latter, there are no free atoms below this node (the reverse edge does not contribute any, because τ is atom-closed), so it is labeled with (0), and b is renamed to 1. $\diamond$

Example Figure 8 depicts the type $\tau = \forall\mathrm{a}.\mu\alpha.\mathrm{a} \to \forall\mathrm{b}.\mathrm{b} \to \alpha$ and its image through N. Here is how the latter is computed. As in the previous example, τ is atom-closed, so its root node is labeled (0), the atom a is renamed to 1, and the topmost arrow node is labeled (1). Let us now consider the arrow's right child, a $\forall$ node. Its greatest free atom is a, which is contributed by the reverse edge. As a result, the $\boldsymbol{\forall}$ node is annotated with (1), and b is renamed to 2. $\diamond$

4 Correctness of the encoding w.r.t. equality

When computing the greatest free atom of some type, replacing a subtree with its own greatest free atom does not affect the end result.

Lemma 4.1 *If a is* $\max \mathrm{fa}(\theta\tau')$*, then* $\max \mathrm{fa}(\theta\{\alpha \mapsto \tau'\}\tau)$ *and* $\max \mathrm{fa}(\theta\{\alpha \mapsto a\}\tau)$ *coincide.* $\diamond$

The encoding commutes with substitutions of types for type variables. This is a key property.

Lemma 4.2 *If a is* $\max \mathrm{fa}(\theta\tau')$*, then* $N(\theta, \{\alpha \mapsto \tau'\}\tau)$ *is* $\{\alpha \mapsto N(\theta, \tau')\}(N(\theta \circ \{\alpha \mapsto a\}, \tau))$. $\diamond$

Proof. Assume $\mathrm{a} = \max \mathrm{fa}(\theta\tau')$ (**1**). The proof is by structural induction on τ. The result is immediate when τ is a variable or an atom. We omit the case where τ is the application of a type constructor T, because it is subsumed by the last case, where τ is a universal type. Thus, we focus on the last case. We may assume $\alpha \in \mathrm{fv}(\tau)$ (**2**), since the result is otherwise immediate.

Let b stand for $\max \mathrm{fa}(\theta\{\alpha \mapsto \tau'\}\tau)$. By Lemma 2.3, we have $\mathrm{b} \geq \max \mathrm{fa}(\{\alpha \mapsto \tau'\}\tau)$ (**3**). By Lemma 2.3 again, (3) implies $\mathrm{b} \geq \max \mathrm{fa}(\tau)$, whence $\mathrm{b}+1 \notin \mathrm{fa}(\tau)$ (**4**). Furthermore, we let the reader check that (3) and (2) imply $\mathrm{b} \geq \max \mathrm{fa}(\tau')$, whence $\mathrm{b}+1 \notin \mathrm{fa}(\tau')$ (**5**). Last, by Lemma 4.1 and by (1), we have $\mathrm{b} = \max \mathrm{fa}(\theta\{\alpha \mapsto \mathrm{a}\}\tau)$ (**6**).

Because τ is a universal type, and by (4), we may write τ under the form $\mu\beta.\forall(\mathrm{b}+1).\tau_1$ (**7**), where $\beta \neq \alpha$ (**8**) and $\beta \notin \mathrm{fv}(\tau')$ (**9**) hold. Then, thanks to (9), (8), and (5), $\{\alpha \mapsto \tau'\}\tau$ is $\mu\beta.\forall(\mathrm{b}+1).\{\alpha \mapsto \tau'\}\tau_1$ (**10**).

Let θ' stand for $\theta \circ \{\beta \mapsto \mathrm{b}\}$. By (9), we have $\theta\tau' = \theta'\tau'$, which together with (1) implies $\mathrm{a} = \max \mathrm{fa}(\theta'\tau')$ (**11**). Also by (9), we have $N(\theta', \tau') = N(\theta, \tau')$ (**12**), and $\beta \notin \mathrm{fv}(N(\theta, \tau'))$ (**13**).

We may now proceed as follows:

$$
\begin{array}{cl}
 & N(\theta, \{\alpha \mapsto \tau'\}\tau) \\
= & N(\theta, \mu\beta.\forall(\mathrm{b}+1).\{\alpha \mapsto \tau'\}\tau_1) \\
 & \text{by (10)} \\
= & \mu\beta.(\mathrm{b})\boldsymbol{\forall}\, N(\theta', \{\alpha \mapsto \tau'\}\tau_1) \\
 & \text{by definition of b and } \theta' \\
= & \mu\beta.(\mathrm{b})\boldsymbol{\forall}\, \{\alpha \mapsto N(\theta', \tau')\}(N(\theta' \circ \{\alpha \mapsto \mathrm{a}\}, \tau_1)) \\
 & \text{by (11) and by the induction hypothesis} \\
= & \{\alpha \mapsto N(\theta, \tau')\}(\mu\beta.(\mathrm{b})\boldsymbol{\forall}\, N(\theta' \circ \{\alpha \mapsto \mathrm{a}\}, \tau_1)) \\
 & \text{by (12), (8), and (13)} \\
= & \{\alpha \mapsto N(\theta, \tau')\}(N(\theta \circ \{\alpha \mapsto \mathrm{a}\}, \tau)) \\
 & \text{by (7), (6), (8), and by definition of } \theta'
\end{array}
$$
$\square$

We now reach the main theorem:

Theorem 4.3 *Let* θ *be arbitrary. Then,* $\tau =_{\mu a} \tau'$ *is equivalent to* $N(\theta, \tau) =_{\mu} N(\theta, \tau')$. $\diamond$

Proof. We first prove the left to right implication. The proof of the right to left implication, which is analogous, is omitted so as to conserve space.

Throughout, θ is arbitrary and fixed. Let R be the relation between terms defined by $N(\theta, \tau)\ R\ N(\theta, \tau')$ if and only if $\tau =_{\mu a} \tau'$. Our goal is to prove that R is a subset of $=_{\mu}$. By the coinduction principle, it suffices to prove that R is *consistent* [10] with respect to the rules in Figure 5, that is, to establish $R \subseteq E_{\mu} R$, where E_{μ} is the monotone function from relations to relations implicitly associated with the rules in Figure 5. Thus, let $\tau =_{\mu a} \tau'$ (**1**). Our goal is to prove that the pair $(N(\theta, \tau), N(\theta, \tau'))$ may be deduced, via one of the rules in Figure 5, from pairs that are members of R.

We reason by cases on the structure of τ and τ'. The cases where τ and τ' are variables or atoms are immediate. The case where they are applications of a type constructor T is subsumed by the last case, where they are universal types. Thus, we focus on the last case.

Let a stand for max $\text{fa}(\theta\tau)$ (**2**). By Lemma 2.3, we have $\text{a} \geq \max \text{fa}(\tau)$, whence $\text{a}+1 \notin \text{fa}(\tau)$ (**3**). By (1) and by Lemmas 2.1 and 2.2, we also have $\text{a} = \max \text{fa}(\theta\tau')$ (**4**) and $\text{a}+1 \notin \text{fa}(\tau')$ (**5**).

By (3) and (5), we may write τ and τ' under the form $\mu\alpha.\forall(\text{a}+1).\tau_1$ and $\mu\alpha.\forall(\text{a}+1).\tau'_1$, respectively. By definition of $=_{\mu a}$, we then have $\{\alpha \mapsto \tau\}\tau_1 =_{\mu a} \{\alpha \mapsto \tau'\}\tau'_1$ (**6**). By definition of N and by (2), $N(\theta,\tau)$ is $\mu\alpha.(\text{a})\boldsymbol{\forall} N(\theta\circ\{\alpha \mapsto \text{a}\},\tau_1)$. Similarly, $N(\theta,\tau')$ is $\mu\alpha.(\text{a})\boldsymbol{\forall} N(\theta\circ\{\alpha \mapsto \text{a}\},\tau'_1)$. Thus, by applying the last rule in Figure 5, the goal becomes to prove that the terms $\{\alpha \mapsto N(\theta,\tau)\}(N(\theta\circ\{\alpha \mapsto \text{a}\},\tau_1))$ and $\{\alpha \mapsto N(\theta,\tau')\}(N(\theta\circ\{\alpha \mapsto \text{a}\},\tau'_1))$ are related by R. By (2), (4), and Lemma 4.2, these terms are precisely $N(\theta,\{\alpha \mapsto \tau\}\tau_1)$ and $N(\theta,\{\alpha \mapsto \tau'\}\tau'_1)$. By (6) and by definition of R, they are related by R. $\square$

As an immediate corollary, we obtain:

Theorem 4.4 *$\tau =_{\mu a} \tau'$ is equivalent to $N(\tau) =_\mu N(\tau')$.* $\diamond$

Theorem 4.4 yields a new decision procedure for type equality in F_μ. Indeed, whether two first-order recursive terms are related by $=_\mu$ may be decided in time $O(n\alpha(n))$, using a standard first-order unification algorithm, such as Huet's [14]. Thus, in order to obtain an efficient decision procedure for $=_{\mu a}$, there only remains to find an efficient method for computing N. This is the topic of §7.

5 Correctness of the encoding w.r.t. unifiability

We have shown that the encoding allows reducing the equality problem from the second order to the first order. We would now like to generalize this result to the problem of unification.

We begin with a few definitions. A (type) substitution θ is *atom-closed* if and only if every type in its image is atom-closed. An *atom-closed* substitution θ *unifies* τ and τ' if and only if $\theta\tau =_{\mu a} \theta\tau'$ holds. τ and τ' are *unifiable* if and only if some atom-closed substitution θ unifies them. A (term) substitution φ *unifies* σ and σ' if and only if $\varphi\sigma =_\mu \varphi\sigma'$ holds. σ and σ' are *unifiable* if and only if some substitution φ unifies them.

A key property, which follows directly from Lemma 4.2, is the following: *if τ' is atom-closed, then $N(\{\alpha \mapsto \tau'\}\tau)$ is* $\{\alpha \mapsto N(\tau')\}N(\tau)$. More generally, the encoding commutes with atom-closed substitutions, as stated by the following lemma. We write $N(\theta)$ for the image of θ through the encoding, defined as the substitution that maps every variable α to the term $N(\theta\alpha)$, and lifted to a function of terms to terms in the standard way. (It must not be confused with $N \circ \theta$, a function from types to terms.)

Lemma 5.1 *If θ and τ are atom-closed, then $N(\theta)(N(\tau))$ is $N(\theta\tau)$.* $\diamond$

This lemma is the main reason why it is meaningful to attempt to unify encodings of types. It immediately allows proving the first result of this section:

Theorem 5.2 *Let τ and τ' be atom-closed. If θ unifies τ and τ', then $N(\theta)$ unifies $N(\tau)$ and $N(\tau')$.* $\diamond$

Proof. Let τ and τ' be atom-closed. Assume θ unifies τ and τ'. By definition, θ is assumed to be atom-closed, and $\theta\tau =_{\mu a} \theta\tau'$ holds. Then, we have

$$\begin{array}{rcll} N(\theta)(N(\tau)) & = & N(\theta\tau) & \text{by Lemma 5.1} \\ & =_\mu & N(\theta\tau') & \text{by Theorem 4.3} \\ & = & N(\theta)(N(\tau')) & \text{by Lemma 5.1} \quad \square \end{array}$$

$$\begin{array}{rcl} Q(\alpha) & = & \alpha \\ Q(\text{a}) & = & \text{a} \\ Q(\mu\alpha.(\text{a})\,T\,\vec{\sigma}) & = & \mu\alpha.T\,Q(\vec{\sigma}) \\ Q(\mu\alpha.(\text{a})\,\boldsymbol{\forall}\,\sigma) & = & \mu\alpha.\forall(\text{a}+1).Q(\sigma) \end{array}$$

Figure 9: The decoding

In words, if two types are unifiable, then so are their encodings. Now, we would like to prove a converse of this theorem, that is, to deduce second-order unifiability from first-order unifiability. Let's look at a few examples, to help develop an intuition. Suppose we wish to know if $\forall\text{a.a} \to \beta$ and $\forall\text{a.a} \to \forall\text{b.b}$ are unifiable. Encoding these types yields a first-order unification problem:

$$(0)\boldsymbol{\forall}(1 \overset{(1)}{\to} \beta) =^? (0)\boldsymbol{\forall}(1 \overset{(1)}{\to} (0)\boldsymbol{\forall}1),$$

whose most general unifier is $\{\beta \mapsto (0)\boldsymbol{\forall}1\}$. Applying a *decoding* to this term substitution, we obtain the type substitution $\{\beta \mapsto \forall\text{b.b}\}$, wich unifies the initial problem, and is indeed its most general unifier.

The *decoding* Q, which we have alluded to above, is defined in Figure 9. Its definition is extremely simple. Atoms and variables are preserved. At constructor application nodes, the annotation (a) is erased. At $\boldsymbol{\forall}$ nodes, a universal quantifier is re-introduced, with the convention that the bound atom is $\text{a}+1$. The next lemma states that the decoding Q is indeed the inverse of the encoding.

Lemma 5.3 *$Q(N(\tau))$ is τ.* $\diamond$

If φ is a term substitution, we define its image through the decoding $Q(\varphi)$ as the type substitution that maps a variable α to $Q(\varphi\alpha)$. It is lifted to a function of types to types in the standard way. (Again, it must not be confused with $Q \circ \varphi$, a function from terms to types.)

The last example was extremely simple. Unfortunately, things do not always work out so easily: two *non*-unifiable types may have unifiable encodings. Consider, for example, the unsatisfiable problem $\forall\text{a.a} \to \beta =^? \forall\text{a.a} \to \text{a}$. Its image through the encoding is

$$(0)\boldsymbol{\forall}(1 \overset{(1)}{\to} \beta) =^? (0)\boldsymbol{\forall}(1 \overset{(1)}{\to} 1),$$

whose most general unifier is $\{\beta \mapsto 1\}$. Applying Q to this term substitution, we obtain $\{\beta \mapsto 1\}$, a type substitution that is *not atom-closed*, and that does not solve the original unification problem.

This example suggests that a first-order unifier is no good unless its image through Q is atom-closed. Let us call *atom-friendly* a term, or term substitution, whose image through Q is atom-closed. We will eventually prove that the existence of an atom-friendly first-order unifier does imply that of a second-order unifier.

However, this intuitive result hides a technical difficulty: *the decoding Q does not preserve equality*, that is, $\sigma =_\mu \sigma'$ does *not* imply $Q(\sigma) =_{\mu a} Q(\sigma')$. Consider, for example, the terms and types in Figure 10. The term σ at upper left is such that the decoding of an unfolding of σ (lower right) is not $=_{\mu a}$-equivalent to the decoding of σ (upper right). The problem is that a valid first-order unfolding step may, due to capture, correspond to an invalid second-order unfolding

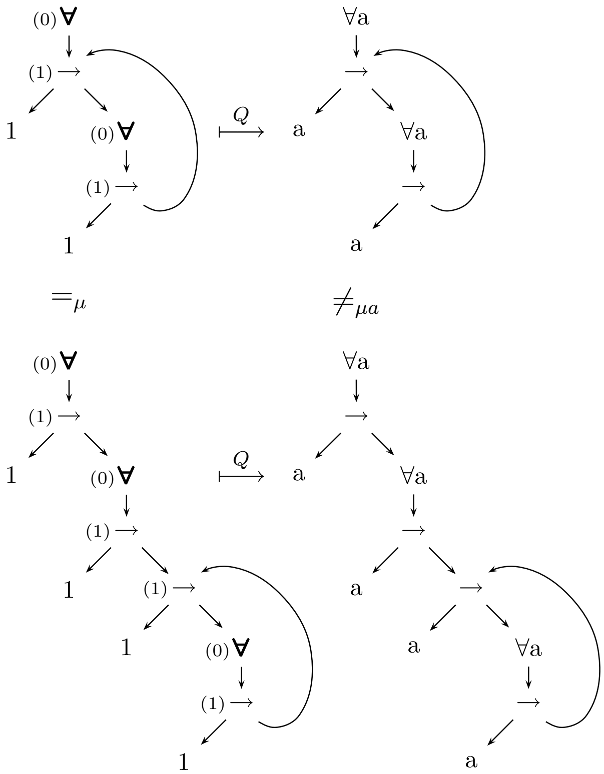

Figure 10: Term unfolding versus type unfolding

step. In other words, the image through Q of a first-order unifier is not necessarily a second-order unifier!

It is worth noting that, for terms that lie in the image of N, the decoding does preserve equality. This is a consequence of Theorem 4.4 and Lemma 5.3. Thus, the term σ at upper left in Figure 10 is not in the image of N. Indeed, it is *not* the encoding of the type τ that appears left in Figure 8, even though τ *is* $Q\sigma$. The presence of the reverse edge is the reason why b was numbered 2, instead of 1, in Figure 8, and it is also the cause of the problem in Figure 10. This is not fortuitous: the encoding was designed to avoid producing problematic terms such as σ.

In the following, we identify a subset of the terms where Q does preserve equality. We refer to these terms as *cycle-friendly*. Furthermore, we prove that every term is related by $=_\mu$ to some cycle-friendly term. This allows us to argue that, if a first-order unification problem admits a unifier, then it admits a cycle-friendly unifier, which does give rise, through Q, to a second-order unifier.

We now give the formal definitions and lemmas required to carry out the development outlined in the previous paragraphs. The end of this section is quite technical. Upon first reading, the reader might wish to skim through it and devote particular attention only to Theorems 5.18 and 5.19.

The *top atom* $\mathrm{ta}(\sigma)$ of a term σ is defined as follows: the top atom of a variable α is 0; the top atom of the terms a, $\mu\alpha.(\mathrm{a})\,T\,\vec{\sigma}$, and $\mu\alpha.(\mathrm{a})\,\forall\,\sigma$ is a.

We continue with a notion of *well-formedness* for first-order terms, whose definition appears in Figure 11. (The substitution φ is omitted in a judgement when it is the identity.) The interest of this notion lies in the following lemma:

$$\frac{}{\varphi \vdash \alpha \text{ wf}} \qquad \frac{\mathrm{a} > 0}{\varphi \vdash \mathrm{a} \text{ wf}}$$

$$\frac{\mathrm{ta}(\varphi\vec{\sigma}) \le \mathrm{a} \qquad \varphi \circ \{\alpha \mapsto \mathrm{a}\} \vdash \vec{\sigma} \text{ wf}}{\varphi \vdash \mu\alpha.(\mathrm{a})\,T\,\vec{\sigma} \text{ wf}}$$

$$\frac{\mathrm{ta}(\varphi\sigma) \le \mathrm{a}+1 \qquad \varphi \circ \{\alpha \mapsto \mathrm{a}\} \vdash \sigma \text{ wf}}{\varphi \vdash \mu\alpha.(\mathrm{a})\,\forall\,\sigma \text{ wf}}$$

Figure 11: Well-formed terms

$$\frac{}{C \vdash \mathrm{a} \text{ cfr}} \qquad \frac{\alpha \notin \mathrm{dom}\, C}{C \vdash \alpha \text{ cfr}} \qquad \frac{\alpha \notin \vec{\alpha} \qquad \mathrm{a} \le \vec{\mathrm{a}}}{C; (\alpha, \mathrm{a}); (\vec{\alpha}, \vec{\mathrm{a}}) \vdash \alpha \text{ cfr}}$$

$$\frac{C; (\alpha, \mathrm{a}) \vdash \vec{\sigma} \text{ cfr}}{C \vdash \mu\alpha.(\mathrm{a})\,T\,\vec{\sigma} \text{ cfr}} \qquad \frac{C; (\alpha, \mathrm{a}) \vdash \sigma \text{ cfr}}{C \vdash \mu\alpha.(\mathrm{a})\,\forall\,\sigma \text{ cfr}}$$

Figure 12: Cycle-friendly terms

when a term is well-formed, its top atom is an upper bound for the atoms that occur free in its decoding.

Lemma 5.4 $\vdash \sigma$ wf *implies* $\max \mathrm{fa}(Q\sigma) \le \mathrm{ta}(\sigma)$. $\diamond$

If σ is well-formed, then its atoms are required to be positive. Thus, if σ is well-formed and has a null top atom, then no atom can be free in $Q(\sigma)$.

Well-formedness is a local property: it imposes constraints between the atom carried by a node and those carried by its children. For this reason, it is preserved by several basic operations, such as unfolding and unification.

Lemma 5.5 $\sigma =_\mu \sigma'$ *and* $\vdash \sigma$ wf *imply* $\vdash \sigma'$ wf. $\diamond$

Lemma 5.6 *If σ and σ' are well-formed, then so is their most general unifier, provided it exists.* $\diamond$

Well-formedness allows stating a generalized version of Lemma 5.3:

Lemma 5.7 *If φ is well-formed and atom-friendly, then $Q(\varphi)(\tau)$ is $Q(\varphi(N(\tau)))$.* $\diamond$

The definition of cycle-friendliness appears in Figure 12. The context C is a list of couples (α, a). It is omitted in a judgement when it is empty. In words, a term is cycle-friendly if and only if, whenever a reverse edge links a leaf α to some inner node $\mu\alpha.(\mathrm{a})\ldots$, the atoms that lie on the direct path from that node down to the leaf are greater than or equal to a. For instance, the term at upper left in Figure 10 is *not* cycle-friendly, because its reverse edge points to a node labeled (1) and there is a node labeled (0) on the path down to the origin of the reverse edge.

All terms that lie in the image of N are well-formed and cycle-friendly. Furthermore, under some conditions, these notions are preserved by substitution.

Lemma 5.8 *If τ is atom-closed, then* $\vdash N(\tau)$ wf *and* $\vdash N(\tau)$ cfr *hold.* $\diamond$

Lemma 5.9 *If φ and σ are well-formed and cycle-friendly, then so is $\varphi\sigma$.* $\diamond$

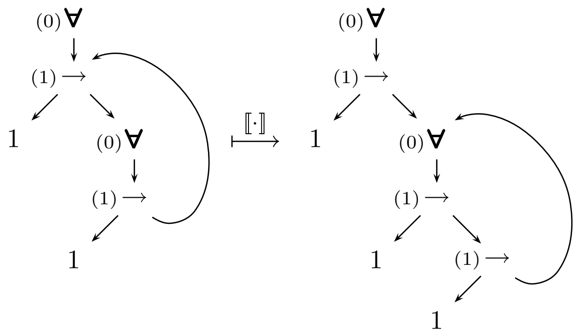

Figure 14: Normalization example

We prove some auxiliary lemmas, by induction:

Lemma 5.10 $(\alpha, a); \vec{\alpha}, \vec{a} \vdash \sigma$ cfr *and* $\alpha \notin \vec{\alpha}$ *and* $\alpha \in \mathrm{fv}(\sigma)$ *imply* $a \leq \vec{a}$. $\diamond$

Lemma 5.11 $(\alpha, \mathrm{ta}(\sigma)); \vec{\alpha}, \vec{a} \vdash \sigma_0$ cfr *and* $\vdash \sigma$ wf *and* $\alpha \notin \vec{\alpha}$ *imply* $\{\alpha \mapsto Q(\sigma)\}(Q(\sigma_0)) = Q(\{\alpha \mapsto \sigma\}\sigma_0)$. $\diamond$

Lemma 5.12 $C \vdash \sigma$ cfr *and* $\mathrm{fv}(\sigma) \cap \vec{\alpha} = \varnothing$ *imply* $(\vec{\alpha}, \vec{a}); C \vdash \sigma$ cfr. $\diamond$

Lemma 5.13 $\vdash \sigma$ cfr *and* $\vdash \sigma'$ cfr *imply* $\vdash \{\alpha \mapsto \sigma'\}\sigma$ cfr. $\diamond$

As claimed earlier, for terms that are (well-formed and) cycle-friendly, Q preserves equality. The proof is by coinduction, using the previous lemmas.

Lemma 5.14 *Assume σ and σ' are well-formed and cycle-friendly. Then, $\sigma =_\mu \sigma'$ implies $Q(\sigma) =_{\mu a} Q(\sigma')$.* $\diamond$

A *normalization* function that maps an arbitrary term σ to a $=_\mu$-equivalent, cycle-friendly term $[\![\sigma]\!]$ is defined in Figure 13. (Again, the context C is omitted when empty.) The idea behind this definition is quite simple: the term is viewed as a graph and traversed until a friendly cycle is encountered. The first rule is the stopping criterion, which checks if we can add a reverse edge to a previously seen node without breaking cycle-friendliness. The other rules simply explore and unfold the term, while recording in the context the names of the encountered nodes. An example is given in Figure 14, which shows the normalized version of the troublesome term of Figure 10.

This definition is by well-founded induction on a nonobvious ordering. A proof is required to ensure that the definition is in fact valid.

Lemma 5.15 *For every C and every σ, $[\![C, \sigma]\!]$ is well-defined.* $\diamond$

As announced above, the properties of normalization are as follows. The first lemma is proved by induction, the second by coinduction.

Lemma 5.16 $\vdash [\![\sigma]\!]$ cfr. $\diamond$

Lemma 5.17 $[\![\sigma]\!] =_\mu \sigma$. $\diamond$

At last, we are ready to prove our second result:

Theorem 5.18 *Let τ and τ' be atom-closed. If φ is atom-friendly and unifies $N(\tau)$ and $N(\tau')$, then τ and τ' are unifiable.* $\diamond$

Proof. Let φ be atom-friendly and satisfy $\varphi(N(\tau)) =_\mu \varphi(N(\tau'))$. We may assume, without loss of generality, that φ is in fact the most general unifier of $N(\tau)$ and $N(\tau')$: indeed, if some unifier is atom-friendly, then the most general unifier must be atom-friendly as well.

By Lemma 5.8, both $N(\tau)$ and $N(\tau')$ are well-formed and cycle-friendly. Thus, by Lemma 5.6, φ is well-formed. Thanks to Lemmas 5.16, 5.17, and 5.5, we may assume, without loss of generality, that φ is also cycle-friendly. Then, by Lemma 5.9, $\varphi(N(\tau))$ and $\varphi(N(\tau'))$ are well-formed and cycle-friendly. We now check that $Q(\varphi)$ unifies τ and τ':

$$\begin{array}{rcll} Q(\varphi)(\tau) & = & Q(\varphi(N(\tau))) & \text{by Lemma 5.7} \\ & =_{\mu a} & Q(\varphi(N(\tau'))) & \text{by Lemma 5.14} \\ & = & Q(\varphi)(\tau') & \text{by Lemma 5.7} \end{array} \qquad \square$$

Theorems 5.2 and 5.18 may be summed up as follows:

Theorem 5.19 *Let τ and τ' be atom-closed. τ and τ' are unifiable if and only if $N(\tau)$ and $N(\tau')$ are unifiable and their most general unifier is atom-friendly.* $\diamond$

Theorem 5.19 yields a decision procedure for unifiability of F_μ types: to determine whether two types are unifiable, one encodes them, in time $O(n \log n)$ (see §7), unifies them using a standard first-order recursive unification algorithm, in time $O(n\alpha(n))$, and checks that the most general unifier is atom-friendly. By construction, the most general unifier, if it exists, is well-formed. As a result, by Lemma 5.4, checking that it is atom-friendly amounts to checking that the top atom of every term in its image is zero. This check may be performed in time $O(n)$. Thus, the time complexity of the overall process is $O(n \log n)$.

In general, *constructing* the most general unifier of the original unification problem requires invoking the normalization function $[\![\cdot]\!]$, whose time complexity we have not yet assessed.

6 Correctness of the encoding w.r.t. entailment

The entailment problem for type equations consists in deciding, given $\tau, \tau', \alpha, \beta$, whether, for every atom-closed substitution θ, $\theta\tau =_{\mu a} \theta\tau'$ implies $\theta\alpha =_{\mu a} \theta\beta$. When this property holds, we write $\tau = \tau' \models \alpha = \beta$. The entailment problem for equations between first-order terms, written $\sigma = \sigma' \models \alpha = \beta$, is defined analogously. By exploiting the theory developed in §5, it is not difficult to prove that the former may be reduced to the latter:

Theorem 6.1 *$\tau = \tau' \models \alpha = \beta$ is equivalent to $N(\tau) = N(\tau') \models \alpha = \beta$.* $\diamond$

The entailment problem, at the first order, may be decided in time $O(n\alpha(n))$, by exploiting the following property: $\sigma = \sigma' \models \alpha = \beta$ holds if and only if either σ and σ' are *non*-unifiable or their most general unifier φ satisfies $\varphi\alpha =_\mu \varphi\beta$. As a result, the entailment problem, at the second order, may be decided in time $O(n \log n)$, where $O(n \log n)$ is the cost of the encoding (see §7).

7 Implementing the encoding

The definition of N (Figure 6) is a nice specification of the encoding, but does not suggest an efficient implementation. Indeed, it suggests traversing the source type τ, and, at every

In order of applicability :

$$
\begin{array}{lcll}
[\![C,\sigma]\!] & = & \alpha & \text{if } \exists\alpha,\sigma_0 \quad (\sigma = \alpha \vee \sigma =_\mu \sigma_0) \\
 & & & \qquad\quad \wedge \; C = (C';\alpha,\sigma_0;(\vec{\alpha},\vec{\sigma})) \\
 & & & \qquad\quad \wedge \; \mathrm{ta}(\sigma_0) \le \mathrm{ta}(\vec{\sigma}) \\
[\![C,\mathrm{a}]\!] & = & \mathrm{a} & \\
[\![C,\alpha]\!] & = & \alpha & \text{if } \alpha \notin \mathrm{dom}\, C \\
 & = & [\![C,\sigma]\!] & \text{if } C = (C';\alpha,\sigma;(\vec{\alpha},\vec{\sigma})) \text{ and } \alpha \notin \vec{\alpha} \\
[\![C,\sigma]\!] & = & \mu\alpha.(\mathrm{a})\, T\, [\![C;(\alpha,\sigma),\vec{\sigma}]\!] & \text{if } \sigma = \mu\alpha.(\mathrm{a})\, T\, \vec{\sigma} \\
[\![C,\sigma]\!] & = & \mu\alpha.(\mathrm{a})\, \forall\, [\![C;(\alpha,\sigma),\sigma']\!] & \text{if } \sigma = \mu\alpha.(\mathrm{a})\, \forall\, \sigma'
\end{array}
$$

Figure 13: Turning a term into a $=_\mu$-equivalent, cycle-friendly term

$$
\begin{array}{rcl}
\varsigma & ::= & \alpha \\
 & | & \mathrm{a} \\
 & | & [\overline{p},n,n']\mu\alpha.(\mathrm{a})\, T\, \vec{\varsigma} \\
 & | & [\overline{p},n,n']\mu\alpha.(\mathrm{a})\, \forall\, \varsigma
\end{array}
$$

Figure 15: Intermediate data structure

node τ', (i) computing the greatest atom a that occurs free in τ', taking reverse edges into account, and (ii) if an atom is bound here, renaming it to $\mathrm{a}+1$ throughout τ'. The time required by this process is quadratic in the size of τ.

Fortunately, by proceeding in a more clever manner, it is possible to achieve a better complexity bound. This is the topic of the present section. We first give a lower-level, but equivalent, definition of N. Then, we briefly describe the data structures required to implement it efficiently.

7.1 A lower-level definition of the encoding

According to the definition of N, we need to compute, for each subtree τ, the atom $\max \mathrm{fa}\, \theta\tau$, where θ depends on the context above τ and maps variables to atoms. In short, θ represents the contribution of the reverse edges whose source node lies inside τ and whose end node lies above τ. A key idea is then to exploit the following identity:

$$\max\, \mathrm{fa}(\theta\tau) = \max\,\{\max\, \mathrm{fa}\,\tau, \max\,\{\theta\alpha \,/\, \alpha \in \mathrm{fv}(\tau)\}\}$$

In words, one may separately compute the greatest atom that appears free in τ, on the one hand, and the contribution of the reverse edges that leave τ, on the other hand.

This suggests splitting the encoding process into two distinct, consecutive phases. The first phase annotates every node with the greatest atom that appears free below it, computed in a bottom-up manner. The second phase then examines each node in a top-down fashion. Using the information gathered by the first phase, it is able to compute the contribution of the reverse edges, to assign the node its definitive name, and to propagate this renaming information towards its children.

The two passes are defined in Figure 16. We now explain them.

7.1.1 First pass

The first pass is represented by the function fp. It accepts a 5-tuple of the form (l, A, R, n, τ) and returns a 4-tuple of the form (A, R, n, ς). The input-output parameters A, R, and n may be implemented using global, mutable variables.

The first pass performs a depth-first traversal of τ, the type to be encoded. Reverse edges are not traversed. Every atom or variable encountered along the way is numbered sequentially; we refer to these numbers as *positions*. The variable n, an integer counter, holds the next unassigned position. After an atom a is found at position n, the association $n \mapsto \mathrm{a}$ is recorded. The variable A, a partial mapping of positions to atoms, is used for this purpose. After a variable α is found at position n, the association $\alpha \mapsto n$ is recorded. The variable R, a relation between variables and positions, is used for this purpose.

Upon entering a node τ, the next unassigned position, that is, the current value of n, is recorded; let us refer to it as n_0. When later leaving the node, the atoms that occur (free or bound) in τ are exactly the atoms whose position (as recorded in A) is greater than or equal to n_0. This is a start, but we need to determine the atoms that occur *free* in τ.

To this end, we require bound atoms to satisfy a certain property, which one might think of as a reverse De Bruijn numbering: *the atom bound at a $\forall$ node must be the node's level*, where the level of a node is defined as the number of $\forall$ nodes that lie on the path from the root to that node. Of course, the machine representation of the type that must be encoded may not satisfy this property, so it is renamed, on the fly, as part of the first pass. The parameter l is used to hold the current level. The last equation in the definition of fp has $\forall l.\tau$ in its left-hand side, which means that whatever atom was bound here is renamed to l on the fly.

We now come back to the problem of determining the atoms that occur free under a node τ. If the node's level is l, then, by the above property, the free atoms of τ are the atoms that occur in τ and that are less than l, that is, the atoms whose position is greater than or equal to n_0 and that are less than l. Thus, the greatest free atom under τ may be written $\max\,\{\mathrm{a} \,/\, (p \mapsto \mathrm{a}) \in A \wedge \mathrm{a} < l \wedge p \ge n_0\}$. This explains why this expression appears in the third and last defining equations for fp.

The first pass produces an annotated first-order term ς, whose syntactic category is defined in Figure 15. This grammar is reminiscent of that of Figure 4. In particular, every non-leaf node carries an annotation (a), which records the greatest atom that appears free under that node. In preparation for the second pass, every node that binds a variable α also records the positions $\overline{p}$ where α occurs. In other words, these are the origins of the reverse edges that lead to the present node. (In Figure 16, we write $(\alpha \mapsto \overline{p})$ for the relation that contains $(\alpha \mapsto p)$ for every $p \in \overline{p}$.) Last, every non-leaf node records the positions n and n' that delimit its subtree: the variables that occur in its subtree

$$
\begin{array}{rcll}
\mathrm{fp}(l, A, R, n, \alpha) & = & (A, R \cup (\alpha \mapsto n), n+1, \alpha) & \\
\mathrm{fp}(l, A, R, n, \mathrm{a}) & = & (A \cup (n \mapsto \mathrm{a}), R, n+1, \mathrm{a}) & \\
\mathrm{fp}(l, A_0, R_0, n_0, \mu\alpha.T\,\tau_1 \ldots \tau_k) & = & (A_k, R', n_k, [\overline{p}, n_0, n_k]\mu\alpha.(\mathrm{a})\,T\,\varsigma_1 \ldots \varsigma_k) & \alpha \notin \mathrm{dom}(R_0) \\
& & \text{if } (A_i, R_i, n_i, \varsigma_i) = \mathrm{fp}(l, A_{i-1}, R_{i-1}, n_{i-1}, \tau_i) & \text{for } i \in \{1, \ldots, k\} \\
& & \text{and } R' \cup (\alpha \mapsto \overline{p}) = R_k & \alpha \notin \mathrm{dom}(R') \\
& & \text{and } \mathrm{a} = \max \{\mathrm{a} \,/\, (p \mapsto \mathrm{a}) \in A_k \wedge \mathrm{a} < l \wedge p \geq n_0\} & \\
\mathrm{fp}(l, A_0, R_0, n_0, \mu\alpha.\forall l.\tau) & = & (A_1, R', n_1, [\overline{p}, n_0, n_1]\mu\alpha.(\mathrm{a})\,\boldsymbol{\forall}\,\varsigma) & \alpha \notin \mathrm{dom}(R_0) \\
& & \text{if } (A_1, R_1, n_1, \varsigma) = \mathrm{fp}(l+1, A_0, R_0, n_0, \tau) & \\
& & \text{and } R' \cup (\alpha \mapsto \overline{p}) = R_1 & \alpha \notin \mathrm{dom}(R') \\
& & \text{and } \mathrm{a} = \max \{\mathrm{a} \,/\, (p \mapsto \mathrm{a}) \in A_1 \wedge \mathrm{a} < l \wedge p \geq n_0\} & \\
\\
\mathrm{sp}(l, F, \phi, \alpha) & = & \alpha & \\
\mathrm{sp}(l, F, \phi, \mathrm{a}) & = & \phi(\mathrm{a}) & \\
\mathrm{sp}(l, F, \phi, [\overline{p}, n, n']\mu\alpha.(\mathrm{a})\,T\,\vec{\varsigma}) & = & \mu\alpha.(\mathrm{b})\,T\ \mathrm{sp}(l, F \cup (\overline{p} \mapsto \mathrm{b}), \phi, \vec{\varsigma}) & \\
& & \text{if } \mathrm{b} = \max (\{\phi(\mathrm{a})\} \cup \{\mathrm{b} \,/\, (p \mapsto \mathrm{b}) \in F \wedge n \leq p < n'\}) & \\
\mathrm{sp}(l, F, \phi, [\overline{p}, n, n']\mu\alpha.(\mathrm{a})\,\boldsymbol{\forall}\,\varsigma) & = & \mu\alpha.(\mathrm{b})\,\boldsymbol{\forall}\ \mathrm{sp}(l+1, F \cup (\overline{p} \mapsto \mathrm{b}), \phi \circ \{l \mapsto \mathrm{b}+1\}, \varsigma) & \\
& & \text{if } \mathrm{b} = \max (\{\phi(\mathrm{a})\} \cup \{\mathrm{b} \,/\, (p \mapsto \mathrm{b}) \in F \wedge n \leq p < n'\}) & \\
\\
N_{alg}(\tau) & = & \mathrm{sp}(1, \emptyset, id, \pi_4(\mathrm{fp}(1, \emptyset, \emptyset, 0, \tau))) &
\end{array}
$$

Figure 16: The first and second passes of the encoding algorithm

have positions in the interval $[n, n')$.

7.1.2 Second pass

The second pass is represented by the function sp. It accepts a 4-tuple (l, F, ϕ, ς) and returns a first-order term σ. The parameter l plays the same role as in the first pass. The parameter ϕ is a renaming of atoms. It explicitly records the α-conversion steps which, in the original definition of N, were implicit.

Recall that we must compute, at each node, the maximum of (i) the greatest atom that occurs free in the subtree rooted at this node, and (ii) the greatest atom contributed by the reverse edges that leave this subtree.

As for the former, max fa(τ) was computed during the first pass, and recorded as the atom (a) carried by the node. There is, however, a subtlety: since we are applying the renaming ϕ, on the fly, to the term at hand, we really wish to compute max fa$(\phi\tau)$. Fortunately, it is possible to prove that ϕ is increasing on fa(τ). (In other words, $l, l' \in \mathrm{fa}(\tau)$ and $l < l'$ imply $\phi(l) < \phi(l')$. This holds mainly because, by construction, $\phi(l')$ is at least $\max \{\mathrm{fa}(\phi\tau) \setminus \phi(l')\} + 1$.) As a result, max fa$(\phi\tau)$ is $\phi(\max \mathrm{fa}(\tau))$, that is, $\phi(\mathrm{a})$. This explains why $\phi(\mathrm{a})$ appears in the third and last defining equations for sp.

As for the latter, we maintain a structure F that plays almost the same role as θ in Figure 6, but, instead of mapping variables to atoms, maps positions (of said variables) to atoms. Consider a node that was annotated, during the first pass, with the interval $[n, n')$. Every variable that appears free in the subtree rooted at this node appears at a position in the interval $[n, n')$. Thus, the greatest atom contributed by the free variables of this subtree (that is, by the reverse edges that leave this subtree) is $\max \{\mathrm{b} \,/\, (p, \mathrm{b}) \in F \wedge n \leq p < n'\}$. This explains why this expression appears in the third and last defining equations for sp.

The previous two paragraphs explain the definition of b in the third and last defining equations for sp. Once b is known, the node is definitively annotated with (b). If an atom is bound at this node (then, it must be l, the node's level), it must be definitively renamed to $\mathrm{b}+1$, which explains why ϕ is composed with $\{l \mapsto \mathrm{b}+1\}$ in the last defining equation for sp.

Last, once the current node has been annotated with b, we know that the reverse edges whose endpoint is this node should be viewed as contributing b to the greatest free atom computation. If the variable bound at the current node is α, then the origins of these edges are the (free) occurrences of α in the subtree rooted at this node, whose positions have been determined during the first pass, and recorded as $\overline{p}$. Thus, before moving on to the current node's children, we update F with the mapping $(\overline{p} \mapsto \mathrm{b})$, which stands for $\{(p \mapsto \mathrm{b}) \,/\, p \in \overline{p}\}$.

7.2 Correctness

Composing the first and second passes yields a mapping N_{alg} of types to terms, whose definition appears in Figure 16. (There, π_4 stands for the function that projects the fourth component out of a tuple.) As desired, N_{alg} provides a correct implementation of the encoding N. This is stated by the following theorem, whose proof is omitted:

Theorem 7.1 $N(\tau) = N_{alg}(\tau)$. $\diamond$

7.3 Complexity

We have divided the encoding task in two passes, each of which consists of a tree traversal. Let n measure the size of the input type τ. One may check that the size of the term ς produced by the first pass is bounded by $O(n)$, even though some nodes are annotated with lists of positions $\overline{p}$. This is because every position $p \in \overline{p}$ represents a distinct variable occurrence in τ. Similarly, the size of the data structures R, A, and F is bounded by $O(n)$.

Then, in order to show that the time complexity of the encoding is $O(n \log n)$, we must check that the amount of work performed at each node, during each pass, is bounded by $O(\log n)$.

By inspection of Figure 16, the non-constant time operations performed at a node are: renaming operations (implicit in the first pass, where the reverse De Bruijn numbering property is enforced, and explicit in the second pass, where the renaming ϕ is constructed and applied), and interactions with the data structures R, A, and F. We study them below.

The renaming operations, which consist in applying a renaming to an atom or extending a renaming with a new binding, may be implemented in time $O(\log n)$ using some flavor of balanced trees. In the second pass, they may in fact be implemented in time $O(1)$ and in linear space, using an array. Indeed, the elements of the domain of ϕ are levels, and the maximum level, which is bounded by the depth of the tree, can easily be computed ahead of time.

Concerning R, the required operations are inserting a new binding, and retrieving and removing all bindings associated with a given variable. Provided variables carry an integer identifier, a map of integers to integer lists, implemented using balanced trees, again does the job in time $O(\log n)$.

Concerning A, the required operations are inserting a new binding and, given n_0 and l, computing $\max \{\mathrm{a} \,/\, (p \mapsto \mathrm{a}) \in A \wedge \mathrm{a} < l \wedge p \geq n_0\}$. The latter may in fact be decomposed into two simpler operations, namely, given A and l, extracting the subset $\{(p \mapsto \mathrm{a}) \,/\, (p \mapsto \mathrm{a}) \in A \wedge \mathrm{a} < l\}$ and, given A and n_0, computing $\max \{\mathrm{a} \,/\, (p \mapsto \mathrm{a}) \in A \wedge p \geq n_0\}$. To implement these operations efficiently, one can use a binary trie where keys are atoms, with an additional invariant: each node of the trie records the maximum position that occurs in a node below it. Subset extraction then simply amounts to truncating the domain of the trie, while taking care to maintain the additional invariant. The last operation amounts to a binary search, where no backtracking is required, thanks to the additional invariant. All three operations may be implemented to run in time $O(\log n)$.

Concerning F, the required operations are inserting a binding, and, given n and n', computing $\max \{\mathrm{b} \,/\, (p \mapsto \mathrm{b}) \in F \wedge n \leq p < n'\}$. One can use the same data structure as in the previous paragraph, except the keys are now positions, and each node holds the maximum atom that occurs below it. The last operation above can be implemented by truncating F along n and n' and reading the maximum atom at the root of the resulting trie, again in time $O(\log n)$.

Thus, we have proved:

Theorem 7.2 *If τ has size n, then $N_{alg}(\tau)$ may be computed in space $O(n)$ and time $O(n \log n)$.* $\diamond$

An OCaml implementation is available online [11].

8 Conclusion

Our results are intended as a first step towards promoting the use of equirecursive types in type-preserving compilers. So far, most type-preserving compilers for object-oriented languages seem to have relied on isorecursive types, because their metatheory was better understood; see, for instance, [15]. We do believe, however, that equirecursive types are more powerful and more elegant, and should be preferred, provided appropriate decision algorithms are available.

It is worth noting that our results still hold when *rows* [19, 20] are added to the syntax of types. The definition of the encoding N requires no change. The key point is that the equational theory of rows is compatible with the notion of *free atoms*, on which the encoding relies: that is, the laws $\mathrm{fa}(l_1 : \tau_1; l_2 : \tau_2; \tau) = \mathrm{fa}(l_2 : \tau_2; l_1 : \tau_1; \tau)$ and $\mathrm{fa}(l : \tau; \partial\tau) = \mathrm{fa}(\partial\tau)$ hold. So, the reduction to first-order recursive terms is identical. There only remains to use (standard) algorithms for comparing or unifying first-order recursive terms in the presence of rows. This is an important point, since many object encodings exploit rows; see, for instance, [15].

The most natural direction for future research is to move from F_μ to F_μ^ω, since higher kinds and type operators are heavily used in many object encodings. In particular, we believe that there are natural object encodings where the μ quantifier is used at higher kinds, as opposed to only at the base kind $\star$. However, the unrestricted combination of type operators and recursive types is problematic, since it gives rise to (i) types whose infinite unfoldings are not regular and (ii) types that do not even have weak head normal forms. Thus, identifying a suitable restriction of equirecursive F_μ^ω that has decidable type equality is an attractive problem.

References

[1] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 242–252, July 1996.

[2] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

[3] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[4] Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 137–146, July 1999.

[5] Gregory D. Collins and Zhong Shao. Intensional analysis of higher-kinded recursive types. Technical Report YALEU/DCS/TR-1240, Yale University, 2002.

[6] Jeffrey Considine. Efficient hash-consing of recursive types. Technical Report 2000-006, Boston University, January 2000.

[7] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report CMU-CS-99-100, Carnegie Mellon University, 1999.

[8] Gilles Dowek. Higher-order unification and matching. In J. Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier Science, 2001.

[9] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2002.

[10] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003.

[11] Nadji Gauthier. Implementation of N. `http://caml.inria.fr/~gauthier/naming.tar.gz`, April 2004.

[12] Neal Glew. An efficient class and object encoding. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–324, October 2000.

[13] Neal Glew. A theory of second-order trees. In *European Symposium on Programming (ESOP)*, volume 2305 of *Lecture Notes in Computer Science*, pages 147–161. Springer Verlag, April 2002.

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

[15] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *ACM International Conference on Functional Programming (ICFP)*, pages 183–196, September 1999.

[16] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, March 2002.

[17] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[18] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 89–98, January 2004.

[19] Didier Rémy. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992.

[20] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[21] Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. In *Computer Science Logic*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer Verlag, August 2003.

[22] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.