Type reconstruction in F[omega]

PAWE[Lstrok] URZYCZYN

Mathematical Structures in Computer Science / Volume 7 / Issue 04 / August 1997, pp 329 - 358
DOI: 10.1017/S0960129597002302, Published online: 08 September 2000

Math. Struct. in Comp. Science (1997), *vol.* 7, *pp.* 329–358. Printed in the United Kingdom

Type reconstruction in $\mathbf{F}_\omega$

PAWEŁ URZYCZYN[†]

Institute of Informatics, University of Warsaw
ul. Banacha 2, 02-097 Warszawa, Poland
urzy@mimuw.edu.pl

Received November 1994; revised 24 June 1996

We investigate Girard's calculus $\mathbf{F}_\omega$ as a 'Curry style' type assignment system for pure lambda terms. First we show an example of a strongly normalizable term that is untypable in $\mathbf{F}_\omega$. Then we prove that every partial recursive function is nonuniformly represented in $\mathbf{F}_\omega$ (even if quantification is restricted to constructor variables of level 1). It follows that the type reconstruction problem is undecidable and cannot be recursively separated from normalization.

1. Introduction

The system $\mathbf{F}_\omega$ was introduced by J.-Y. Girard in his *Thèse d'État* (Girard 1972) (an English reference is Girard (1986)) as a tool for proving properties of higher-order propositional logics. The system is an extension of the second-order polymorphic lambda calculus, known also as 'system **F**'. The extension allows not only for (quantification over) type variables, but also variables for functions from types to types, functions from functions to functions and so on, *i.e.*, it involves an infinite hierarchy of *type constructors* classified according to their *levels* in a manner similar to the way types are divided into levels in the simply typed lambda calculus.

For example, $\forall\varphi\,((\alpha \to \varphi(\alpha)) \to (\alpha \to \varphi(\varphi(\alpha)))$ is a legal type in $\mathbf{F}_\omega$, where α is an ordinary type variable and φ is a function variable, which is applied to types and returns types. We say that α is of kind **Prop** and φ is of kind **Prop** $\Rightarrow$ **Prop**. (We use **Prop** instead of **Type** to denote the kind of ordinary types, because some authors use the latter notation for the set of all kinds.) An instantiation of φ with the type constructor $\lambda\alpha.\,\beta \to \alpha$ turns our example type into $(\alpha \to \beta \to \alpha) \to (\alpha \to \beta \to \beta \to \alpha)$. A general exposition of the properties of $\mathbf{F}_\omega$ can be found in Gallier (1990) and Giannini *et al.* (1993). See also Barendregt (1990), Barendregt and Hemerik (1990), Coquand (1990) and Pierce *et al.* (1989) – the last is recommended for instructive examples.

Girard's systems **F** and $\mathbf{F}_\omega$ are 'Church style' calculi (in the terminology of Barendregt (1990)): types are parts of expressions and there is nothing like an 'untypable term', since an untypable term is just not a term at all. There is, however, a direct connection

[†] This work was partly supported by NSF grants CCR–9113196 and CCR-9417382 and by KBN Grants 2 1192 91 01 and 2 P301 031 06. A preliminary version of this paper was presented at the Symposium 'Typed Lambda Calculus and Applications', Utrecht, The Netherlands, 1993.

between this approach and the untyped lambda calculus: we can use a 'type erasing' procedure to obtain a 'Curry style' system that assigns types to pure lambda terms. This translation between typed and untyped terms creates the *type reconstruction problem*: for a given pure lambda term, determine whether it can be assigned a type or not. To be precise, terms with free variables may be typable or not, depending on types assigned to their free variables, by a *type environment*. We write $E \vdash M : \tau$ for 'M has type τ in the environment E'. Thus, our problem has to be either restricted to closed terms or to be (equivalently) stated as follows:

$$\text{Given } M, \text{ does there exist } E \text{ and } \tau \text{ such that } E \vdash M : \tau?$$

The type reconstruction problem for systems **F** has recently been shown to be undecidable by Joe Wells in Wells (1994). Our paper addresses the problem for $\mathbf{F}_\omega$. The main result is also negative: the type reconstruction problem for $\mathbf{F}_\omega$ is undecidable. But here we have a stronger property: type reconstruction cannot be recursively separated from normalization. That is, every set of normalizable lambda terms containing all terms typable in $\mathbf{F}_\omega$ must also be undecidable.

An explanation is in order here: if all strongly normalizable terms were typable in $\mathbf{F}_\omega$, our result would be of no interest, since strong normalization is known to be undecidable, see *e.g.* Leivant (1983). Thus, one has first to ask whether the set of typable terms is a proper subset of the strongly normalizable terms. For system **F**, the first example has been given in Giannini and Ronchi (1988), more examples can be found in Malecki (1990). The example of Giannini and Ronchi (1988) is typable in $\mathbf{F}_\omega$, which means that the latter can type strictly more than **F**. Regarding the examples of Malecki (1990), we conjecture that all of them are also typable in $\mathbf{F}_\omega$.

However, we show below (Theorem 3.1) that there exist strongly normalizable terms that cannot be typed in $\mathbf{F}_\omega$. Warning: the existence of untypable terms is sometimes incorrectly derived from the known fact that the classes of integer functions representable (in the ordinary 'uniform' way) in **F** and $\mathbf{F}_\omega$ are different, and are proper subclasses of the class of recursive functions. However, the fact, *cf.* Krivine (1987), that a certain term (function representation) cannot be assigned a *particular type* (integer to integer) does not mean that it cannot be assigned any type at all. In fact, all recursive functions can be represented in the untyped lambda calculus by terms in normal form, and all normal forms are typable in **F**. Thus, the above mentioned result of Krivine (1987) does not apply here.

At first glance it might seem that determining whether a given term has a given type in a given environment could be easier that determining whether it has any type at all. That is, one could consider the following problem:

$$\text{Given } M, \ E \text{ and } \tau, \text{ does } E \vdash M : \tau \text{ hold?}$$

Let us call the above problem the 'type checking' problem. The impression we mentioned above is generally wrong, namely type reconstruction is easier: it reduces effectively to type checking. Indeed, for a given M, with free variables $\{x_1, \ldots, x_n\}$, it suffices to ask whether KyM has type α in the environment $\{(y : \alpha)\} \cup \{(x_i : \forall\alpha\alpha) \mid i = 1, \ldots, n\}$ (where K is $\lambda xy.\, x$, and y is a new variable). For system **F**, the undecidability of type checking was

an important step towards proving the undecidability of type reconstruction, and was also an open question until Wells (1994). That type checking for $\mathbf{F}_\omega$ is not decidable, could be considered folklore, as it has been independently discovered by several people (at least by Jerzy Tiuryn and the author). As far as we know, no proof of this fact has been published yet. It follows of course from our main result, but direct proofs are much shorter, so we think it is worthwhile to present one in Section 2 (Proposition 2.4).

1.1. *Related work*

The first paper we are aware of addressing the type reconstruction problem for system **F** is Leivant (1983). The decidability of this problem remained an open question for many years, despite numerous attempts to solve it. There were several partial solutions, applying either to fragments of system **F**, or to certain variations of the system (Boehm 1985; Giannini and Ronchi 1991; Giannini *et al.* 1993; Kfoury and Tiuryn 1992). The best lower bound known before 1993 was exponential, and was obtained by Henglein (Henglein 1990) (see also Henglein and Mairson (1994)). Finally, the problem was shown undecidable by Wells by a brilliant reduction from semi-unification.

A related issue is type reconstruction for the language ML and its modifications, see *e.g.* Kfoury *et al.* (1994), Kfoury *et al.* (1993a), Kanellakis and Mitchell (1989) and Mairson (1990) (the latter two merged into Kanellakis *et al.* (1991) for a final presentation). For instance, the semi-unification problem of Kfoury *et al.* (1993a), addressed first for its connection to an extension of ML (Kfoury *et al.* 1993b), was later used as a tool in Giannini *et al.* (1993) and finally in Wells (1994). In addition, there are similar techniques applied to obtain the exponential lower bounds for type inference in ML and in system **F** (Mairson 1990; Henglein and Mairson 1994). Here we would like to draw the reader's attention to the proof method used in the latter two papers, because our approach partly follows these ideas. The method is quite an involved refinement of the following simple observation: there are typable terms of length $\mathcal{O}(n)$ that can only be assigned types of depth at least exponential in n (when drawn as binary trees). Consider, for example, terms of the form $2(2(\ldots 2(K)\ldots))$, where 2 is the Church numeral $\lambda f x.f(fx)$, and K is $\lambda xy.\,x$. The idea is to replace K with a tricky representation of one step of a Turing Machine so that the whole term can represent an exponentially long computation. This TM simulation is then put in such a context that a failing computation forces untypability.

The type reconstruction problem for $\mathbf{F}_\omega$ has a shorter history. A modification of $\mathbf{F}_\omega$, similar to that of Boehm (1985), was shown in Pfenning (1993) to have undecidable type reconstruction. Another variation of the original problem: the 'conditional' type reconstruction is also undecidable by a result of Giannini *et al.* (1993). Finally, a nonelementary lower bound has been obtained by Henglein and Mairson (1994) using a similar method to that mentioned above. This time, instead of composing 2's as in the previous case, one can consider terms of the form $222\ldots2$ and apply them to the Turing Machine simulator.

1.2. *Main result*

The main result of this paper is that type reconstruction in $\mathbf{F}_\omega$ is undecidable (Theorem 5.7). The proof given below is a revised and simplified version of that given in

Urzyczyn (1992) and briefly reported in Urzyczyn (1993). However, at the cost of a little complication, it is slightly modified so that we obtain one more result, which may be of independent interest. To explain that additional result, let us first recall that an integer function f is said to be *representable* in the untyped lambda calculus iff there is a combinator F such that Fn beta-reduces to m whenever $f(n) = m$. (We identify numbers with the corresponding Church numerals.) For partial functions, this can be extended by assuming that Fn has no normal form if $f(n)$ is undefined. It is well known that all partial recursive functions are representable.

The notion of representability in typed lambda calculi is stronger: it is usually required that the term F has type $\omega \to \omega$, where ω is a common type of all numerals. In polymorphic calculi one can take, for example, $\omega = \forall\alpha\,((\alpha \to \alpha) \to \alpha \to \alpha)$, and a classical result of Girard (Girard 1986; Gallier 1990), states that the representable functions of $\mathbf{F}_\omega$ coincide with those provably total in higher-order arithmetics.

In general, a total function representable in the untyped lambda calculus by a term F does not have to be representable in a typed calculus, since F does not have the type $\omega \to \omega$, *cf.* Krivine (1987). It may happen, however, that a weaker condition is true, namely that the application Fn is typable for each n, but the type assigned to F may be different for different n. This means that numerals are used with their 'private' types rather than with one common type ω. Such a function is called *nonuniformly* representable in the underlying typed system (an undefined function value may be represented by an application Fn that does not normalize). The notion of nonuniform function representation was introduced by Leivant in Leivant (1990) for simply typed terms, but in that context it is equivalent to a very weak form of polymorphism ('askew representation') for the simple reason that there are no really 'private' types for numerals available in the simply typed calculus.

It happens to be quite different in $\mathbf{F}_\omega$. Our Theorem 5.6 states that all partial recursive functions are nonuniformly representable. The main result (Theorem 5.7) is then obtained by the obvious reduction of the halting problem, and we can also easily conclude that there is no recursive set of normalizable lambda terms containing all typable ones. In particular, whatever extension of $\mathbf{F}_\omega$ we may consider, if it has the normalization property, it must also have undecidable type reconstruction. (An extension that first comes to mind is the Calculus of Constructions. However, the Curry-style Calculus of Constructions is already known to type exactly the same terms as $\mathbf{F}_\omega$ (Giannini *et al.* 1993).)

1.3. *Proof method*

The proof of our main result is essentially based on the following trick. Consider a term W, which is (roughly) of the form

$$W \equiv \lambda x.\,\mathbf{if}\ \mathit{Final}\ x\ \mathbf{then}\ \lambda w.\,\mathit{nil}\ \mathbf{else}\ \lambda w.\,w(\mathit{Next}\ x)w\ \mathbf{fi},$$

and put it into the context WC^0W, where C^0 is a term representing an initial configuration (ID) of a Turing Machine. Of course, W is designed so that $\mathit{Next}\ x$ and $\mathit{Final}\ x$ behave as a 'next ID' function and as a test for a final ID, respectively (assuming an appropriate representation of a machine ID is substituted for x). Observe now that our term WC^0W

has the following reduction sequence:

$$WC^0W \twoheadrightarrow WC^1W \twoheadrightarrow WC^2W \twoheadrightarrow \cdots,$$

where $C^1, C^2, \ldots$ is the sequence of configurations obtained in the computation of our machine. It is not difficult to prove that WC^0W has a normal form (in fact it is strongly normalizable) iff our Turing Machine converges on the initial ID represented by C^0. The author learned this approach from Chapter 9 of Barendregt (1984), where it is used (following Kleene) to show representability of partial recursive functions in the λI-calculus.

We can, however, show more: our term is typable in F_ω iff the machine converges, and this means that the halting problem reduces to typability. In fact type reconstruction is also undecidable at each level (see below), as we need only a quite restricted machinery in our F_ω typings. The only way type constructors are used is as follows: let γ be a constructor variable of kind **Prop** $\Rightarrow$ **Prop** $\Rightarrow$ **Prop**, and take an object variable x of type $\forall\gamma\,(\gamma\tau\sigma)$ (here γ is applied to two type arguments). Now, both σ and τ are valid types for x, since γ can be instantiated by left and right projections $\pi_L = \lambda\alpha_1\alpha_2.\,\alpha_1$ and $\pi_R = \lambda\alpha_1\alpha_2.\,\alpha_2$.

The analogy that comes to mind here is that $\forall\gamma\,(\gamma\tau\sigma)$ behaves very much like an intersection type $\tau\wedge\sigma$, *cf.* Cardonne and Coppo (1990). It is known (Pottinger 1980; Leivant 1983) that all strongly normalizable terms are typable with intersection types. The reason for this is that intersection types are flexible enough for the *subject expansion property*: types are not only preserved by beta-reductions but also by beta-expansions (provided the resulting terms are strongly normalizable). The higher-order quantification does not reach that far (as we have already stated, there are untypable terms satisfying strong normalization) but still we can simulate (to some extent) the behaviour of intersection types. Thus we are able to 'pull types backward' along some reduction sequences, *i.e.*, to have a kind of restricted subject expansion behaviour (*cf.* Lemma 5.4).

With a term W as above we are able to handle the halting problem for Turing Machines. In order to show our Theorem 5.6, which concerns representability of functions, we must modify the above idea so that the input-output behaviour of Turing Machines is reflected. In particular, instead of $\lambda w.\,nil$, we must use something like $\lambda w.\,output(x)$. This adds a few technical complications. The Turing Machine coding used in this construction is essentially taken from Henglein and Mairson (1994), which in turn follows the ideas of Kanellakis and Mitchell (1989), Mairson (1990), and Kanellakis *et al.* (1991). The very few modifications we make are of a purely technical nature. However, the *typing* must be quite different in our case.

The method of proof used in Wells (1994) for system **F** differs entirely from ours. It is not known whether an extension is possible either way. Wells' proof does not seem to be easily generalized to $\mathbf{F}_\omega$, and our proof does not work for **F**, because it makes an essential use of higher-order polymorphism. In particular, it remains an open question whether the class of functions nonuniformly represented in **F** is a proper extension of the class of functions represented uniformly over type ω, and whether typability in **F** is recursively separable from normalization.

1.4. *Synopsis*

In Section 2 we introduce the system $\mathbf{F}_\omega$ by Curry style type inference rules for pure lambda terms. We do not consider the Church style (explicit typing) variant of the system at all. The reader wishing to study the relationship between the two systems is referred to Giannini *et al.* (1993). For each n, we also define the subsystems $\mathbf{F}_\omega^{(n)}$ by restricting kinds to be of level at most n.

Section 2 also contains a proof of the undecidability of type-checking (Proposition 2.4), two useful lemmas and several miscellaneous definitions. Section 3 contains a proof of 'strongly normalizable is not necessarily typable' (Theorem 3.1). Section 4 describes the Turing Machine coding and the basic strategy used to type machine configurations. Finally, Section 5 describes the typing of the iterator term W, and concludes the proofs of our main Theorems 5.6 and 5.7.

2. The system $\mathbf{F}_\omega$

2.1. *Kinds and constructors*

We begin with the notion of a *kind*, the set of all kinds defined as the smallest set containing a constant **Prop** and such that whenever ∇_1 and ∇_2 are kinds, $(\nabla_1 \Rightarrow \nabla_2)$ is a kind also. The *levels* of kinds are defined as follows: level(**Prop**) is 0, and level$(\nabla_1 \Rightarrow \nabla_2)$ is the maximum of level$(\nabla_1) + 1$ and level(∇_2).

The next step is to introduce the notion of a *constructor*. For each kind ∇, we assume an infinite set of *constructor variables of kind* ∇, denoted by Var_∇. Elements of $Var_{\mathbf{Prop}}$ are called *type variables*. We will adopt a terminological convention: whenever we talk about variables occurring in types, we mean type variables, unless we explicitly specify their kinds. For each kind ∇, we define *constructors of kind* ∇ as follows:

— A constructor variable of kind ∇ is a constructor of kind ∇.
— If φ is a constructor of kind $(\nabla_1 \Rightarrow \nabla_2)$ and τ is a constructor of kind ∇_1, then $(\varphi\tau)$ is a constructor of kind ∇_2.
— If α is a constructor variable of kind ∇_1 and τ is a constructor of kind ∇_2, then $(\lambda\alpha.\,\tau)$ is a constructor of kind $(\nabla_1 \Rightarrow \nabla_2)$.
— If α is a constructor variable of arbitrary kind and τ is a constructor of kind **Prop**, then $(\forall\alpha\tau)$ is a constructor of kind **Prop**.
— If τ and σ are constructors of kind **Prop**, then $(\tau \to \sigma)$ is a constructor of kind **Prop**.

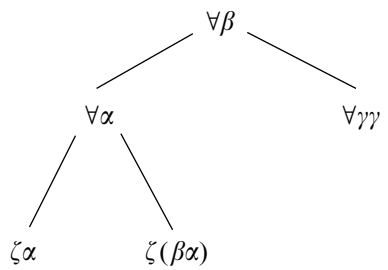
We write $\tau : \nabla$ to denote that τ is a constructor of kind ∇. Constructors of kind **Prop** are called *types*.

In order to avoid too many parentheses we adopt the conventions that applications associate to the left and arrows associate to the right (that is, $\varphi\sigma\tau$ means $(\varphi\sigma)\tau$ and $\sigma \to \tau \to \rho$ abbreviates $\sigma \to (\tau \to \rho)$, and similarly for $\Rightarrow$) and that quantifiers have higher priority than arrows (so that $\forall\alpha\tau \to \sigma$ means $(\forall\alpha\tau) \to \sigma$, rather than $\forall\alpha\,(\tau \to \sigma)$). This notation is motivated by the Curry–Howard isomorphism: since types are nothing other than formulas, quantifier scopes should be understood as in higher-order propositional calculus, *cf.*, for example, the notation of Gabbay (1981).

If τ is a constructor, $FV(\tau)$ denotes the set of all constructor variables free in τ. The definition of $FV(\tau)$ is by induction: $FV(\alpha) = \{\alpha\}$, $FV(\varphi\tau) = FV(\varphi) \cup FV(\tau)$, $FV(\sigma \to \tau) = FV(\sigma) \cup FV(\tau)$, $FV(\forall\alpha\tau) = FV(\tau) - \{\alpha\}$, and $FV(\lambda\alpha.\,\tau) = FV(\tau) - \{\alpha\}$. The symbol $\tau\{\rho/\alpha\}$ denotes the effect of substituting ρ for all free occurrences of α in τ (provided of course that everything is correctly *kinded*), after an appropriate renaming of bound variables, if necessary. Note that 'bound' means here either λ-bound or $\forall$-bound. The notation $\{\tau_1/\alpha_1, \ldots \tau_n/\alpha_n\}$ will be used for simultaneous substitutions. From now on we will identify constructors that are the same except for a renaming of bound variables.

An important definition follows: for two types σ and τ, we write $\sigma \preceq \tau$ iff σ has the form $\forall\vec{\alpha}\sigma'$, where $\forall\vec{\alpha}$ denotes a sequence of universal quantifiers, σ' has no outermost quantifiers and $\tau = \forall\vec{\beta}\sigma'\{\vec{\rho}/\vec{\alpha}\}$, for some sequence of constructors $\vec{\rho}$ of appropriate kinds and some sequence $\vec{\beta}$ of variables not free in σ. The relation $\preceq$ is called the *containment* relation (Mitchell 1988; Giannini *et al.* 1993), and it is easy to see that it is a quasi-order (*i.e.*, it is reflexive and transitive).

A type (*i.e.*, a constuctor of kind **Prop**) can be seen to be in the form of a binary tree with nodes corresponding to arrows and optionally labelled with one or more quantifiers. For instance, the type $\forall\beta\,(\forall\alpha\,(\zeta\alpha \to \zeta(\beta\alpha)) \to \forall\gamma\gamma)$, where α, γ are type variables and β, ζ are variables of kind **Prop** $\Rightarrow$ **Prop**, can be pictured as the following tree:



In Section 5 we will also use binary constructor variables as labels for binary nodes. Now we note that each path in a type tree must end with a constructor variable or an application of a constructor variable to a number of constructors. We say that the variable *owns* the path. More formally, a variable $\alpha : \nabla_1 \Rightarrow \nabla_2 \Rightarrow \cdots \Rightarrow \nabla_n \Rightarrow$ **Prop** owns a path $X \in \{L, R\}^*$ in a type σ if one of the following conditions is satisfied:

— $\sigma = \alpha\varphi_1 \ldots \varphi_n$, where φ_i's are constructors of the corresponding kinds ∇_i, and X is the empty sequence;
— $\sigma = \tau_1 \to \tau_2$, and either $X = LX'$ and α owns X' in τ_1 or $X = RX'$ and α owns X' in τ_2;
— $\sigma = \forall\beta\tau$ and α owns X in τ.

In the last condition above it may be the case that α and β is the same variable. Thus the notion of a path owner is sensitive to alpha-conversion, and will be used always with

respect to a particular choice of bound variables. Using the above pictured type as an example, γ owns R and ζ owns both LL and LR. The variables α and β own nothing. The following is a simple but useful lemma.

Lemma 2.1. Suppose a variable β owns a path X in $\forall\vec{\alpha}\sigma$, and that $\forall\vec{\alpha}\sigma \preceq \tau$. Then either $\beta \in \vec{\alpha}$ or β owns X in τ.

Proof. Easy. □

The beta-reduction on constructors is defined as in the simply typed lambda calculus: a constructor of the form $(\lambda\alpha.\tau)\rho$ is called a *redex*, and we have the usual rule:

$$(\beta) \qquad (\lambda\alpha.\tau)\rho \longrightarrow \tau\{\rho/\alpha\}.$$

The rule (β) defines the one-step reduction relation $\longrightarrow$ and the many-step reduction relation $\longrightarrow\!\!\!\!\!\rightarrow$ in the usual way. It follows from the strong normalization theorem for simply typed lambda calculus that each constructor is strongly normalizable, see, for example, Gallier (1990). In the type inference system below, we identify constructors with their normal forms: in rule (INST) we mean the normal form of $\sigma\{\rho/\alpha\}$, rather than $\sigma\{\rho/\alpha\}$ itself.

2.2. *Type assignment*

Our type inference system $\mathbf{F}_\omega$ assigns types to terms of untyped lambda calculus (object terms). These are defined as usual: first assume that we have an infinite set V of variables, then define the set of terms as the smallest set T such that $V \subseteq T$ and that for all $M, N \in T$ and $x \in V$, we also have $(MN) \in T$ and $(\lambda x.M) \in T$. We adopt all the ordinary notational conventions for lambda terms.

A *type environment* is a set E of *type assumptions*, each of the form $(x : \sigma)$, where x is a variable and σ is a type, such that if $(x : \sigma), (x : \sigma') \in E$, then $\sigma = \sigma'$. Thus, one can think of an environment as a finite partial function from variables into types. If E is an environment, $E(x : \sigma)$ is an environment such that

$$E(x : \sigma)(y) = \begin{cases} \sigma & \text{if } x \equiv y \\ E(y) & \text{if } x \not\equiv y, \end{cases}$$

and $FV(E)$ denotes the union of $FV(E(x))$ for all x such that $E(x)$ is defined.

Our system derives *type judgements* of the form $E \vdash M : \tau$, where E is an environment, M is a term and τ is a type. The system consists of the following rules

$$\text{(VAR)} \qquad E \vdash x : \sigma \qquad \text{if } (x : \sigma) \text{ is in } E$$

$$\text{(APP)} \qquad \frac{E \vdash M : \tau \to \sigma,\ E \vdash N : \tau}{E \vdash (MN) : \sigma}$$

$$\text{(ABS)} \qquad \frac{E(x : \tau) \vdash M : \sigma}{E \vdash (\lambda x.M) : \tau \to \sigma}$$

$$\text{(GEN)} \qquad \frac{E \vdash M : \sigma}{E \vdash M : \forall \alpha\, \sigma} \qquad \text{where } \alpha \notin FV(E)$$

$$\text{(INST)} \qquad \frac{E \vdash M : \forall \alpha \sigma}{E \vdash M : \sigma\{\rho/\alpha\}}$$

In rule (INST), the variable α and the type ρ must be of the same kind. Recall that an application of (INST) involves the normalization process for the constructor $\sigma\{\rho/\alpha\}$.

For a given $n \geqslant 0$ we consider a subsystem of the above type inference system, denoted by $\mathbf{F}_\omega^{(n)}$, defined by the additional restriction that only constructors of level at most n may occur in types. Thus, $\mathbf{F}_\omega^{(0)}$, allowing only for the kind **Prop**, is the second-order polymorphic lambda calculus.

We do not use the notation $\mathbf{F}_n$, because it is ambiguous, as different authors use it in different sense. In his 1986 paper (Girard 1986), J.-Y. Girard defines $\mathbf{F}_n$ as the calculus with level n constructors. This terminology has been used in Giannini and Ronchi (1988). However, several other authors, *e.g.* Gallier (1990), Henglein and Mairson (1994) and Pierce *et al.* (1989), write $\mathbf{F}_2$ for $\mathbf{F}$, thus shifting up by 2 the whole numeration $\mathbf{F}_n$.

The reader is referred to Giannini *et al.* (1993) for a discussion of the properties of $\mathbf{F}_\omega$. We recall here two useful facts. First, our system has the *subject reduction property*: if we derive $E \vdash M : \tau$ and M reduces to some M', we can also derive $E \vdash M' : \tau$. The second thing is that we may simplify the system so that it becomes 'syntax-oriented' in the following sense: the last rule used in a derivation of a type judgement $E \vdash M : \tau$ is fully determined by the shape of M. The simplified system has the following rules:

$$\text{(VAR}') \qquad E \vdash x : \tau \qquad \text{if } (x : \sigma) \text{ is in } E \text{ and } \sigma \preceq \tau$$

$$\text{(APP}') \qquad \frac{E \vdash M : \tau \to \sigma,\ E \vdash N : \tau}{E \vdash (MN) : \forall \vec{\alpha} \sigma'} \qquad \text{provided } \sigma \preceq \sigma' \text{ and } \vec{\alpha} \cap FV(E) = \varnothing$$

$$\text{(ABS}') \qquad \frac{E(x : \tau) \vdash M : \sigma}{E \vdash (\lambda x.M) : \forall \vec{\alpha}\, (\tau \to \sigma)} \qquad \text{provided } \vec{\alpha} \cap FV(E) = \varnothing$$

The following lemma is taken from Giannini *et al.* (1993), but its counterpart for $\mathbf{F}$ already occurs in Mitchell (1988).

Lemma 2.2. A type judgement $E \vdash M : \tau$ can be derived in $\mathbf{F}_\omega$ iff it can be derived in the simplified system.

We conclude this subsection with one more definition. Let $\mathbb{N}$ denote the set of all nonnegative integers. For $n \in \mathbb{N}$, we identify n with the n-th *Church numeral*, that is, the lambda term $\lambda f x.\, f^n(x)$. If $f : \mathbb{N} \to \mathbb{N}$ is a partial recursive function then a term F *represents* f *nonuniformly in* $\mathbf{F}_\omega$ iff

— for each $n \in \mathrm{Dom}(f)$, the expression Fn is typable in $\mathbf{F}_\omega$ and its normal form is $f(n)$;
— Fn does not have a normal form when $n \notin \mathrm{Dom}(f)$.

2.3. *The hierarchy of levels and the type-checking problem*

To see the system at work, let us do the following exercise: assign types to terms of the form $222\ldots2K$. This is easily done for $2K$ in system **F**, but $22K$ can already be shown untypable in **F** (Urzyczyn 1995). In $\mathbf{F}_\omega$ we can assign the type

$$\forall\beta\,(\forall\alpha\,(\alpha\to\beta\alpha)\to\forall\alpha\,(\alpha\to\beta\,(\beta\alpha)))$$

(where α : **Prop**, and β : **Prop** $\Rightarrow$ **Prop**) to the second 2 in $22K$, and the expression becomes typable. For the next step, typing $222K$, the simplest solution is to assign

$$\forall\gamma\,(\forall\beta\,(\forall\alpha\,(\alpha\to\beta\alpha)\to\forall\alpha\,(\alpha\to\gamma\beta\alpha))\to\forall\beta\,(\forall\alpha\,(\alpha\to\beta\alpha)\to\forall\alpha\,(\alpha\to\gamma(\gamma\beta)\alpha)))$$

to the middle 2, where γ is of kind (**Prop** $\Rightarrow$ **Prop**) $\Rightarrow$ (**Prop** $\Rightarrow$ **Prop**). We leave it for the reader to complete the type assignment, and to see that each $222\ldots2K$ can be typed in $\mathbf{F}_\omega$ with the help of type constructors of increasing levels.

At first glance, the use of higher level constructors seems necessary here, and one is tempted to believe that the 'typing power' of higher-order polymorphism increases together with the level of constructors allowed, especially because the term $22K$ is untypable in **F**.

It is quite surprising that, as we show in Urzyczyn (1992), all these terms are typable in $\mathbf{F}^{(1)}_\omega$, that is, constructors of level 1 (functions from types to types) are sufficient. We believe that a proper generalization of the technique developed for $22\ldots2K$ should give a proof of the following

Conjecture 2.3. All terms typable in $\mathbf{F}_\omega$ are typable in $\mathbf{F}^{(1)}_\omega$.

As we said before, the undecidability of type-checking for $\mathbf{F}_\omega$ seems to be well known, but no proof has been published. We think it is useful to have a sketch of a proof in print.

Proposition 2.4. The type checking problem for $\mathbf{F}_\omega$ is undecidable.

Sketch of proof. We reduce the second-order unification problem of Goldfarb (1981) to type checking in $\mathbf{F}_\omega$ (in fact, already in $\mathbf{F}^{(1)}_\omega$). With no loss of generality we can work with a signature consisting of one binary function symbol '$\to$' (written in infix notation) and two individual constants α and β. First we make the following observation: we can assume that an instance of second-order unification has the form of a single equation $\tau=\rho$, where the unknowns (for simplicity, all of them are binary function unknowns) occurring in the left-hand side are disjoint from those occurring in the right-hand side. It follows from Goldfarb (1981) that this assumption can be made with no loss of generality, except perhaps the last part, which requires observing that for binary unknowns ζ and ζ', the equations $\zeta\alpha\beta=\zeta'\alpha\beta$ and $\zeta\beta\alpha=\zeta'\beta\alpha$, force (informally) that $\zeta=\zeta'$, that is, the appropriate coordinates of every solution must be equal.

Having made the initial preparation, assume we are given an instance $\tau=\rho$ of second-order unification, where the unknowns of τ are $\zeta_1,\ldots,\zeta_k$ and the unknowns of ρ are $\zeta_{k+1},\ldots,\zeta_n$. We ask whether the term $Ky(x(xy))$ has type α in the environment consisting of type assumptions $(y:\forall\alpha\alpha)$ and $(x:\forall\zeta_1,\ldots,\zeta_n(\tau\to\rho))$. One can check now that this type assignment is correct iff there exists an appropriate instantiation for the ζ's that allows for function composition of the two occurrences of x, that is, iff our equation $\tau=\rho$ has a solution. □

3. Strongly normalizable but untypable

The aim of this section is to show our first main result: the type inference rules of $\mathbf{F}_\omega$ do not suffice to type all strongly normalizable terms. Our counter-example is the following term:

$$M \equiv (\lambda x.\, z(x1)(x1'))(\lambda y.\, yyy)$$

where $1 \equiv \lambda f u.\, fu$ and $1' \equiv \lambda v g.\, gv$. Clearly, M is strongly normalizable, and it is an easy exercise to see that it becomes typable in $\mathbf{F}_\omega$ after just one reduction step.

Theorem 3.1. The above term M cannot be typed in $\mathbf{F}_\omega$, and thus the class of typable terms is a proper subclass of the class of all strongly normalizable terms.

The proof of our theorem will be obtained with help of a sequence of lemmas. The first lemma is as follows.

Lemma 3.2. Suppose that M is typable in $\mathbf{F}_\omega$. Then there exist types σ, σ^*, σ°, σ_1, σ_2, σ_3, ρ, ρ^*, ρ°, π and τ such that, for appropriate vectors of variables $\vec{\alpha}$, $\vec{\beta}$, and $\vec{\delta}$:

(a) $\vdash 1 : \forall\vec{\alpha}\sigma^*$ and $\vdash 1' : \forall\vec{\alpha}\sigma^\circ$;
(b) $\forall\vec{\beta}\,(\forall\vec{\alpha}\sigma \to \rho) \preceq \forall\vec{\alpha}\sigma^* \to \rho^*$ and $\forall\vec{\beta}\,(\forall\vec{\alpha}\sigma \to \rho) \preceq \forall\vec{\alpha}\sigma^\circ \to \rho^\circ$;
(c) $\forall\vec{\alpha}\sigma \preceq \sigma_1, \sigma_2, \sigma_3$;
(d) $\sigma_1 = \sigma_2 \to \forall\vec{\delta}\tau$;
(e) $\forall\vec{\delta}\tau \preceq \sigma_3 \to \pi$, and $\pi \preceq \rho$.

In addition, we may assume that σ and τ have no outermost quantifiers.

Proof. If our term is typable in $\mathbf{F}_\omega$, then, by Lemma 2.2, it is typable in the simplified system, and we may assume that $\vdash \lambda y.\, yyy : \forall\vec{\beta}\chi$ and $\vdash \lambda x.\, z(x1)(x1') : \forall\vec{\beta}\chi \to \eta$, for appropriate χ and η. (Remember that according to our convention, the quantifier scope does *not* extend to the end of a formula.) Since $\forall\vec{\beta}\chi$ is a type assigned to an abstraction, it must be the case that $\chi = \forall\vec{\alpha}\sigma \to \rho$ for some σ, ρ, such that $y : \forall\vec{\alpha}\sigma \vdash yyy : \rho$, and we may assume that σ has no outermost quantifiers. The latter type judgement (in the simplified system) means that the types σ_1, σ_2 and σ_3 assigned to the three occurrences of y may be assumed to satisfy Conditions (c), (d) and (e), for appropriate $\forall\vec{\delta}\tau$ and π.

The type judgement $\vdash \lambda x.\, z(x1)(x1') : \forall\vec{\beta}\chi \to \eta$ may be satisfied only when it holds that $x : \forall\vec{\beta}\chi \vdash z(x1)(x1') : \eta$, and this implies that Conditions (a) and (b) must hold (types $\forall\vec{\alpha}\sigma^* \to \rho^*$ and $\forall\vec{\alpha}\sigma^\circ \to \rho^\circ$ are assigned to the two occurrences of x). □

In what follows we assume that Conditions (a)–(e) of the above lemma are true, and we are heading towards a contradiction.

Lemma 3.3. There is a variable $\alpha' \in \vec{\alpha}$ such that α' owns in σ a nonempty prefix X_1 of the infinite path $LLL\ldots$

Proof. Clearly, some variable γ must own in σ a prefix L^m of $LLL\ldots$ Suppose that γ is not in $\vec{\alpha}$. It follows from Lemma 2.1 that γ must own the same path L^m in both σ_1 and σ_2. This yields a contradiction since $\sigma_1 = \sigma_2 \to \forall\vec{\delta}\tau$, and thus γ would have to own two different prefixes of $LLL\ldots$ in σ_1: one of length m and one of length $m+1$. Finally,

we observe that X_1 is nonempty, as otherwise $\forall\vec{\alpha}\sigma^*$ must have the form $\forall\vec{\alpha}\,(\alpha'\rho_1\dots\rho_k)$, for some constructors $\rho_1,\dots,\rho_k$, and the latter type cannot be assigned to 1. □

Lemma 3.4. There is a variable $\alpha'' \in \vec{\alpha} \cup \vec{\delta}$ such that α'' owns in σ a prefix X_2 of the infinite path $RLRLRL\dots$.

Proof. Suppose that a variable γ, neither in $\vec{\alpha}$ nor in $\vec{\delta}$, owns in σ a prefix X of $RLRLRL\dots$ First note that $|X| \geqslant 1$, because of the previous lemma, and we must have $X = RX'$, for some X'. By Lemma 2.1, the variable γ must own X' in $\forall\vec{\delta}\,\tau$ and in $\sigma_3 \to \pi$ (see Conditions (d) and (e) of Lemma 3.2). It follows that $X = RLX''$, for some X'', and γ owns X'' in σ_3. A contradiction follows from the fact that γ must also own X in σ_3, by Lemmas 2.1 and 3.2(c). □

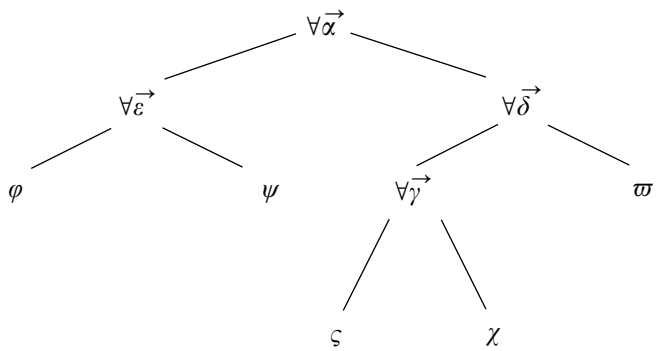
Lemma 3.5. $|X_1| \geqslant 2$ and $|X_2| \geqslant 3$.

Proof. If $X_1 = L$, the type of 1 would be of the form $\forall\vec{\alpha}\,(\forall\vec{\gamma}(\alpha'\rho_1\dots\rho_k) \to \eta)$. Thus, the type $\forall\vec{\gamma}\,(\alpha'\rho_1\dots\rho_k)$ would have to be assigned to f. This is impossible because, with $\alpha' \notin \vec{\gamma}$, the latter cannot be instantiated to a function type. In a similar way (using $1'$ instead of 1) we show that X_2 is neither empty, nor equal to R or RL. □

From the above lemmas we conclude that

$$\sigma \equiv \forall\vec{\varepsilon}\,(\varphi \to \psi) \to \forall\vec{\delta}(\forall\vec{\gamma}\,(\varsigma \to \chi) \to \varpi)$$

for some types φ, ψ, ς, χ, ϖ, and some vectors of variables $\vec{\varepsilon}$ and $\vec{\gamma}$. That is, $\forall\vec{\alpha}\sigma$ is of the following shape:



Let σ'_2 and σ'_3 be obtained from σ_2 and σ_3, respectively, by removing the leading quantifiers. It is natural to use the following notation:

$$\begin{aligned}
\sigma_1 &\equiv \forall\vec{\varepsilon}\,(\varphi_1 \to \psi_1) \to \forall\vec{\delta}(\forall\vec{\gamma}\,(\varsigma_1 \to \chi_1) \to \varpi_1)\\
\sigma'_2 &\equiv \forall\vec{\varepsilon}(\varphi_2 \to \psi_2) \to \forall\vec{\delta}(\forall\vec{\gamma}\,(\varsigma_2 \to \chi_2) \to \varpi_2)\\
\sigma'_3 &\equiv \forall\vec{\varepsilon}(\varphi_3 \to \psi_3) \to \forall\vec{\delta}(\forall\vec{\gamma}\,(\varsigma_3 \to \chi_3) \to \varpi_3)\\
\sigma^* &\equiv \forall\vec{\varepsilon}(\varphi^* \to \psi^*) \to \forall\vec{\delta}(\forall\vec{\gamma}\,(\varsigma^* \to \chi^*) \to \varpi^*)\\
\sigma^\circ &\equiv \forall\vec{\varepsilon}(\varphi^\circ \to \psi^\circ) \to \forall\vec{\delta}(\forall\vec{\gamma}\,(\varsigma^\circ \to \chi^\circ) \to \varpi^\circ).
\end{aligned}$$

Now assume that α' owns the path L^n in φ (that is, it owns L^{n+2} in σ).

Lemma 3.6. The variable α' owns the path L^{n-1} in ς.

Proof. First note that if a variable ξ owns in ς a path L^k, for $k \leqslant n$, then ξ is either in $\vec{\alpha}$ or in $\vec{\delta}$. Indeed, otherwise ξ must own L^k in ς_1. But an appropriate instantiation of $\vec{\delta}$ in ς_1 creates the type $\forall\vec{\varepsilon}\,(\varphi_3 \to \psi_3)$, and thus ξ also owns the path L^{k-1} in φ_3. Since α' owns L^n in φ, there are only arrows and quantifiers at the leftmost path of φ_3 down to the depth n. This implies $k-1 \geqslant n$, that is, $k > n$, which is a contradiction.

Now recall that $\vdash 1 : \forall\vec{\alpha}\sigma^*$ and thus $\vdash 1 : \sigma^*$ also. We must then have

$$f : \forall\vec{\varepsilon}\,(\varphi^* \to \psi^*), u : \forall\vec{\gamma}\,(\varsigma^* \to \chi^*) \vdash fu : \varpi^*.$$

This means that the types $\forall\vec{\varepsilon}\,(\varphi^* \to \psi^*)$ and $\forall\vec{\gamma}\,(\varsigma^* \to \chi^*)$ can be instantiated to some $\psi \to \varphi$ and ψ, respectively. Since α' owns L^{n+1} in $\forall\vec{\varepsilon}\,(\varphi^* \to \psi^*)$, it must also own L^n in ψ. On the other hand, ψ is an instance of $\forall\vec{\gamma}\,(\varsigma^* \to \chi^*)$. Inspecting the first n steps of the path $LLL\ldots$ in $\varsigma^* \to \chi^*$, we find either arrows or an occurrence of some $\alpha''' \in \vec{\alpha} \cup \vec{\delta}$ that owns some L^k in $\varsigma^* \to \chi^*$. If such an α''' is there, it must remain in ψ, and thus we must have $k = n$. In addition, it must be that α''' is just α'. In order to conclude the proof we just need to observe that something must own L^n in $\varsigma^* \to \chi^*$, as otherwise an arrow would occur in ψ instead of α'. □

Proof of Theorem 3.1. Consider the term $1'$ and its type σ°. We proceed in a similar way as in the proof of Lemma 3.6. The following type assignment must hold:

$$v : \forall\vec{\varepsilon}\,(\varphi^\circ \to \psi^\circ), g : \forall\vec{\gamma}\,(\varsigma^\circ \to \chi^\circ) \vdash gv : \varpi^\circ.$$

From Lemma 3.6 we know that α' owns L^n in $\varsigma \to \chi$, and thus also in $\varsigma^\circ \to \chi^\circ$, and in the appropriate instance (type of g). Thus, α' also owns L^{n-1} in an appropriate instance of $\varphi^\circ \to \psi^\circ$ (type of v) and it owns L^{n-2} in φ°. But it cannot occur there, since there are only arrows at the leftmost path in φ up to the depth n. □

4. Coding of a Turing Machine

As we mentioned in the Introduction, the Turing Machine coding used here is borrowed from Henglein and Mairson (1994). Let us briefly restate the main definitions. In what follows, a tuple of terms, say $\langle M_1, \ldots, M_k\rangle$ is identified with the term $\lambda x.\, xM_1 \ldots M_k$, where x is not free in $M_1, \ldots, M_k$. Thus, we have $\langle M_1, \ldots, M_k\rangle pr_i^k \longrightarrow\!\!\!\!\!\rightarrow M_i$, for all $i = 1, \ldots, k$, where $pr_i^k = \lambda z_1, \ldots, z_k.\, z_i$ is the i-th projection on k arguments. An empty tuple, denoted *nil*, is defined as $\lambda x.\, x$. Lists (denoted, for example, by '$[a, b, \ldots, z]$') are defined so that **cons**(a, L) is just $\langle a, L\rangle$. Thus, for instance, $[a, b, c]$ is represented by $\lambda x.\, xa(\lambda y.\, yb(\lambda z.\, zc(\lambda x.\, x)))$.

Elements of finite sets are represented by projections (in particular this applies to *true* and *false* represented by pr_1^2 and pr_2^2), and functions on finite sets are tuples of values: a function f defined on a k-element set $\{a_1, \ldots, a_k\}$ is identified with the term $\langle f(a_1), \ldots, f(a_k)\rangle$, which reduces to $f(a_i)$ when applied to pr_i^k. As we have agreed to

identify a_i with the i-th projection, we may assume that the application fa_i is a well-formed lambda term representing the function value. This applies as well to (curried) functions on 2 arguments: each fa_i is a tuple of the form $\langle f(a_i, b_1), \ldots, f(a_i, b_\ell)\rangle$, where $b_1, \ldots, b_\ell$ are the possible values for the second argument.

The machinery described above provides the means to represent computations of Turing Machines by lambda terms. Our machine has a single tape infinite to the right. An input value N is represented as a string of l's of length N, preceded by a '$' to mark the beginning of the tape and followed by blank symbols. The output might be handled similarly, but for technical reasons it is convenient to assume that the machine contains an *output register* capable of storing an integer, initially zero, which may be incremented by 1 but may not be decremented.

Another modification we make here may seem quite ridiculous at first glance, but it proves to be surprisingly useful from the technical point of view and lets us avoid a lot of extra work. In addition, it is useful methodologically, because it allows us to eliminate applications of higher-order polymorphism that are not relevant to the core 'crank' of the proof. (In earlier versions of this paper, higher-order polymorphism was also used for secondary purposes.) This idea, suggested to the author by Joe Wells, is as follows: in addition to the integer output register, we will also have a Boolean *stop register*, initially false. The computation ends when this value becomes true.

That is, the transition function D applied to a pair $\langle q, b\rangle$ (state and tape symbol) returns a quintuple $\langle p, c, v, \Delta, h\rangle$, consisting of a new state, a new tape symbol, a head movement (*left*, *stay* or *right*), an output increment $\Delta \in \{0, 1\}$, and a stop flag. It is a routine exercise to check that each partial recursive function is computed by such a machine.

In what follows, we assume that such a deterministic Turing Machine is fixed, together with an initial value N, which determines an initial configuration C^0, and that the machine converges on C^0 in exactly M steps. Following Henglein and Mairson (1994), we assume without loss of generality that our machine satisfies certain conditions (some of them quite peculiar, but technically useful):

— There are m internal states, denoted $\mathtt{s}_1, \ldots, \mathtt{s}_m$ (with no need for a final state).
— There are r tape symbols, denoted $\mathtt{d}_1, \ldots, \mathtt{d}_r$, including the blank symbol $\mathtt{d}_r$, and the symbols 'l' and '$' necessary to code the input.
— The machine never attempts to move left, when reading the leftmost tape cell.
— It never writes a blank.
— It does not move *if and only if* it reads a blank.

Note that a consequence of our assumption about no final state being chosen is that the transition function D must be total. With the ordinary conventions about the input, we may assume that the tape is always of the form '$a_1 a_2 \ldots a_n$ *blank blank* ... ', where all the a_i's are different from *blank*. If the current state is $\mathtt{s}_j$ and the tape head is positioned at the k-th cell (note that $k \leqslant n + 1$), the resulting configuration is identified with the term

$$\langle \mathtt{s}_j, [a_{k-1}, a_{k-2}, \ldots, a_1, \mathit{blank}], [a_k, \ldots, a_n, \mathit{blank}], \mathit{output}, \mathit{stop}\rangle,$$

where '*output*' (respectively, '*stop*') is the Church numeral (respectively, Boolean) representing the contents of the output (respectively, stop) register. Of course states and tape

symbols are identified with the appropriate projections. The extra blank at the end of the first list is added to ensure that the code of the left-hand part of the tape is always a pair, not *nil*, as in the initial configuration:

$$\langle \mathsf{s}_1, [\mathit{blank}], [\$, \mathsf{I}, \mathsf{I}, \ldots, \mathsf{I}, \mathit{blank}], 0, \mathit{false}\rangle.$$

The transition function D is coded as a tuple of tuples, so that, for $i = 1, \ldots, m$ and $j = 1, \ldots, r$, we have $D\mathsf{s}_i\mathsf{d}_j = \langle p_{i,j}, c_{i,j}, v_{i,j}, \Delta_{i,j}, h_{i,j}\rangle$, where $p_{i,j}$, $c_{i,j}$, $v_{i,j}$, $\Delta_{i,j}$ and $h_{i,j}$ are (projections corresponding to) the new state, the symbol to be written, the move to be performed, the output increment and the Boolean stop flag. (A 'move' is either $\mathit{left} = pr^3_1$ or $\mathit{stay} = pr^3_2$ or $\mathit{right} = pr^3_3$. The three possible moves will also be denoted by mv_1, mv_2 and mv_3.) Note that $\Delta_{i,j}$ is not a Church numeral but a projection. The 'next ID' function *Next* is now defined as a lambda-term that can be (informally) written as follows:

$$\begin{aligned}
\mathit{Next} \equiv \lambda C.\quad & \textbf{assume } C = \langle q, \langle a, L\rangle, \langle b, R\rangle, k, g\rangle \\
\textbf{in}\quad & \textbf{assume } Dqb = \langle p, c, v, \Delta, h\rangle \\
\textbf{in}\quad & \textbf{let } y = \Delta k(\mathit{succ}\, k) \\
\textbf{in}\quad & \textbf{let } F = \langle p, L, \langle a, \langle c, R\rangle\rangle, y, h\rangle \\
\textbf{in}\quad & \textbf{let } S = \langle p, \langle a, L\rangle, \langle c, \langle \mathit{blank}, \mathit{nil}\rangle\rangle, y, h\rangle \\
\textbf{in}\quad & \textbf{let } T = \langle p, \langle c, \langle a, L\rangle\rangle, R, y, h\rangle \\
\textbf{in}\quad & vFST.
\end{aligned}$$

The symbol *succ* stands for the ordinary representation of the successor function on the Church numerals: $\lambda n\lambda fx.\, f(nfx)$. The **let**'s and **assume**'s are abbreviations the meaning of which should be obvious. For instance, the expression '**assume** $Dqb = \langle p, c, v, \Delta, h\rangle$ **in** ...' should read

$$'(\lambda pcv\Delta h \ldots)(Dqb\, pr^5_1)(Dqb\, pr^5_2)(Dqb\, pr^5_3)(Dqb\, pr^5_4)(Dqb\, pr^5_5)'.$$

It is a routine, though time-consuming, exercise to check that our function *Next* is correct. That is, if C is (a code of) a legal ID, then *Next* C is equal (with respect to beta-conversion) to (the code of) the next ID.

4.1. *Coding of a computation*

The precise definition of our term W, informally announced in the Introduction, is as follows:

$$W \equiv \lambda x.\, xpr^5_5\, (\lambda w.\, xpr^5_4)(\lambda w.\, w(\mathit{Next}\ x)w). \tag{4.1}$$

Here, the expression xpr^5_5 stands for the final state test, in the following sense: if C is a quintuple with a Boolean (a 2-argument projection) as the fifth coordinate (in particular when C represents a machine ID), then WC evaluates either to $\lambda w.\, xpr^5_4$ or to $\lambda w.\, w(\mathit{Next}\ C)w$. If we now inspect the behaviour of the term WC^0W, where C^0 is the initial configuration, we find the following sequence of reductions:

$$WC^0W \twoheadrightarrow WC^1W \twoheadrightarrow WC^2W \twoheadrightarrow \cdots,$$

where $C^1, C^2, \ldots$ is the resulting sequence of configurations. If our Turing Machine diverged on C^0, this sequence would be infinite, and thus our term could not be typable.

We want to show that otherwise it can be typed, and thus we have assumed above that the machine converges in exactly M steps. In this case the reduction sequence is finite, and its result is the output numeral (the fourth component of C^M):

$$WC^0W \twoheadrightarrow WC^1W \to \cdots \to WC^{M-1}W \twoheadrightarrow WC^MW \twoheadrightarrow C^M pr_4^5.$$

In fact, typability of WC^0W is not enough. What we want is that the function computed by our machine is nonuniformly representable in $\mathbf{F}_\omega$. Thus, the initial configuration C^0 must be written as a function of the input numeral N. This can be done as follows. Let $Step = \lambda L.\langle \mathsf{I}, L\rangle$, and let

$$Base = \lambda x.\langle \mathsf{s}_1, [blank], \langle \$, xStep\,[blank]\rangle, 0, false\rangle.$$

One can see that $BaseN$ reduces to the initial configuration

$$C^0 = \langle \mathsf{s}_1, [blank], [\$, \mathsf{I}, \mathsf{I}, \ldots, \mathsf{I}, blank], 0, false\rangle,$$

with N occurrences of 'I' on the tape. One can now consider the following term

$$T \equiv (\lambda w.\, w(BaseN)w)W.$$

Of course, we have $T \twoheadrightarrow (\lambda w.\, wC^0w)W \twoheadrightarrow WC^0W$. Our main result is a consequence of the fact that T is typable in $\mathbf{F}_\omega$, and this is our main goal. In the following two subsections we make the necessary preparation for the main construction of Section 5. That is, we show how to assign certain types to the functions *Next* and *Base*, and to the machine configurations $C^0, \ldots, C^M$. These types have free variables, and in Section 5 we will use their particular substitution instances.

4.2. *Typing the function Next*

The techniques of Kanellakis *et al.* (1991) and Henglein and Mairson (1994) allow us to derive a quantifier-free (or rather universally polymorphic) type for the function *Next*, so that expressions of the form $Next(Next(\cdots Next(C^0)\cdots))$ become well typed. In fact, one can do more with this typing: Henglein and Mairson (1994) provides types for terms of the form $222\ldots2Next\,C^0$ (compare this with the example typing of $222\ldots2K$ in Section 2). Unfortunately, for our purposes this will not work, as we cannot avoid inside quantifiers. For this reason, and to make the paper more self-contained, we repeat the construction here, with the necessary modifications.

We begin with a few definitions. As we make an extensive use of tuples, it is natural to introduce product notation '$\tau_1 \times \cdots \times \tau_\ell$' to abbreviate $\forall\alpha((\tau_1 \to \cdots \to \tau_\ell \to \alpha) \to \alpha)$ with α not free in $\tau_1, \ldots \tau_\ell$. A product type as above will typically be assigned to a tuple $\langle M_1, \ldots, M_\ell\rangle$, where $M_i : \tau_i$ holds for all $i = 1, \ldots, \ell$. Since we cannot avoid quantifiers anyway, we sometimes take the liberty of using product types for more transparency. However, product types (and quantified types in general) cannot be used everywhere, and in some cases we prefer to use an open type $(\tau_1 \to \cdots \to \tau_\ell \to \alpha) \to \alpha$. The most important property of product types is as follows.

Lemma 4.2. If $E \vdash \langle M_1, \ldots, M_\ell \rangle : (\tau_1 \to \cdots \to \tau_\ell \to \alpha) \to \alpha$ (in particular if $E \vdash \langle M_1, \ldots, M_\ell \rangle : \tau_1 \times \cdots \times \tau_\ell$), then $E \vdash M_i : \tau_i$, for all $i = 1, \ldots, \ell$.

Proof. This is a consequence of Lemma 2.2. The details are left to the reader. □

Of course, a projection pr_i^k is assigned, whenever possible, a type of the form $\tau_1 \to \cdots \to \tau_k \to \tau_i$. Such a type is called an *ordinary projection type*. Similarly, the most convenient type for a list of length n of elements of a k-element set has the form $\rho_1 \times (\rho_2 \times \cdots (\rho_n \times \mathbf{Nil}) \cdots)$, where $\mathbf{Nil} = \forall\alpha(\alpha \to \alpha)$ is a straightforward type for *nil*, and $\rho_1, \rho_2, \ldots, \rho_n$ are ordinary projection types (with k arguments). Such types are called *ordinary list types*.

In order to type the function *Next*, we must be able to type the transition function D, that is, we need types for all its components. What we define are open types with free variables to be substituted depending on the context. Let $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, r\}$. The type for the expression $D\mathsf{s}_i\mathsf{d}_j = \langle p_{i,j}, c_{i,j}, v_{i,j}, \Delta_{i,j}, h_{i,j} \rangle$ has the form

$$[D\mathsf{s}_i\mathsf{d}_j] = [p_{i,j}] \times [c_{i,j}] \times [v_{i,j}] \times (\omega \to \omega \to \omega) \times [h_{i,j}].$$

Here the product components are obviously meant to be assigned to the appropriate elements of the quintuple. Assume that $p_{i,j} = \mathsf{s}_n$, $c_{i,j} = \mathsf{d}_k$ and $v_{i,j} = mv_\ell$, for some $n \in \{1, \ldots, m\}$, some $k \in \{1, \ldots, r\}$ and some $\ell \in \{1, 2, 3\}$.

The output increment component $\Delta_{i,j}$ is a projection on two arguments of type ω, and is assigned the type $\omega \to \omega \to \omega$. Other components are projections too, and we define $[c_{i,j}]$ as an open type $\zeta_1^c \to \cdots \to \zeta_r^c \to \zeta_k^c$, where the ζ's are fresh type variables (one can take the same ζ's for all i, j). These free variables may later be substituted by different types, for different applications of the function *Next*. Similarly, we define $[p_{i,j}] = \zeta_1^p \to \cdots \to \zeta_m^p \to \zeta_n^p$ and $[v_{i,j}] = \zeta_1^v \to \zeta_2^v \to \zeta_3^v \to \zeta_\ell^v$.

Type $[h_{i,j}]$, assigned to the last component, is a little more complex. As we can see from (4.1), the projection $h_{i,j}$ will eventually be used to choose between two possibilities: one is the term $(\lambda w.\, xpr_4^5)$, representing termination, the other is the term $(\lambda w.\, w(Next\ x)w)$, representing another computation step. First, let $\mathbf{Quit} = \forall\alpha(\alpha \to \omega)$, and we have $\lambda w.\, xpr_4^5 : \mathbf{Quit}$, provided $xpr_4^5 : \omega$. We define $[h_{i,j}]$ as either $\mathbf{Quit} \to (\eta \to \omega) \to (\nu \to \omega)$ or $\mathbf{Quit} \to (\eta \to \omega) \to (\eta \to \omega)$, depending on whether $h_{i,j}$ is *true* or *false*. Here η and ν are fresh type variables, the same for all i, j. Of course $[h_{i,j}]$ is a correct type for $h_{i,j}$, and we may conclude that $D\mathsf{s}_i\mathsf{d}_j : [D\mathsf{s}_i\mathsf{d}_j]$ holds in the empty environment, for each i, j. Note that the variable η is to be later instantiated in such a way that $\eta \to \omega$ becomes a type of $(\lambda w.\, w(Next\ x)w)$. Similarly, the variable ν will be substituted by a type assigned to a copy of the term W.

The next step is to define a type for $D\mathsf{s}_i$. It may seem appropriate to choose here the product $[D\mathsf{s}_i\mathsf{d}_1] \times \cdots \times [D\mathsf{s}_i\mathsf{d}_j]$, but it will be seen later to be more convenient to take $[D\mathsf{s}_i] := ([D\mathsf{s}_i\mathsf{d}_1] \to \cdots \to [D\mathsf{s}_i\mathsf{d}_r] \to \delta) \to \delta$, with a fresh δ (the same δ for all i will do). Of course, we have $D\mathsf{s}_i : [D\mathsf{s}_i]$ in the empty environment, and it follows that D has type $[D] := [D\mathsf{s}_1] \times \cdots \times [D\mathsf{s}_m]$. Let us note here that the free variables of each $[D\mathsf{s}_i]$ and of $[D]$ are all the ζ's, η, ν and δ.

The types that we may assign to machine ID's will be seen as substitution instances of

the following product:

$$\mathbf{Config} = \mathbf{State} \times (\mathbf{a} \times \mathbf{L}) \times (\mathbf{b} \times \mathbf{R}) \times \omega \times \mathbf{Halt},$$

where **a**, **L**, **b** and **R** are type variables and **Halt** abbreviates the type $\mathbf{Quit} \to (\xi \to \omega) \to (\beta \to \omega)$, with ξ and β — type variables. Of course, the intended meaning of **Halt** is similar to that of $[h_{i,j}]$. The difference is that **Halt** is a common 'pattern' for types assigned to the stop register which may be either true or false. The first component **State** is a pattern of a type for the state component (although it will *not* itself be a type for a projection). If we now look at the definition of *Next*, we will see that the following is a reasonable definition for **State**:

$$\mathbf{State} = [D\mathsf{s}_1] \to \cdots \to [D\mathsf{s}_m] \to \mathbf{b} \to \mathbf{Tuple},$$

where **Tuple** is a type for $\langle p, c, v, \Delta, h\rangle$ (recall that $D = \lambda q.q(D\mathsf{s}_1)\ldots(D\mathsf{s}_m)$ and thus Dqb equals $q(D\mathsf{s}_1)\ldots(D\mathsf{s}_m)b$). Note that if we had quantified δ in the $[D\mathsf{s}_i]$'s, we would never be able to turn **State** into an ordinary projection type by a substitution. The next step is to define

$$\mathbf{Tuple} = \mathbf{p} \times \mathbf{c} \times ([F] \to [S] \to [T] \to \mathbf{out}) \times (\omega \to \omega \to \omega) \times \mathbf{h},$$

where **p**, **c**, **h** and **out** are type variables, and $[F]$, $[S]$ and $[T]$ are defined as follows:

— $[F] = \mathbf{p} \times \mathbf{L} \times (\mathbf{a} \times (\mathbf{c} \times \mathbf{R})) \times \omega \times \mathbf{h}$;
— $[S] = \mathbf{p} \times (\mathbf{a} \times \mathbf{L}) \times (\mathbf{c} \times (\mathbf{Blank} \times \mathbf{Nil})) \times \omega \times \mathbf{h}$;
— $[T] = \mathbf{p} \times (\mathbf{c} \times (\mathbf{a} \times \mathbf{L})) \times \mathbf{R} \times \omega \times \mathbf{h}$.

Here $\mathbf{Blank} = \varepsilon_1 \to \cdots \to \varepsilon_r \to \varepsilon_r$, with fresh type variables $\varepsilon_1, \ldots, \varepsilon_r$ (abbreviated $\vec{\varepsilon}$). One can easily guess that these types are to be assigned to the **let**-bound F, S and T in the definition of *Next*. As we have completed the definition of **Config**, let us collect all of its free variables: we have $FV(\mathbf{Config}) = FV([D]) \cup \{\vec{\varepsilon}, \mathbf{a}, \mathbf{L}, \mathbf{b}, \mathbf{R}, \mathbf{p}, \mathbf{c}, \mathbf{h}, \mathbf{out}, \xi, \beta\}$.

Now we claim that the function *Next* has the following type (and of course all types that are substitution instances of this pattern):

$$\mathit{Next} : \mathbf{Config} \to \mathbf{out}. \tag{4.3}$$

Clearly, the argument type is quite a bit more complicated than the result type: the variable **out**, which is meant to represent the type of the next ID, is free in **State**. This means a type of the result ID must be itself a proper part of the type of its predecessor.

In order to see that the above typing is correct, let us assume $C : \mathbf{Config}$. Since **Config** is a product type, we may assign its type components to the appropriate projections of C, that is, we may assume that we work in an environment containing $q : \mathbf{State}$, $a : \mathbf{a}$, $L : \mathbf{L}$, $b : \mathbf{b}$, $R : \mathbf{R}$, $g : \mathbf{Halt}$ and $k : \omega$. We know that $D : [D\mathsf{s}_1] \times \cdots \times [D\mathsf{s}_m]$, and this is equal to $\forall\alpha(([D\mathsf{s}_1] \to \cdots \to [D\mathsf{s}_m] \to \alpha) \to \alpha)$. Instantiating α by $\mathbf{b} \to \mathbf{Tuple}$ allows us to derive the typing $Dq : \mathbf{b} \to \mathbf{Tuple}$. It follows that $Dqb : \mathbf{Tuple}$, and this immediately implies $p : \mathbf{p}$, $c : \mathbf{c}$, $v : [F] \to [S] \to [T] \to \mathbf{out}$, $\Delta : \omega \to \omega \to \omega$ and $h : \mathbf{h}$. The latter gives $y : \omega$, and this allows us to derive $F : [F]$, $S : [S]$ and $T : [T]$ (since of course $\mathit{blank} : \mathbf{Blank}$), and finally to get $vFST : \mathbf{out}$, which is exactly what we need.

4.3. *Types for configurations*

We have already assumed above that our machine converges in M steps when started in the initial configuration C^0. To be more specific, let $C^0, \ldots, C^M$ be all the ID's occurring in the computation, with C^M — a final configuration, and let C^{M+1} be the 'after final' configuration: the ID obtained by applying a machine step to C^M. (This makes sense because the transition function of our machine is total.) The next stage in the construction is to assign some particular types $[C^n]$, for $n = 0, \ldots, M+1$, to the configurations C^n. As we said before, some particular substitution instances of these types will be needed in Section 5. Now we develop the construction only to the point when we can correctly assign $C^n : [C^n]$.

Assume that, for all n, we have $C^n = \langle q^n, \langle a^n, L^n\rangle, \langle b^n, R^n\rangle, k^n, g^n\rangle$. As can be easily guessed, each $[C^n]$ is defined as a product of the form

$$[C^n] = [q^n] \times ([a^n] \times [L^n]) \times ([b^n] \times [R^n]) \times \omega \times [g^n], \tag{4.4}$$

where the five components are to be assigned to the current state, to the left and right part of the tape and to the output and stop registers. Types $[C^n]$ are defined by a downward induction on n. We begin with a definition of $[C^{M+1}]$ and first take the last component $[g^{M+1}]$ to be $\mathbf{Quit} \to (\forall\alpha\alpha \to \omega) \to (\forall\alpha\alpha \to \omega)$, independently of the value of g^{M+1}. The other components of $[C^{M+1}]$ are chosen to be as simple as possible: we want just $C^{M+1} : [C^{M+1}]$, and thus we just assign '$\forall\alpha\alpha \to \cdots \to \forall\alpha\alpha \to \forall\alpha\alpha$' to all projections occurring in the term C^{M+1}. For $n \leqslant M$, we need more: we want $[C^n]$ to be a substitution instance of **Config**. Two of the free variables of **Config**, namely ξ and β, occurring free only in **Halt**, the last coordinate of **Config**, play a special role in our construction. Thus, it is convenient to present $[C^n]$ as:

$$[C^n] = \mathbf{Config}\{\mathscr{S}^n\}\{\xi^n/\xi,\ \tau^n/\beta\}, \tag{4.5}$$

where ξ^n is a fresh type variable, τ^n is defined below, and S^n stands for a simultaneous substitution of certain types to all other free variables in **Config**. It will sometimes be convenient to use the following notation: $\mathscr{S}^n(\mathbf{p}) = [p^n]$, $\mathscr{S}^n(\mathbf{c}) = [c^n]$, $\mathscr{S}^n(\mathbf{h}) = [h^n]$, $\mathscr{S}^n(\mathbf{out}) = \mathbf{Out}^n$, so that one can write:

$$\mathscr{S}^n = \{[a^n]/\mathbf{a},\ [L^n]/\mathbf{L},\ [b^n]/\mathbf{b},\ \ [R^n]/\mathbf{R}, [p^n]/\mathbf{p},\ [c^n]/\mathbf{c},\ [h^n]/\mathbf{h},\ \mathbf{Out}^n/\mathbf{out}, \ldots\}.$$

In what follows, we define $\mathscr{S}^n$ and τ^n, by a backward induction, in such a way that the following conditions hold:

(i) $C^n : [C^n]$;
(ii) types $[a^n]$, $[b^n]$ and $[q^n]$, for all n, and types $[p^n]$, $[c^n]$, for $n \leqslant M$, are ordinary projection types;
(iii) types $[L^n]$ and $[R^n]$ are ordinary list types (*i.e.*, their components are ordinary projection types);
(iv) $\tau^M = \kappa$, and $\tau^n = \xi^n$, for $n < M$, where κ is another fresh type variable;
(v) the only variables that occur free in $[C^n]$ are κ and ξ^k, for $k \geqslant n$, except that $[C^{M+1}]$ is closed.

The reader will easily observe that Condition (i) and Lemma 4.2 imply that, for example,

$L^n : [L^n]$. Together with Condition (iii), this guarantees that the components of types L^n and R^n correspond to appropriate tape symbols in C^n.

Assume that $[C^{n+1}]$ has been constructed already so that (with $n+1$ replacing n) it has the form of (4.4) and (4.5), and that the induction hypothesis, consisting of Conditions (i)–(v), is satisfied. In addition, assume that $Dq^nb^n = \langle q^{n+1}, c^n, v^n, \Delta^n, h^n\rangle$, for some c^n, v^n, Δ^n, and h^n. We begin the construction of $[C^n]$ with the easy part: we define $\mathbf{Out}^n := [C^{n+1}]$, $[h^n] := [g^{n+1}]$ and $[p^n] := [q^{n+1}]$. The definition of τ^n is determined by the induction hypothesis, part (iv). Types $[a^n]$, $[c^n]$, $[L^n]$, and $[R^n]$ are defined by cases depending on the value of v^n (here, $head(\tau \times \sigma)$ denotes τ and $tail(\tau \times \sigma)$ denotes σ):

Case 1: if $v^n = left$, then $[a^n] = [b^{n+1}]$, $[c^n] = head\,[R^{n+1}]$, $[L^n] = [a^{n+1}] \times [L^{n+1}]$, and $[R^n] = tail\,[R^{n+1}]$.

Case 2: if $v^n = stay$, then $[a^n] = [a^{n+1}]$, $[c^n] = [b^{n+1}]$, $[L^n] = [L^{n+1}]$, and $[R^n] = \mathbf{Nil}$.

Case 3: if $v^n = right$, then $[a^n] = head\,[L^{n+1}]$, $[c^n] = [a^{n+1}]$, $[L^n] = tail\,[L^{n+1}]$, and $[R^n] = [b^{n+1}] \times [R^{n+1}]$.

The applications of *tail* and *head* in the above definition are correct, because of the induction hypothesis $C^{n+1} : [C^{n+1}]$ and because the list L^{n+1} must be nonempty after a right move, as well as R^{n+1} must be nonempty after a left move.

The value of v^n also determines our definition of $\mathscr{S}^n(\varepsilon_i)$. If $v^n = stay$, it must be the case that R^{n+1} begins with a blank. Thus, $head\,[R^{n+1}]$ must have the form $\rho_1 \to \cdots \to \rho_r \to \rho_r$ (recall it must be an ordinary projection type). We take $\mathscr{S}^n(\varepsilon_j) = \rho_j$, for all j. If $v^n \neq stay$, types $\mathscr{S}^n(\varepsilon_j)$ will not matter at all and we may take, for example, $\mathscr{S}^n(\varepsilon_j) := \forall\alpha\alpha$.

Now suppose that $q^n = \mathsf{s}_x$ and $b^n = \mathsf{d}_y$. In order to ensure that $[q^n]$ is an ordinary type for the x-th projection, we must define $\mathscr{S}^n$ so that we can have the equality

$$[D\mathsf{s}_x]\{\mathscr{S}^n\} = [b^n] \to \mathbf{Tuple}\{\mathscr{S}^n\}. \tag{4.6}$$

Indeed, we have $[q^n] = \mathbf{State}\{\mathscr{S}^n\} = [D\mathsf{s}_1]\{\mathscr{S}^n\} \to \cdots \to [D\mathsf{s}_m]\{\mathscr{S}^n\} \to [b^n] \to \mathbf{Tuple}\{\mathscr{S}^n\}$. Since $[D\mathsf{s}_x] = ([D\mathsf{s}_x\mathsf{d}_1] \to \cdots \to [D\mathsf{s}_x\mathsf{d}_r] \to \delta) \to \delta$, we must define

$$[b^n] := [D\mathsf{s}_x\mathsf{d}_1]\{\mathscr{S}^n\} \to \cdots \to [D\mathsf{s}_x\mathsf{d}_r]\{\mathscr{S}^n\} \to \mathbf{Tuple}\{\mathscr{S}^n\},$$

and we also must take $\mathscr{S}^n(\delta) = \mathbf{Tuple}\{\mathscr{S}^n\}$. There is no circularity in these definitions, because $\mathbf{b}$ is neither free in $[D\mathsf{s}_x]$ nor in **Tuple**, and δ is not free in **Tuple**. But $[b^n]$ must also be an ordinary type for the y-th projection, which amounts to the equality

$$[D\mathsf{s}_x\mathsf{d}_y]\{\mathscr{S}^n\} = \mathbf{Tuple}\{\mathscr{S}^n\}.$$

Let us recall that the left-hand side of this equation is obtained from

$$[D\mathsf{s}_x\mathsf{d}_y] = [p_{x,y}] \times [c_{x,y}] \times [v_{x,y}] \times (\omega \to \omega \to \omega) \times [h_{x,y}],$$

while the right-hand side is:

$$[p^n] \times [c^n] \times ([F]\{\mathscr{S}^n\} \to [S]\{\mathscr{S}^n\} \to [T]\{\mathscr{S}^n\} \to \mathbf{Out}^n) \times (\omega \to \omega \to \omega) \times [h^n].$$

If we look at the appropriate definitions, we find that $[h^n] = [g^{n+1}] = \mathbf{Quit} \to (\xi^{n+1} \to \omega) \to (\tau^{n+1} \to \omega)$, and that $[h_{i,j}]$ is either $\mathbf{Quit} \to (\eta \to \omega) \to (v \to \omega)$ or $\mathbf{Quit} \to (\eta \to \omega) \to (\eta \to \omega)$. Thus the last components of the above two products become identical

after substituting ξ^{n+1} for η, and κ for ν (except that when $n = M$ we substitute $\forall\alpha\alpha$ for both η and ν). In order to match $[p_{x,y}]$ with $[p^n]$ we note that the latter is an ordinary projection type of the correct projection. The same holds for $[c_{x,y}]$ versus $[c^n]$. The essential part is thus unifying the third components, and (since $\mathbf{Out}^n = [C^{n+1}]$) this amounts to checking whether $[C^{n+1}]$ is equal to either $[F]\{\mathscr{S}^n\}$, $[S]\{\mathscr{S}^n\}$ or $[T]\{\mathscr{S}^n\}$, depending on whether v^n is *left*, *stay* or *right*. This boring task is left entirely to the reader.

Note that the occurrences of δ outside $[D\mathsf{s}_x]$ and occurrences of other variables in $FV([D])$ outside $[D\mathsf{s}_x\mathsf{d}_y]$ are not relevant for the typing $C^n : [C^n]$. That is why we could choose to have only one set of ζ's for all i, j, and only one δ, η and ν.

We are now faced with the task of proving that the induction hypothesis holds for n. For $n = M + 1$ the conditions are obviously satisfied, so we assume $n \leqslant M$. To verify (i), recall that $C^n = \langle q^n, \langle a^n, L^n\rangle, \langle b^n, R^n\rangle, k^n, g^n\rangle$ and that $[C^n] = [q^n] \times ([a^n] \times [L^n]) \times ([b^n] \times [R^n]) \times \omega \times [g^n]$. From the induction hypothesis for $n+1$, and from the definitions above, we easily obtain $a^n : [a^n]$, $b^n : [b^n]$, $L^n : [L^n]$ $R^n : [R^n]$, $k^n : \omega$ and $g^n : [g^n]$. Thus it remains to check that $q^n : [q^n]$, which follows from Condition (4.6), and we are done.

Parts (ii)–(iv) are obvious, and (v) follows by a simple induction. Note that κ occurs in $[C^M]$ because $\tau^M = \kappa$, but it does not occur free in $\mathbf{Out}^M = [C^{M+1}]$. But for $n < M$, we have κ free in $\mathbf{Out}^n = [C^{n+1}]$, because the latter type contains a copy of $[C^M]$.

4.4. *Input handling*

In the above construction we have assumed a fixed computation of our machine, determined by a fixed input N. The next step is to show how to type the operation *Base*, which makes an initial configuration out of the input numeral. Recall that

$$Base = \lambda x.\langle \mathsf{s}_1, [blank], \langle \$, xStep\,[blank]\rangle, 0, false\rangle,$$

where $Step = \lambda L.\langle \mathsf{l}, L\rangle$. We want to show that the expression $BaseN$ has type $[C^0]$. To see this, observe that

$$[C^0] = [q^0] \times ([a^0] \times \mathbf{Nil}) \times ([b^0] \times (\phi_1 \times \cdots \times (\phi_N \times (\rho \times \mathbf{Nil})))) \times \omega \times [g^0],$$

where $[a^0]$ and ρ are ordinary projection types of *blank*, $[b^0]$ is an ordinary projection type of \$, and all the ϕ's are ordinary projection types of l. Assuming that l is, for example, d_1, each of the latter types is of the form $\phi_j = \rho_1^j \to \rho_2^j \to \cdots \to \rho_r^j \to \rho_1^j$, for some $\rho_1^j, \ldots, \rho_r^j$. Let $\vec{\alpha}$ denote the sequence of variables $\alpha_1, \ldots, \alpha_r$. We can assign the following type to the number N:

$$[N] = \forall\beta'\,(\forall\beta(\beta \to \forall\vec{\alpha}\,(\phi_0 \times \beta)) \to \beta' \to \phi_1 \times (\cdots \times (\phi_N \times \beta')\cdots)),$$

where $\phi_0 = \alpha_1 \to \alpha_2 \to \cdots \to \alpha_r \to \alpha_1$. Indeed, for f of type $\forall\beta\,(\beta \to \forall\vec{\alpha}\,(\phi_0 \times \beta))$ and x of type β', one obtains $fx : \forall\vec{\alpha}\,(\phi_0 \times \beta')$. Using an instantiation, we get $fx : \phi_N \times \beta'$. The reader is invited to verify that $f(fx) : \phi_{N-1} \times (\phi_N \times \beta)$, and so on, finally resulting in $f^N(x) : \phi_1 \times (\cdots \times (\phi_N \times \beta')\cdots)$, as desired.

Since $Step : \forall\beta\,(\beta \to \forall\vec{\alpha}\,(\phi_0 \times \beta))$, we can easily see that

$$Base : [N] \to [C^0],$$

but we have to remember that $[C^0]$ depends on N. This typing is of course different for different values of N, and this is the place where the nonuniformity in function representation is essential.

5. Typing the computation

Let us recall from Section 4.1 that our main task is to show typability of the term

$$T \equiv (\lambda w.\, w(BaseN)w)W,$$

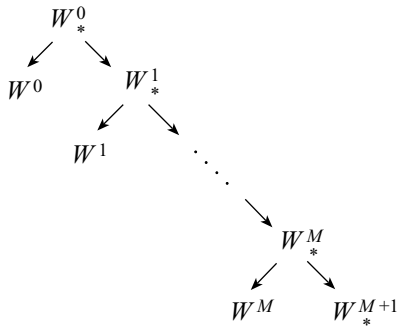
where N is a fixed input (on which our Turing Machine terminates in M steps) and where

$$W \equiv \lambda x.\, xpr_5^5\,(\lambda w.\, xpr_4^5)(\lambda w.\, w(Next\ x)w)$$

is the main tool for the Turing Machine simulation. As we have already observed, if C represents a machine ID, then WC evaluates either to $\lambda w.\, xpr_4^5$ or to $\lambda w.\, w(Next\ C)w$. For a given integer N, the term $BaseN$ of the previous section reduces to a representation of the initial configuration C^0 with the input value N. Thus, our term T reduces to $(\lambda w.\, wC^0w)W$, and the latter can be further reduced as described below (where we use indices to distinguish between different copies of W):

$$\begin{aligned}(\lambda w.\, wC^0w)W_*^0 &\twoheadrightarrow W^0C^0W_*^1 \twoheadrightarrow\ W^1C^1W_*^2 \to \cdots\\ \cdots &\to W^{M-1}C^{M-1}W_*^M \twoheadrightarrow\ W^MC^MW_*^{M+1} \twoheadrightarrow C^Mpr_4^5.\end{aligned}$$

The fourth component of C^M, which is obtained from $C^Mpr_4^5$, is the output numeral. There are two kinds of copies of W here: the 'final' (nonstarred) copies W^n, and the 'general' (starred) copies, which are still to split, according to the following scheme:



Our goal is to construct a type for W_*^0, so that T is well typed. For this, we will have to define closed types $[W_*^n]$ for all the W_*^n's, and closed types $[W^n]$ to be assigned to the W^n's ($n = 0, \ldots, M$). The definition goes by a backward induction, starting from $n = M$ and ending with $n = 0$. (Type $[W_*^{M+1}]$ may be the same as $[W_*^M]$.)

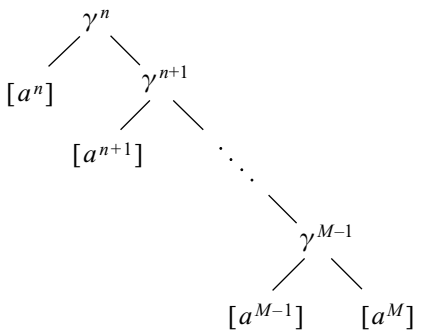
Let us make a digression here, which may perhaps help in seeing the basic idea of our construction. Each of the types $[W^n]$ will be assigned to a copy of W applied to C^n. Thus, what we want, is that

$$[W^n] = [x^n] \to Z^n \to \omega,$$

where the argument part $[x^n]$ of $[W^n]$ is a substitution instance of $[C^n]$ and Z^n is a type that may be derived for W_*^{n+1}. (For technical reasons, types Z^n and $[W_*^{n+1}]$ will not be identical, but they will differ very little, namely $[W_*^{n+1}] \preceq Z^n$ and $Z^n \preceq [W_*^{n+1}]$ will hold.) Types $[W_*^n]$ must be sufficiently polymorphic to allow for all the $[W_k]$'s, for $k \geqslant n$, to be obtained by instantiations. The idea is that the type substituted *e.g.* for **a** in the argument part of $[W_*^n]$, will be of the form

$$[a_*^n] := \gamma^n[a^n](\gamma^{n+1}[a^{n+1}](\cdots(\gamma^{M-1}[a^{M-1}][a^M])\cdots)),$$

where the types $[a^k]$ are to be substituted for **a** in $[W_k]$ and the γ's are binary constructor variables, to be used as 'switches'. These 'switches' are universally quantified and instantiated with the *left projection* $\pi_L = \lambda\alpha_1\alpha_2.\,\alpha_1$ or the *right projection* $\pi_R = \lambda\alpha_1\alpha_2.\,\alpha_2$. Type $[a_*^n]$ is best presented as the following 'ladder', which very much resembles the relationships between all the copies of W:



Note that instantiating γ_n with π_L turns the above type into $[a^n]$, while using π_R gives another, shorter 'ladder' for $[a_*^{n+1}]$. Similar 'ladders' will occur in $[W_*^n]$ at all positions where there are differences between the $[W^k]$'s for $k \geqslant n$.

Before we actually begin our inductive definition, we need some technical preparation. In the construction to follow, β stands for a fixed type variable (intentionally the same as β in **Config**), and $\vec{\gamma} = \gamma^0, \ldots, \gamma^{M-1}$ is a sequence of constructor variables, each of kind **Prop** $\Rightarrow$ **Prop** $\Rightarrow$ **Prop**. Finally, $\vec{\vec{\gamma}}$ is a family of new constructor variables $\gamma^{n,k}$, for $n, k = 0, \ldots, M-1$, each again of kind **Prop** $\Rightarrow$ **Prop** $\Rightarrow$ **Prop**. The family $\vec{\vec{\gamma}}$ should be seen as a collection of M separate copies $\gamma^{n,0}, \ldots, \gamma^{n,M-1}$ of the sequence $\gamma^0, \ldots, \gamma^{M-1}$. (Actually, only the final segments $\gamma^{n,n}, \ldots, \gamma^{n,M-1}$ of these sequences will be used.) All the γ's should be interpreted as 'switches', to be eventually instantiated with the left or right projections.

There are two essential substitutions applied to $\vec{\vec{\gamma}}$: let the symbol $\{\mathscr{L}\}$ stand for $\{\pi_L/\gamma^{k,k}, \gamma^{n+1,k}/\gamma^{n,k}\}_{0 \leqslant n < k \leqslant M-1}$, and let $\{\mathscr{R}\}$ be $\{\pi_R/\gamma^{k,k}, \gamma^{n+1,k}/\gamma^{n,k}\}_{0 \leqslant n < k \leqslant M-1}$. For a given n, the substitutions $\{\mathscr{L}\}$ and $\{\mathscr{R}\}$ behave as follows: they leave $\gamma^{n,k}$ unchanged, for $k < n$, substitute a projection for $\gamma^{n,n}$, and renumber the remaining variables $\gamma^{n,n+1}, \ldots, \gamma^{n,M-1}$, to obtain the sequence $\gamma^{n+1,n+1}, \ldots, \gamma^{n+1,M-1}$.

Types to be assigned to the W_*^n's will be constructed according to a certain pattern. We

say that a type $[W]$ is *good for* W iff it is of the form

$$[W] = \forall\beta\vec{\varphi}\,([x] \to \beta \to \omega),$$

(where φ is an arbitrary sequence of constructor variables and β is a type variable) provided $[W]$ satisfies the following additional conditions: first, $[x]$ can be presented as

$$[x] = \mathbf{Config}\{\mathscr{S}\}\{\Xi/\xi\}, \tag{5.1}$$

where $\mathscr{S}$ is a simultaneous substitution defined on all free variables of **Config**, except ξ and β. (In particular, this means that β is left free in $[x]$.) For uniformity, we use the notation $\mathscr{S}(\mathbf{out}) = \mathbf{Out}$. The type Ξ must be as follows:

$$\Xi = \forall\beta\vec{\vec{\gamma}}\,(\sigma \to \beta \to \omega),$$

for some σ, and we require that

$$\mathbf{Out} = \sigma^L\{\forall\beta\vec{\vec{\gamma}}\,(\sigma^R \to \beta \to \omega)/\beta\}, \tag{5.2}$$

where $\sigma^R = \sigma\{\mathscr{R}\}$ and $\sigma^L = \sigma\{\mathscr{L}\}$. Let us stress that the above equations (and all the others) are taken modulo alpha-conversion, and it is just a matter of convenience to write our types with $\vec{\vec{\gamma}}$ as bound variables.

Lemma 5.3. If type $[W]$ is good for W, then $W : [W]$ holds (in the empty environment).

Proof. We must show that the assumption $x : [x]$ implies

$$xpr_5^5\,(\lambda w.\, xpr_4^5)(\lambda w.\, w(Next\ x)w) : \beta \to \omega.$$

Since the type of x is a substitution instance of **Config**, it must be a product of five components, the last of them being a proper instance of $\mathbf{Quit} \to (\xi \to \omega) \to (\beta \to \omega)$. That is, we have $xpr_5^5 : \mathbf{Quit} \to (\Xi \to \omega) \to (\beta \to \omega)$. Since $\lambda w.\, xpr_4^5$ has type $\mathbf{Quit} = \forall\alpha\,(\alpha \to \omega)$, we need only to show that $\lambda w.\, w(Next\ x)w : \Xi \to \omega$. For this, assume w to be of type $\Xi = \forall\beta\vec{\vec{\gamma}}\,(\sigma \to \beta \to \omega)$. The second occurrence of w is assigned the type $\forall\beta\vec{\vec{\gamma}}\,(\sigma^R \to \beta \to \omega)$, obtained by instantiating $\vec{\vec{\gamma}}$ with $\{\mathscr{R}\}$ and then generalizing. For the first occurrence of w, we use $\{\mathscr{L}\}$ instead of $\{\mathscr{R}\}$, and we also substitute β with $\forall\beta\vec{\vec{\gamma}}\,(\sigma^R \to \beta \to \omega)$, to obtain the type $\mathbf{Out} \to \forall\beta\vec{\vec{\gamma}}\,(\sigma^R \to \beta \to \omega) \to \omega$. Since the type of x is a substitution instance of **Config**, by (4.3) we have that $Next\ x : \mathbf{Out}$, and thus the application $w(Next\ x)w$ is well typed, and is assigned type ω. □

A crucial step in our proof is that if we have two good types for W, then sometimes we may join them to obtain another good type for W that is more general than both of them (and thus behaves very much like their intersection).

Lemma 5.4. Let $[W_1] = \forall\beta\vec{\varphi}\,([x_1] \to \beta \to \omega)$ and $[W_2] = \forall\beta\vec{\varphi}\,([x_2] \to \beta \to \omega)$ be good types for W (note that the sequence $\vec{\varphi}$ is the same in both cases). Assume that a variable $\gamma : \mathbf{Prop} \Rightarrow \mathbf{Prop} \Rightarrow \mathbf{Prop}$ does occur in the sequence $\vec{\varphi}$, and is neither free in $[x_1]$ nor in $[x_2]$. Then there exists a type $[W] = \forall\beta\vec{\varphi}\,([x] \to \beta \to \omega)$, which is good for W, and in addition has the property that $[x]\{\pi_L/\gamma\} = [x_1]$ and $[x]\{\pi_R/\gamma\} = [x_2]$.

Proof. Let us assume that for $i = 1, 2$ we have

$$[x_i] = \mathbf{Config}\{\mathscr{S}_i\}\{\Xi_i/\xi\},$$

and that γ does not occur in $\vec{\gamma}$ (this is legitimate due to alpha-conversion). We want $[x]$ to be as in (5.1). For this we define $\mathscr{S}(\zeta) = \gamma\mathscr{S}_1(\zeta)\mathscr{S}_2(\zeta)$, for all variables $\zeta \in \mathrm{Dom}(\mathscr{S})-\{\mathbf{out}\}$. We cannot do the same for **out**, because **Out** is determined by Equation (5.2). Also, $\Xi \to \omega$ should be a legal type for the subterm $\lambda w.\, w(Next\ x)w$, and thus Ξ cannot be of the form $\gamma\Xi_1\Xi_2$. Let us assume that $\Xi_i = \forall\beta\vec{\gamma}\,(\sigma_i \to \beta \to \omega)$ and let $\sigma = \gamma\sigma_1\sigma_2$. We define $\Xi = \forall\beta\vec{\gamma}\,(\sigma \to \beta \to \omega)$. Now we must take $\mathbf{Out} = \sigma^L\{\forall\beta\vec{\gamma}\,(\sigma^R \to \beta \to \omega)/\beta\}$, and we need only show that $\mathbf{Out}\{\pi_L/\gamma\} = \mathscr{S}_1(\mathbf{out})$ and $\mathbf{Out}\{\pi_R/\gamma\} = \mathscr{S}_2(\mathbf{out})$. For this, observe that $\sigma^L = \gamma\sigma_1^L\sigma_2^L$ and $\sigma^R = \gamma\sigma_1^R\sigma_2^R$ and thus $\sigma^L\{\pi_L/\gamma\} = \sigma_1^L$ and $\sigma^L\{\pi_R/\gamma\} = \sigma_2^L$. The same holds for σ^R. It follows that, for example, $\mathbf{Out}\{\pi_L/\gamma\} = \sigma_1^L\{\forall\beta\vec{\gamma}\,(\sigma_1^R \to \beta \to \omega)/\beta\} = \mathscr{S}_1(\mathbf{out})$. □

Let us finally turn to our induction. For each $n = 0, \ldots, M$, we define the types $[W_*^n]$ and $[W^n]$ by a backward induction beginning with $n = M$ and ending with $n = 0$. In order to outline the general shape of our types, let us recall the definition of $[C^n]$, for $n \leqslant M$:

$$[C^n] = \mathbf{Config}\{\mathscr{S}^n\}\{\xi^n/\xi,\ \tau^n/\beta\}.$$

The variables ξ and β occur free in $\mathbf{Config}\{\mathscr{S}^n\}$ only in the last component, which has the form $\mathbf{Quit} \to (\xi \to \omega) \to (\beta \to \omega)$. Thus, ξ^n occurs free only in the last component of $[C^n]$. Similarly, κ is free only in the last component of $[C^M]$. But if $n < M$, then κ and all ξ^k for $k > n$ occur free in $\mathbf{Out}^n = [C^{n+1}]$. Thus, the free variables of all the $[C^n]$'s are κ and ξ^k for $k \geqslant n$.

Types $[W^n]$ are arrow types of the form

$$[W^n] = [x^n] \to Z^n \to \omega,$$

obtained from $[W_*^n]$ by instantiation, and are used for the applications $W^nC^nW_*^{n+1}$. Types $[W_*^n]$ are intended to be good types for W, and they are of the following shape:

$$[W_*^n] = \forall\beta\vec{\gamma}\,([x_*^n] \to \beta \to \omega).$$

We first consider the case $n = M$. For this, put

$$\Xi^M = \forall\beta\vec{\gamma}\,([C^{M+1}] \to \beta \to \omega)$$

and define

$$[x_*^M] = \mathbf{Config}\{\mathscr{S}^M\}\{\Xi^M/\xi\}.$$

Note that the only free variable of $[x_*^M]$ is β, because Ξ^M is closed. We can now see that $[W_*^M]$ is good for W. Clearly, our type is of the required form with $\Xi = \Xi^M$ and $\sigma = \mathbf{Out}^M = [C^{M+1}]$. Since the latter is a closed type, Equation (5.2) is immediately satisfied.

To define $[W^M]$, we need an auxiliary type

$$Z^M = \forall\beta\vec{\gamma}\,([x_*^M] \to \beta \to \omega),$$

which is meant to be assigned to the last copy of W occurring in our computation – the one that is never applied to an argument. Since only β is free in $[x_*^M]$, the above type Z^M differs from $[W_*^M]$ only in the leading (and redundant) quantifiers. (This difference

is essential for the proof of Lemma 5.5.) Now we can define:

$$[W^M] = [x^M] \to Z^M \to \omega,$$

where

$$[x^M] = [x^M_*]\{Z^M/\beta\}.$$

Clearly, $[W^M]$ is obtained from $[W^M_*]$ by instantiation, and thus $W : [W^M]$ holds. One can also easily check that $[x^M] = [C^M]\{\Xi^M/\xi^M\}\{Z^M/\kappa\}$, whence $C^M : [x^M]$.

Let us point out here a difference between the cases $n = M$ and $n < M$. We have $\lambda w.\, wC^{M+1}w : \Xi^M \to \omega$, but since WC^MW does not reduce to $(\lambda w.\, wC^{M+1}w)W$, we do not have to guarantee that Ξ^M is a legal type for W (instead, we derive $W : Z^M$). This is essential for the well-foundedness of our construction.

Assume now that $n < M$. The idea is to define $[W^n_*]$ from $[W^{n+1}_*]$ and $[W^n]$ using Lemma 5.4. However, the type $[W^n]$, which is actually used for the application WC^n is an arrow type and thus it is not good. This means that to define $[W^n_*]$, for $n < M$, we need one more auxiliary type

$$[W^n_\circ] = \forall\beta\vec{\gamma}\,([x^n_\circ] \to \beta \to \omega).$$

Type $[x^n_\circ]$ is of the form

$$[x^n_\circ] = \mathbf{Config}\{\mathscr{S}^n\}\{\Xi^k/\xi^k\}_{k>n}\{\Xi^n/\xi\}\{Z^M/\kappa\},$$

where the Ξ's are certain closed types, still to be defined. One can easily see that the only free variable of $[x^n_\circ]$ is β. From $[W^n_\circ]$ (for $n < M$) we obtain

$$[W^n] = [x^n] \to \Xi^n \to \omega$$

by an appropriate instantiation, so that

$$[x^n] = [x^n_\circ]\{\Xi^n/\beta\}.$$

Note that $Z^n = \Xi^n$, for $n < M$, while $Z^M \neq \Xi^M$. One can easily verify that $[x^n]$ is obtained from $[C^n]$ by substitution, namely we have

$$[x^n] = [C^n]\{\Xi^k/\xi^k\}_{k\geqslant n}\{Z^M/\kappa\}.$$

In what follows, the symbol $[W^M_\circ]$ is another notation for $[W^M_*]$. The following lemma states that our construction can be successfully completed.

Lemma 5.5. There exist types Ξ^n, such that if $[W^n_*]$ and $[W^n_\circ]$ are defined as above, then, for each $n = M, \ldots, 0$:

(i) $[W^n_*]$ and $[W^n_\circ]$ are good types for W;
(ii) for $n < M$, we have that $[x^n_*]\{\mathscr{L}^n\} = [x^n_\circ]$ and $[x^n_*]\{\mathscr{R}^n\} = [x^{n+1}_*]$, where $\{\mathscr{L}^n\}$ abbreviates the substitution $\{\pi_L/\gamma^n\}$, and $\{\mathscr{R}^n\}$ stands for $\{\pi_R/\gamma^n\}$.

Proof. For $n = M$ we have already verified (i), and (ii) is not applicable. For uniformity, we can extend (ii) so that it also holds for $n = M$ by assuming that $[x^M_\circ]$ and $[x^{M+1}_*]$ both stand for $[x^M_*]$, and that $\{\mathscr{L}^M\}$ and $\{\mathscr{R}^M\}$ are identity substitutions (there is no variable γ^M).

Now let $n < M$ and assume that types Ξ^k for $k > n$ have already been defined, and

that the induction hypothesis holds for $n+1$ in place of n. We begin with the definition of Ξ^n. We must do it so that, for $w' : [W^n]$, $c : [x^n]$ and $w'' : [W_*^{n+1}]$, the application $w'cw''$ is well typed. (This is because $[W^n]$ and $[W_*^{n+1}]$ are to be assigned, respectively, to the copies W^n and W_*^{n+1} of W.) For this reason, it would suffice to have Ξ^n equal to $[W_*^{n+1}]$. We want, however, $[W_\circ^n]$ to be good for W, and thus we have to be more careful here, and we also need more notation. Recall that $\vec{\vec{\gamma}}$ consists of variables $\gamma^{n,k}$, for $n,k = 0,\ldots,M-1$. For each i, we use $[x_{**}^i]$ to denote the type $[x_*^i]\{\gamma^{i,k}/\gamma_k\}_{k\geqslant i}$ – a copy of $[x_*^i]$ using its own private variables for the 'switches' $\vec{\gamma}$. (Let us remark here that this causes a renaming of bound $\vec{\vec{\gamma}}$.) We define:

$$\Xi^n = \forall\beta\vec{\vec{\gamma}}\,([x_{**}^{n+1}] \to \beta \to \omega),$$

which is *almost* the same as $[W_*^{n+1}]$ (the only difference is the redundant quantifiers), and certainly suffices to get $w'cw'' : \omega$. Observe that $[W_*^{n+1}] \preceq \Xi^n = Z^n$.

Before we define $[W_*^n]$ for $n < M$, we show that $[W_\circ^n]$ is good for W. Of course, our type is of the correct shape, and we have to verify Equation (5.2) with $\mathbf{Out} = \mathbf{Out}^n\{\Xi^k/\xi^k\}_{k\geqslant n}\{Z^M/\kappa\}$ and $\sigma = [x_{**}^{n+1}]$ (because $\Xi = \Xi^n$). Since $[x^{n+1}]$ is equal to $[C^{n+1}]\{\Xi^k/\xi^k\}_{k\geqslant n+1}\{Z^M/\kappa\}$, and $\mathbf{Out}^n = [C^{n+1}]$, we simply have $\mathbf{Out} = [x^{n+1}]$ (remember that ξ^n does not occur free in $[C^{n+1}]$).

If $n = M-1$, Condition (5.2) is equivalent to the equation

$$[x^M] = [x_{**}^M]\{\forall\beta\vec{\vec{\gamma}}([x_{**}^M] \to \beta \to \omega)/\beta\}.$$

But $[x_{**}^M] = [x_*^M]$, and thus the above can be rewritten as '$[x^M] = [x_*^M]\{Z^M/\beta\}$' – the definition of $[x^M]$. (Note that this would not work for $[W_*^M]$ in place of Z^M.)

Now assume $n < M-1$. From the induction hypothesis (ii) we know that $[x_*^{n+1}]\{\mathscr{L}^{n+1}\} = [x_\circ^{n+1}]$ and $[x_*^{n+1}]\{\mathscr{R}^{n+1}\} = [x_*^{n+2}]$. By a renaming of variables, this translates to $\sigma^L = [x_{**}^{n+1}]\{\mathscr{L}\} = [x_\circ^{n+1}]$ (remember that γ's do not occur free in $[x_\circ^{n+1}]$) and $\sigma^R = [x_{**}^{n+1}]\{\mathscr{R}\} = [x_{**}^{n+2}]$. Thus, we can write Equation (5.2) as follows:

$$[x^{n+1}] = [x_\circ^{n+1}]\{\forall\beta\vec{\vec{\gamma}}\,([x_{**}^{n+2}] \to \beta \to \omega)/\beta\}.$$

By definition, $[x^{n+1}] = [x_\circ^{n+1}]\{\Xi^{n+1}/\beta\}$. And this is exactly what we need, straight from the definition of Ξ^{n+1}.

This concludes the verification of (i). Now as we know that both $[W_\circ^n]$ and $[W_*^{n+1}]$ are good, we may apply Lemma 5.4 to obtain $[W_*^n]$, with γ^n used as γ, and of course we easily get part (ii) of the induction hypothesis. □

The proof of correctness of our construction has been completed, and we can now conclude with our main result.

Theorem 5.6. Each partial recursive function is nonuniformly represented in $\mathbf{F}_\omega$.

Proof. Let f be a partial recursive function computed by a given Turing Machine with output. Then f is represented by the term $F = \lambda N.(\lambda w.\, w(BaseN)w)W$. If the machine converges, the application FN is typable. Indeed, we have $N : [N]$, $W : [W_*^0]$ and $BaseN : [C^0]$ in the empty environment. Since $[x^0]$ is a substitution instance of $[C^0]$, we also have $BaseN : [x^0]$, and one can easily derive $wcw : \omega$ from $w : [W_*^0]$ and $c : [x^0]$,

using Lemma 5.5(ii) for $n = 0$ (note that $[x^0] = [x^0_*]\{\mathscr{L}^0\}\{\Xi^0/\beta\}$). Otherwise, FN has an infinite reduction sequence and therefore must be untypable. □

Theorem 5.7. The type reconstruction problem for $\mathbf{F}_\omega$ is undecidable, as well as for all the subsystems $\mathbf{F}^{(n)}_\omega$, for $n > 0$. In addition, the set of terms typable in $\mathbf{F}^{(1)}_\omega$ is not recursively separable from the set of normalizable terms.

Proof. The above representation is effective, that is, the application FN has been effectively obtained from a given Turing Machine and an input value. This means that we have reduced the halting problem to the type reconstruction problem. In addition, the whole typing works in $\mathbf{F}^{(1)}_\omega$. This implies undecidability at all levels.

It should also be obvious that typability cannot be recursively separated from *strong* normalization, since we have shown so far that the term FN is either typable or has an infinite reduction sequence. However, in the latter case, FN cannot have a normal form at all. To see this, observe that the infinite reduction sequence is quasi-leftmost, *i.e.*, the leftmost redex is reduced infinitely often (see Barendregt (1984) for the proof that quasi-leftmost reductions are normalizing). □

Acknowledgment

The first version of the main proof was so complicated that Joe Wells was perhaps the only person patient enough to read and understand it in full. His comments (and especially the idea of stop register) provided a significant help in improving the proof. Remarks from Fritz Henglein, Harry Mairson and an anonymous referee helped in further improvements and corrections.

References

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, North-Holland.

Barendregt, H. P. and Hemerik, K. (1990) Types in lambda calculi and programming languages. In: Jones, N. (ed.) Proc. 3rd European Symposium on Programming. *Springer-Verlag Lecture Notes in Computer Science* **432** 1–36.

Barendregt, H. P. (1990) Lambda calculi with types. In: Abramsky, S., Gabbay, D. M. and Maibaum, T. (eds.) *Handbook of Logic in Computer Science*, Oxford University Press 117–309.

Boehm, H. J. (1985) Partial polymorphic type inference is undecidable. In: *Proc. 26th Foundations of Computer Science*, IEEE 339–345.

Cardonne, F. and Coppo, M. (1990) Two extensions of Curry's type inference system. In: Odifreddi, P. (ed.) *Logic and Computer Science*, Academic Press 19–75.

Coquand, T. (1990) Metamathematical investigations of a calculus of constructions. In: Odifreddi, P. (ed.) *Logic and Computer Science*, Academic Press 91–122.

Gabbay, D. M. (1981) *Semantical Investigations in Heyting's Intuitionistic Logic*, Reidel.

Gallier, J. H. (1990) On Girard's 'Candidats de Reductibilité'. In: Odifreddi, P. (ed.) *Logic and Computer Science*, Academic Press 123–203.

Giannini, P. and Ronchi Della Rocca, S. (1988) Characterization of typings in polymorphic type discipline. In: *Proc. 3rd Logic in Computer Science*, IEEE 61–70.

Giannini, P. and Ronchi Della Rocca, S. (1991) Type inference in polymorphic type discipline, In: Ito, T. and Meyer, A. R. (eds.) Proc. Theoretical Aspects of Computer Software. *Springer-Verlag Lecture Notes in Computer Science* **526** 18–37.

Giannini, P., Ronchi Della Rocca, S. and Honsell, F. (1993) Type inference: some results, some problems. *Fundamenta Informaticae* **19** (1-2) 87–126.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieure*, Thèse d'État, Université Paris VII.

Girard, J.-Y. (1986) The system **F** of variable types fifteen years later. *Theoret. Comput. Sci.* **45** 159–192.

Goldfarb, W. D. (1981) The undecidability of the second-order unification problem. *Theoret. Comput. Sci.* **13** 225–230.

Henglein, F. (1990) A lower bound for full polymorphic type inference: Girard/Reynolds typability is DEXPTIME-hard, Technical Report RUU-CS-90-14, University of Utrecht.

Henglein, F. and Mairson, H. G. (1994) The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming* **4** (4) 435–477.

Kanellakis, P. C. and Mitchell, J. C. (1989) Polymorphic unification and ML typing. In: *Proc. 16th Principles of Programming Languages*, ACM 105–115.

Kanellakis, P. C., Mairson H. G. and Mitchell J. C. (1991) Unification and ML type reconstruction. In: Lassez, J.-L. and Plotkin, G. (eds.) *Computational Logic, Essays in Honor of Alan J. Robinson*, MIT Press.

Kfoury, A. J. and Tiuryn, J. (1992) Type reconstruction in finite-rank fragments of the second-order λ-calculus. *Information and Computation* **98** (2) 228–257.

Kfoury, A. J., Tiuryn, J. and Urzyczyn, P. (1993a) The undecidability of the semi-unification problem. *Information and Computation* **102** (1) 83–101.

Kfoury, A. J., Tiuryn, J. and Urzyczyn, P. (1993b) Type reconstruction in the presence of polymorphic recursion. *ACM TOPLAS* **15** (2) 290–311.

Kfoury, A. J., Tiuryn, J. and Urzyczyn, P. (1994) An analysis of ML typability. *Journal of the ACM* **41** (2) 368–398.

Krivine, J. L. (1987) Un algorithme non typable dans le système **F**. *F.C.R. Acad. Sci. Paris*, Série I **304** 123–126.

Leivant, D. (1983) Polymorphic type inference. In: *Proc. 10th Principles of Programming Languages*, ACM 88–98.

Leivant, D. (1990) Discrete polymorphism. In: *Proc. 6th LISP and Functional Programming*, ACM 288–297.

Mairson, H. G. (1990) Deciding ML typability is complete for deterministic exponential time. In: *Proc. 17th Principles of Programming Languages*, ACM 382–401.

Malecki, S. (1990) Generic terms having no polymorphic types. In: Paterson, M. S. (ed.) Proc. International Colloquium on Automata Languages and Programming. *Springer-Verlag Lecture Notes in Computer Science* **443** 46–59.

Mitchell, J. C. (1988) Polymorphic type inference and containment. *Information and Computation* **76** (2-3) 211–249.

Pfenning, F. (1993) On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae* **19** (1-2) 185–199.

Pierce, B., Dietzen, S. and Michaylov, S. (1989) Programming in higher-order typed lambda calculi. Research Report CMU-CS-89-111, Carnegie-Mellon University.

Pottinger, G. (1980) A type assignment for the strongly normalizable λ-terms. In: Seldin, J. P. and Hindley, J. R. (eds.) *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press 561–577.

Urzyczyn, P. (1993) Type reconstruction in $\mathbf{F}_\omega$ is undecidable. In: Bezem, M. and Groote, J. F. (eds.) Proc. Typed Lambda Calculus and Applications. *Springer-Verlag Lecture Notes in Computer Science* **664** 418–432.

Urzyczyn, P. (1992) Type reconstruction in $\mathbf{F}_\omega$, Research Report BUCS #92-01, Boston University.

Urzyczyn, P. (1995) Positive recursive type assignment. In: Wiedermann, J. and Hajek, P. (eds.) Proc. 20th Mathematical Foundations of Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **969** 382–391.

Wells, J. B. (1994) Typability and type checking in the second-order λ-calculus calculus are equivalent and undecidable. In: *Proc. 9th Logic in Computer Science*, IEEE 176–185.