

General recursion on second order term algebras^{*}

Alessandro Berarducci¹ and Corrado Böhm²

¹ Università di Pisa, Dipartimento di Matematica,
Via Buonarroti 2, 56127 Pisa, Italy,
berardu@dm.unipi.it,
<http://fibonacci.dm.unipi.it/~berardu/>

² Università di Roma “La Sapienza”,
Dipartimento di Scienze dell’Informazione,
Via Salaria 113, 00198 Roma, Italy,
boehm@dsi.uniroma1.it

Abstract. Extensions of the simply typed lambda calculus have been used as a metalanguage to represent “higher order term algebras”, such as, for instance, formulas of the predicate calculus. In this representation bound variables of the object language are represented by bound variables of the metalanguage. This choice has various advantages but makes the notion of “recursive definition” on higher order term algebras more subtle than the corresponding notion on first order term algebras. Despeyroux, Pfenning and Schürmann pointed out the problems that arise in the proof of a canonical form theorem when one combines higher order representations with primitive recursion.

In this paper we consider a stronger scheme of recursion and we prove that it captures all partial recursive functions on second order term algebras. We illustrate the system by considering typed programs to reduce to normal form terms of the untyped lambda calculus, encoded as elements of a second order term algebra. First order encodings based on de Bruijn indexes are also considered. The examples also show that a version of the intersection type disciplines can be helpful in some cases to prove the existence of a canonical form. Finally we consider interpretations of our typed systems in the pure lambda calculus and a new gödelization of the pure lambda calculus.

1 Introduction

Despite some limitations, first order term rewriting systems can serve as a framework for functional programming. In this approach one has a first order signature whose symbols are partitioned in two sets: constructor symbols and program symbols. A data structure is identified with the set of normal forms of a given type built up from a given set of constructor symbols. To each program symbol is associated a set of recursive equations which, when interpreted as rewrite rules, define the operational semantics of the program.

^{*} Dedicated to Nicolaas G. de Bruijn

This approach however is not adequate to handle programs on syntactic data structures involving variable-binding operators. Consider for instance a program to find the prenex normal form of a formula of the predicate calculus, or a program to reduce lambda terms. In principle one can encode the relevant data structures as first order term algebras, and represent the corresponding programs by first order term rewriting systems. It is however more convenient to assume that the programming environment (namely the “metatheory”) is based on some version of the lambda calculus. In this approach bound variables of the object language are represented by bound variables of the metalanguage (rather than by strings of characters or de Bruijn indexes). This means that one can make use of the built-in procedures to handle renaming of bound variables and substitutions, which otherwise must be implemented separately in each case. Along these lines [14] shows that many important syntactic data structures involving variable-binding mechanisms can be represented in a version of the typed lambda calculus with additional constant symbols (higher order constructors). In this representation scheme the elements of a given data structure correspond to the “canonical forms” of a certain type built up from a certain set of higher order constructor symbols. We call “higher order term algebra” the set of canonical forms representing the elements of a given data structure.

In order to do some functional programming on such algebras we must introduce a notion of “recursion”. A difficulty is that an element of a higher order algebra is a normal form of atomic type, and yet it can have subterms of functional type; so it is not clear, at least semantically, in which sense these algebras can be considered to be inductively defined. The problem is considered in [13, 12] under different perspectives. In the presence of recursion the existence of canonical forms becomes a rather delicate issue. Essentially this depends on the fact that, a recursively defined function applied to a formal parameter (i.e. a bound variable), cannot reduce to a canonical form. In other words, to be able to evaluate a recursive function, or more generally a function defined by cases, it is necessary that its recursive argument is “constructed” rather than parametric. Thus, to prove that a given term has a canonical form, one must ensure that occurrences of recursively defined programs applied to formal parameters do not arise dynamically during the computation. The solution proposed in [12], in the case of *primitive* recursion on higher order algebras, is a typed lambda calculus enriched with modalities whose purpose is to make a type distinction between parametric and constructed objects. This is reminiscent of the notion of “safe” recursion of [5] or of the “tiers” of [16]. Following a different line of research, Constable [11] introduced an extension of the typed lambda calculus with *general* recursion, based on fixed points operators. Here one is only interested in first order data structures (booleans, integers), and the issues are different: the main type distinction to be made concerns partial versus total functions rather than parametric versus constructed objects.

Inspired by the work of these authors and continuing the work done in [6, 9] in an untyped setting, we propose an extension of the simply typed lambda calculus with a scheme of recursion based on the distinction between program

symbols and constructor symbols. We show that our scheme is able to capture all partial recursive functions on first order or second order term algebras.

Much of the emphasis of this paper is on the examples. We begin with a program to compute the prenex normal form of a formula of the predicate calculus (represented as in [14]). Then, as an example of a partial recursive function on a higher order term algebra, we consider (typed) programs to reduce terms of the untyped lambda calculus to normal form (where inputs and outputs are encoded as elements of a higher order term algebra). Similar programs have been discussed in [17, 6, 3] in an untyped setting.

In the case of partial functions the canonical form theorem takes the following conditional form: if a computation terminates, then it terminates in a canonical form. To prove that a given program has this property, a typing system based on the simple types may not be sufficient and a more refined typing based on a variant of the intersection type disciplines of [4] may be useful. We illustrate this fact proving the canonical form theorem for our program to reduce lambda terms.

All the experiments have been computer-tested by a software called “CuCh machine”. The CuCh machine is essentially an interpreter of the pure lambda calculus together with a macro to transform recursive definitions into lambda terms developed by Böhm and Piperno and explained in [6, 9]. Quite remarkably the interpretation described in these papers does not make use of the fixed point operator.

2 Extensions of the typed lambda calculus

Given a set of atomic types we generate a set of types as follows.

Definition 1. *The types α are generated by the grammar*

$$\alpha ::= \langle \text{atomic type} \rangle \mid \alpha_1 \times \dots \times \alpha_n \rightarrow \langle \text{atomic type} \rangle$$

Note that we allow cartesian products only on the left of the arrow. So $\alpha_1 \times \alpha_2$ is not a type.

Definition 2. *A signature Σ is a set of “constant declarations” of the form $c:\alpha$, where α is a type (over a given set of atomic types). If $c:\alpha \in \Sigma$ we say that c is a constant symbol of type α .*

We allow a constant symbol to have more than one type in a given signature, even an infinite set of types. If the set is finite, this amounts to the possibility of assigning to a constant symbol the intersection of all the types of the set, in the sense of the intersection type disciplines of [4].

Definition 3. *Given a signature Σ , a basis B is a set of “parameter declarations” of the form $y:\alpha$, where y is a variable and α is a type. We stipulate that a basis cannot contain two different declarations $y:\alpha$ and $y:\beta$ for the same variable. The following type assignment system defines inductively the set of terms t of type α over a basis B .*

$$\begin{array}{c}
(\text{Const.}) \frac{c: \alpha \in \Sigma}{\vdash c: \alpha} \qquad (\text{Var.}) \frac{x: \alpha \in B}{B \vdash x: \alpha} \\
(\rightarrow \text{I}) \frac{B, x_1: \alpha_1, \dots, x_n: \alpha_n \vdash t: \alpha}{B \vdash \lambda x_1 \dots x_n. t: \alpha_1 \times \dots \times \alpha_n \rightarrow \alpha} \\
(\rightarrow \text{E}) \frac{B \vdash t: \alpha_1 \times \dots \times \alpha_n \rightarrow \alpha \quad B \vdash t_1: \alpha_1 \cdots B \vdash t_n: \alpha_n}{B \vdash tt_1 \dots t_n: \alpha} \\
(\text{Weakening}) \frac{B \vdash t: \alpha \quad B' \supset B}{B' \vdash t: \alpha}
\end{array}$$

In rule $(\rightarrow \text{I})$, B does not contain declarations for x_1, \dots, x_n . If we can deduce $B \vdash t: \alpha$ by the above rules we say that t is a term of type α over the basis B .

Note that $\lambda xy.t$ cannot be applied to a single argument: it necessarily requires two arguments (so it is not identified with $\lambda x.(\lambda y.t)$, which is not a legitimate term because we only allow atomic types on the left hand side of an arrow). This feature of our system enables to encode bijectively the elements of various (higher order) data structures as the closed normal forms of a given atomic type over a given signature (thus rendering superfluous the distinction between normal form and canonical form in [14])

The behaviour of the constant symbols of the signature is dictated by a set of reduction rules.

Definition 4. Given a signature Σ , a **reduction rule** is a pair (t_1, t_2) , written $t_1 := t_2$, such that for every basis B and every type α , if $B \vdash t_1: \alpha$, then $B \vdash t_2: \alpha$.

Definition 5. A set P of reduction rules over a signature Σ determines an extension $\Lambda(\Sigma, P)$ of the simply typed lambda calculus as follows. The terms of this calculus are as defined in Definition 3. The **reduction relation** “ \rightarrow ” between typable terms of $\Lambda(\Sigma, P)$ is obtained by adding to the β -rule $(\lambda x_1 \dots x_n. t)t_1 \dots t_n \rightarrow t[t_1/x_1, \dots, t_n/x_n]$ all the reductions $t_1 \rightarrow t_2$ for each reduction rule $t_1 := t_2$ of the program P . By definition, the reduction relation is transitive and closed under substitutions and contexts. We identify terms which differ only by a renaming of bound variables.

The following “subject reduction” theorem holds.

Theorem 1. If $t_1 \rightarrow t_2$ in $\Lambda(\Sigma, P)$ and $B \vdash t_1: \alpha$, then $B \vdash t_2: \alpha$.

3 General recursive programs on second order term algebras

So far nothing forbids the presence of reduction rules for which the Church-Rosser property for $\Lambda(\Sigma, P)$ fails. We will now introduce further restrictions on the shape of the reduction rules which are always satisfied in all the examples,

and which ensure that the Church-Rosser property holds and that $A(\Sigma, P)$ is interpretable in the untyped lambda calculus. This is done by distinguishing between constructor symbols and program symbols (or destructors), as in [15, 6, 2, 10].

Definition 6. *We say that a set of reduction rules P over a signature Σ is **dichotomic** if the constant symbols of Σ can be partitioned in two sets, **program symbols** and **constructor symbols** in such a way that each reduction rule of P has the form*

$$\langle \text{program} \rangle (\langle \text{constructor} \rangle x_1, \dots, x_n) y_1 \dots y_m := t$$

We assume moreover that the rules of P are mutually exclusive and exhaustive in the sense that for each closed term t whose outermost symbol is a program symbol, and whose first argument begins with a constructor symbol, there is one and only one rule $t_1 := t_2$ such that t is a substitution instance of t_1 . We also require that each program symbol appears in as the left-most symbol of at least one equation of P . So if P is empty all the symbols of the signature are constructor symbols. Finally we will assume that the right-hand sides of the reduction rules of P have no β -redexes.

We now make some further assumptions concerning the complexity of the types of the signature.

Definition 7. *The **level** of a type is defined as the maximum number of nested occurrences of arrow symbols.*

So the level of an atomic type is zero and the level of a type of the form $\alpha_1 \times \dots \times \alpha_n \rightarrow \beta$ is the maximum of the levels of $\alpha_1, \dots, \alpha_n$ plus 1 (recall that β is atomic, so it has level zero).

Definition 8. *Given a dichotomic set of rules P over a signature Σ , we say that Σ and P are **second order**, if the types of the program symbols have level at most 1 and the types of all constructor symbols have level at most 2.*

All the examples we consider in this paper are based on dichotomic rules on a second order signature. The second order assumption will be used in section 8 to prove the representability of every partial recursive function.

4 The prenex normal form example

Our first example of a program on second order term algebras is a procedure **Pnf** to put a formula of the predicate calculus into prenex normal form.

Definition 9. *(Formulas of the predicate calculus) For simplicity we consider formulas of the predicate calculus whose only non-logical symbol is a binary relation symbol A . Following [14], to represent such formulas as typed terms of the extended lambda calculus we need an atomic type ι (individuals), an atomic type o (predicates), and the following constructor symbols:*

$$\begin{aligned}
\text{Fa} & : (\iota \rightarrow o) \rightarrow o \\
\text{Ex} & : (\iota \rightarrow o) \rightarrow o \\
\text{not} & : o \rightarrow o \\
\text{imp} & : o \times o \rightarrow o \\
\text{A} & : \iota \times \iota \rightarrow o
\end{aligned}$$

Universal quantification $\forall xP$ will be represented as $\text{Fa}(\lambda x.P)$ and existential quantification $\exists xP$ will be represented as $\text{Ex}(\lambda x.P)$. So a quantifier transforms a function from individuals to predicates into a predicate. The constructors **not** and **imp** stand for the negation and the implication sign.

For example, the formula $\forall x\forall y(\text{A } x y \rightarrow \neg \text{A } y x)$ is represented by the term $\text{Fa}(\lambda x.(\text{Fa}(\lambda y.\text{imp}(\text{A } x y)(\text{not}(\text{A } y x))))$ of type o . It is easy to see that the representation function maps bijectively closed formulas of the predicate calculus into closed normal forms of type o .

Definition 10. (*Prenex normal form*) The program **Pnf** defined below, computes the prenex normal form of a formula.

Program symbols:

$$\begin{aligned}
\mathbf{Pnf} & : o \rightarrow o \\
\mathbf{nPnf} & : o \rightarrow o \\
\mathbf{iPnf} & : o \times o \rightarrow o \\
\mathbf{iP2} & : o \times o \rightarrow o
\end{aligned}$$

Reduction rules:

$$\mathbf{Pnf}(\text{Fa } t) := \text{Fa}(\lambda T.(\mathbf{Pnf}(tT))) \quad (1)$$

$$\mathbf{Pnf}(\text{Ex } t) := \text{Ex}(\lambda T.(\mathbf{Pnf}(tT))) \quad (2)$$

$$\mathbf{Pnf}(\text{not } L) := \mathbf{nPnf}(\mathbf{Pnf} L) \quad (3)$$

$$\mathbf{Pnf}(\text{imp } L M) := \mathbf{iPnf}(\mathbf{Pnf} L)(\mathbf{Pnf} M) \quad (4)$$

$$\mathbf{Pnf}(\text{A } u v) := \text{A } u v \quad (5)$$

$$\mathbf{nPnf}(\text{Fa } t) := \text{Ex}(\lambda T.(\mathbf{Pnf}(\text{not}(tT)))) \quad (6)$$

$$\mathbf{nPnf}(\text{Ex } t) := \text{Fa}(\lambda T.(\mathbf{Pnf}(\text{not}(tT)))) \quad (7)$$

$$\mathbf{nPnf}(\text{not } L) := L \quad (8)$$

$$\mathbf{nPnf}(\text{imp } L M) := \text{not}(\text{imp } L M) \quad (9)$$

$$\mathbf{nPnf}(\text{A } u v) := \text{not}(\text{A } u v) \quad (10)$$

$$\mathbf{iPnf}(\text{Fa } t) y := \text{Ex}(\lambda T.(\mathbf{Pnf}(\text{imp}(tT) y))) \quad (11)$$

$$\mathbf{iPnf}(\text{Ex } t) y := \text{Fa}(\lambda T.(\mathbf{Pnf}(\text{imp}(tT) y))) \quad (12)$$

$$\mathbf{iPnf}(\text{not } L) y := \mathbf{iP2} y(\text{not } L) \quad (13)$$

$$\mathbf{iPnf}(\text{imp } L M) y := \mathbf{iP2} y(\text{imp } L M) \quad (14)$$

$$\mathbf{iPnf}(\text{A } u v) y := \mathbf{iP2} y(\text{A } u v) \quad (15)$$

$$\mathbf{iP2}(\text{Fa } t) x := \text{Fa}(\lambda T.(\mathbf{Pnf}(\text{imp } x(tT)))) \quad (16)$$

$$\mathbf{iP2}(\text{Ex } t) x := \text{Ex}(\lambda T.(\mathbf{Pnf}(\text{imp } x(tT)))) \quad (17)$$

$$\mathbf{iP2}(\text{not } L) x := \text{imp } x(\text{not } L) \quad (18)$$

$$\mathbf{iP2}(\text{imp } L M) x := \text{imp } x(\text{imp } L M) \quad (19)$$

$$\mathbf{iP2}(\text{A } u v) x := \text{imp } x(\text{A } u v) \quad (20)$$

The above set P of equations defines an extension $\Lambda(P, \Sigma)$ of the typed lambda calculus as in Definition 5 (where Σ is signature of P).

Remark 1. The role of the various equations should be clear. For instance equation 11 takes care of the fact that the prenex normal form of a formula of the form $(\forall x A) \rightarrow B$ is equal to the prenex normal form of $\exists x(A \rightarrow B)$, provided x does not occur free in B . Note that the proviso is implicitly taken into account by the built-in rules of the λ -calculus with renaming of bound variables to avoid unwanted capture of variables.

5 A partial recursive example

The program for the prenex normal form is total recursive, namely it always terminates. As an example of a partial recursive second order program we will define a program \mathbf{Nf} (typable within our system) to reduce closed terms of the untyped lambda calculus to normal form. The program \mathbf{Nf} is an improvement of the one we presented in [6] (in an untyped metatheory). The main novelty is the use of the auxiliary data structure “list” which allows for a considerable gain in efficiency. Our program should also be compared with the one in [17], which is very elegant but conceptually rather complex, as it requires, recursively, a duplication of terms into a “functional part” and an “argument part”.

Definition 11. *To represent untyped lambda terms in our typed metatheory we use the following signature Σ_0 of second order constructors.*

$$\begin{aligned} \text{App} &: \text{exp} \times \text{exp} \rightarrow \text{exp}; \\ \text{Abs} &: (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp} \end{aligned}$$

Definition 12. *Given a term t of the untyped lambda calculus, define $\lceil t \rceil$ inductively as follows:*

$$\begin{aligned} \lceil x \rceil &:= x; \\ \lceil MN \rceil &:= \text{App} \lceil M \rceil \lceil N \rceil; \\ \lceil \lambda x.M \rceil &:= \text{Abs}(\lambda x. \lceil M \rceil) \end{aligned}$$

So for instance

$$\lceil (\lambda x.xx)(\lambda x.xx) \rceil = \text{App}(\text{Abs}(\lambda x.\text{App } x x))(\text{Abs}(\lambda x.\text{App } x x))$$

Remark 2. The above encoding is adequate: the map $t \mapsto \lceil t \rceil$ is a bijection from terms of the untyped lambda calculus, to normal forms of type exp over Σ_0 .

The above representation is a variant of the one in [17] where the author defines $\lceil x \rceil = \text{Var } x$, with Var a new constructor. We will use the following characterization of normal forms:

Remark 3.

$$\text{nf}(x A_1 \dots A_n) = x \text{nf}(A_1) \dots \text{nf}(A_n); \quad (1)$$

$$\text{nf}(\lambda x. B) = \lambda x. \text{nf}(B); \quad (2)$$

$$\text{nf}((\lambda x. B) A_1 \dots A_n) = \text{nf}(B[A_1/x] A_2 \dots A_n) \quad (3)$$

We will define a program \mathbf{Nf} such that $\mathbf{Nf}[t] = \lceil \text{nf}(t) \rceil$ for each *closed* term t having a normal form $\text{nf}(t)$. Although we are not interested in the behaviour of \mathbf{Nf} on open terms, the above equations suggest that to carry out the recursion we are forced to take into account not only closed terms, but also open terms. However we cannot hope to have $\mathbf{Nf}[t] = \lceil \text{nf}(t) \rceil$ even for open terms, because otherwise taking $n = 0$ in the first equation above we get $\mathbf{Nf}[x] = \lceil x \rceil$, which implies $\mathbf{Nf} x = x$ (as $\lceil x \rceil = x$), namely \mathbf{Nf} is the identity function. So we must decide what is the relevant equation for \mathbf{Nf} on open terms. The solution is $\mathbf{Nf}[t]^{\text{Box}} = \lceil \text{nf}(t) \rceil$, where $[t]^{\text{Box}}$ is obtained from $[t]$ by substituting each free variable x by $\text{Box} x$. Here Box is an auxiliary constructor of type $\text{exp} \rightarrow \text{exp}$. So for instance $\lceil \lambda y. xy \rceil^{\text{Box}} = \text{Abs}(\lambda y. \text{App}(\text{Box} x) y)$. Note that for closed terms $[t]^{\text{Box}} = [t]$.

We are finally ready to give the reduction rules defining \mathbf{Nf} . The idea is that \mathbf{Nf} will introduce a Box each time an abstraction is passed over, and will eliminate it when the corresponding variable is reached. The dots “...” in the equations of Remark 3 suggest the use of the data structure “list”. In other words it is convenient to generalize the program $\mathbf{Nf} : \text{exp} \rightarrow \text{exp}$ to a program $\mathbf{Reduce} : \text{exp} \times \text{list} \rightarrow \text{exp}$ which compute the normal form of a term applied to a list of terms.

Definition 13. (A program to compute the normal form of a closed lambda term) Besides the constructors App, Abs we use the following auxiliary constructor symbols:

$$\begin{aligned} \text{Box} & : \text{exp} \rightarrow \text{exp}; \\ \text{nil} & : \text{list}; \\ \text{cons} & : \text{exp} \times \text{list} \rightarrow \text{list} \end{aligned}$$

where list is an atomic type to represent lists of objects of type exp .

Program symbols:

$$\begin{aligned} \mathbf{Reduce} & : \text{exp} \times \text{list} \rightarrow \text{exp}; \\ \mathbf{RBox} & : \text{list} \times \text{exp} \rightarrow \text{exp}; \\ \mathbf{RAbs} & : \text{list} \times (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp} \end{aligned}$$

Reduction rules:

$$\mathbf{Reduce}(\text{App } x y) L := \mathbf{Reduce} x(\text{cons } y L); \quad (1)$$

$$\mathbf{Reduce}(\text{Abs } f) L := \mathbf{RAbs} L f; \quad (2)$$

$$\mathbf{Reduce}(\text{Box } u) L := \mathbf{RBox} L u; \quad (3)$$

$$\mathbf{RAbs} \text{ nil } f := \text{Abs}(\lambda u. \mathbf{Reduce}(f(\text{Box } u)) \text{nil}); \quad (4)$$

$$\mathbf{RAbs}(\text{cons } y L) f := \mathbf{Reduce}(f y) L; \quad (5)$$

$$\mathbf{RBox} \text{ nil } u := u; \quad (6)$$

$$\mathbf{RBox}(\text{cons } y L) u := \mathbf{RBox} L(\text{App } u(\mathbf{Reduce} y \text{ nil})) \quad (7)$$

The above rules define a system $\Lambda(P, \Sigma)$, and within that system we define $\mathbf{Nf} = \lambda x. \mathbf{Reduce} x \text{ nil}$.

Note that an occurrence of the auxiliary constructor Box is introduced in step 4 and is eliminated in step 3.

Example 1. (Example of the computation of a normal form)

Let us compute the normal form of the term $(\lambda xy. xyy)\lambda x. x$. We will use the notation $t_1 \rightarrow_i t_2$ to express that the term t_1 has been rewritten as t_2 applying the rule i to a subterm of t_1 . For better readability we will underline the relevant subterm (when it is not the whole term).

$$\begin{aligned} & \mathbf{Reduce}(\text{App}(\text{Abs}(\lambda x. \text{Abs}(\lambda y. \text{App}(\text{App } x y) y)))(\text{Abs}(\lambda x. x)))\text{nil} \\ \rightarrow_1 & \mathbf{Reduce}(\text{Abs}(\lambda x. \text{Abs}(\lambda y. \text{App}(\text{App } x y) y)))(\text{cons}(\text{Abs}(\lambda x. x))\text{nil}) \\ \rightarrow_2 & \mathbf{RAbs}(\text{cons}(\text{Abs}(\lambda x. x))\text{nil})(\lambda x. \text{Abs}(\lambda y. \text{App}(\text{App } x y) y)) \\ \rightarrow_5 & \mathbf{Reduce}(\underline{(\lambda x. \text{Abs}(\lambda y. \text{App}(\text{App } x y) y))}(\text{Abs}(\lambda x. x)))\text{nil} \\ \rightarrow_{\beta} & \mathbf{Reduce}(\text{Abs}(\lambda y. \text{App}(\text{App}(\text{Abs}(\lambda x. x)) y) y))\text{nil} \\ \rightarrow_2 & \mathbf{RAbs} \text{ nil}(\lambda y. \text{App}(\text{App}(\text{Abs}(\lambda x. x)) y) y) \\ \rightarrow_4 & \text{Abs}(\lambda u. \mathbf{Reduce}(\underline{(\lambda y. \text{App}(\text{App}(\text{Abs}(\lambda x. x)) y) y)}(\text{Box } u))\text{nil}) \\ \rightarrow_{\beta} & \text{Abs}(\lambda u. \mathbf{Reduce}(\text{App}(\text{App}(\text{Abs}(\lambda x. x))(\text{Box } u))(\text{Box } u))\text{nil}) \\ \rightarrow_1 & \text{Abs}(\lambda u. \mathbf{Reduce}(\text{App}(\text{Abs}(\lambda x. x))(\text{Box } u))(\text{cons}(\text{Box } u)\text{nil})) \\ \rightarrow_1 & \text{Abs}(\lambda u. \mathbf{Reduce}(\text{Abs}(\lambda x. x))(\text{cons}(\text{Box } u)(\text{cons}(\text{Box } u)\text{nil}))) \\ \rightarrow_2 & \text{Abs}(\lambda u. \mathbf{RAbs}(\text{cons}(\text{Box } u)(\text{cons}(\text{Box } u)\text{nil}))(\lambda x. x)) \\ \rightarrow_5 & \text{Abs}(\lambda u. \mathbf{Reduce}(\underline{(\lambda x. x)}(\text{Box } u))(\text{cons}(\text{Box } u)\text{nil})) \\ \rightarrow_{\beta} & \text{Abs}(\lambda u. \mathbf{Reduce}(\text{Box } u)(\text{cons}(\text{Box } u)\text{nil})) \\ \rightarrow_3 & \text{Abs}(\lambda u. \mathbf{RBox}(\text{cons}(\text{Box } u)\text{nil})u) \\ \rightarrow_7 & \text{Abs}(\lambda u. \mathbf{RBox} \text{ nil}(\text{App } u(\mathbf{Reduce}(\text{Box } u)\text{nil}))) \\ \rightarrow_6 & \text{Abs}(\lambda u. \text{App } u(\mathbf{Reduce}(\text{Box } u)\text{nil})) \\ \rightarrow_3 & \text{Abs}(\lambda u. \text{App } u(\mathbf{RBox} \text{ nil } u)) \\ \rightarrow_6 & \text{Abs}(\lambda u. \text{App } u u) \end{aligned}$$

A formal proof of correctness of the program \mathbf{Nf} is long and tedious: it is based on the observation that our reduction rules simulate the equations in Remark 3.

Remark 4. The program in Definition 13, unlike the program in [17], has the property that $\mathbf{Nf}[t]$ is strongly normalizing (every reduction path terminates) whenever t is strongly normalizing.

6 Canonical forms

One of the main difficulties which must be overcome when programming on higher order term algebras is that the typing information may not be sufficient to ensure that a given closed normal form is canonical, namely it represents an element of a given data structure.

Example 2. Consider the system $\Lambda(P, \Sigma)$ of Definition 13. By Remark 2, the closed normal forms of type exp over the signature $\{\text{App}, \text{Abs}\} \subset \Sigma$ represent the untyped lambda terms. Unfortunately within the system $\Lambda(P, \Sigma)$ there are “exotic” closed normal forms of type exp (over Σ) which do not represent any lambda term. Examples of such terms are $\text{Abs}(\lambda x. \mathbf{Nf} x)$ and $\text{Box}(\lambda x. x)$. The first term is particularly bad since it is a normal form containing an occurrence of a program symbol: this difficulty arises with second order programming and has no analogue in the first order case (i.e. when all constructors have level at most 1).

The presence of exotic terms shows that the typing information may be too weak to prove that a program has the correct range. To solve the problem we use a feature of our system that we have not yet exploited: the fact that we allow constructor symbols to have more than one type. The following result illustrates the technique.

Theorem 2. (*Canonical form theorem for \mathbf{Nf}*) *Given a closed term t of the untyped lambda calculus, if $\mathbf{Nf}[t]$ has a normal form in the system $\Lambda(P, \Sigma)$ of Definition 13, then its normal form has the shape $[t']$ for some t' .*

Proof. By the adequacy of the representation (Remark 2), it suffices to show that the normal form of $\mathbf{Nf}[t]$ is a term over the signature $\{\text{App}, \text{Abs}\} \subset \Sigma$ (necessarily of type exp by the subject reduction theorem). To this aim we redefine the signature used in Definition 13 as follows, using two new atomic types $\square\text{exp}$ and $\square\text{list}$ (*warning:* we are not taking \square as a new type constructor):

$$\begin{array}{ll} \text{App} & : \text{exp} \times \text{exp} \rightarrow \text{exp}, \quad \square\text{exp} \times \square\text{exp} \rightarrow \square\text{exp}; \\ \text{Abs} & : (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}, \quad (\square\text{exp} \rightarrow \square\text{exp}) \rightarrow \square\text{exp}; \\ \text{Box} & : \text{exp} \rightarrow \square\text{exp}; \\ \text{nil} & : \square\text{list}; \\ \text{cons} & : \square\text{exp} \times \square\text{list} \rightarrow \square\text{list}; \\ \text{Reduce} & : \square\text{exp} \times \square\text{list} \rightarrow \text{exp}; \\ \text{RBox} & : \square\text{list} \times \text{exp} \rightarrow \text{exp}; \\ \text{RAbs} & : \square\text{list} \times (\square\text{exp} \rightarrow \square\text{exp}) \rightarrow \text{exp} \end{array}$$

So App and Abs have two types each. The reduction rules of Definition 13 are still correctly typed in this new signature and \mathbf{Nf} has type $\square\text{exp} \rightarrow \text{exp}$. The idea is that $\square\text{exp}$ represents the objects of the form $[t]^{\text{Box}}$ and exp represents the objects of the form $[t]$, as in the discussion following Remark 3. Over the new signature the representation $[t]$ of a closed term t , has both type exp and $\square\text{exp}$, so $\mathbf{Nf}[t]$ is correctly typed and has type exp . By the subject reduction theorem if the computation of $\mathbf{Nf}[t]$ terminates, the result has type exp . An easy induction shows that a closed normal form of type exp is necessarily a term over the signature $\{\text{App}, \text{Abs}\} \subset \Sigma$, hence it is a term of the form $[t']$.

Remark 5. The signature Σ of Definition 13 is obtained from the signature Σ' in the proof of Theorem 2 by identifying $\square\text{exp}$ with exp and $\square\text{list}$ with list . This suggests the following definition.

Definition 14. *We say that a signature Σ' **refines** a signature Σ , if Σ and Σ' have the same number of constant symbols, with the same names but possibly different types, and Σ can be obtained from Σ' by substituting, in the constant declarations, some atomic types with other types (so in particular by identifying some atomic types).*

Theorem 2 shows that to prove that a program of a given system $\Lambda(P, \Sigma)$ has the correct range (or more generally has some desired property), it may be convenient to try to refine the signature, in such a way that the new signature still respects P . Note that when we refine a signature we make less terms typable: in fact, if Σ' refines Σ , and a closed term t has type α over Σ' , then t has type α^s over Σ , where α^s is a substitution instance of α (with the same substitution as in the definition of refinement). The notion of refinement is clearly related with the notion of principal type scheme in the intersection type disciplines [19].

Example 3. Another interesting refinement of the signature of Definition 13, which respects the corresponding reduction rules, uses infinitely many types for each constant symbol parameterized by an index i ranging over the integers \mathbf{Z} :

$$\begin{aligned} \text{Abs} & : (\text{exp}_{i+1} \rightarrow \text{exp}_{i+1}) \rightarrow \text{exp}_{i+1}; \\ \text{App} & : \text{exp}_{i+1} \times \text{exp}_{i+1} \rightarrow \text{exp}_{i+1}; \\ \text{Box} & : \text{exp}_i \rightarrow \text{exp}_{i+1}; \\ \text{nil} & : \text{list}_i; \\ \text{cons} & : \text{exp}_{i+1} \times \text{list}_{i+1} \rightarrow \text{list}_{i+1}; \\ R & : \text{exp}_{i+1} \times \text{list}_{i+1} \rightarrow \text{exp}_i; \\ \mathbf{RBox} & : \text{list}_{i+1} \times \text{exp}_i \rightarrow \text{exp}_i; \\ \mathbf{RAbs} & : \text{list}_{i+1} \times (\text{exp}_{i+1} \rightarrow \text{exp}_{i+1}) \rightarrow \text{exp}_i \end{aligned}$$

Now consider the terms $\lambda x. \mathbf{Nf}(\mathbf{Nf} x)$ and $\text{Abs}(\lambda x. \mathbf{Nf} x)$. Both terms can be typed in the original signature of Definition 13 in which \mathbf{Nf} has type $\text{exp} \rightarrow \text{exp}$. Neither of them can be typed in the refined signature in the proof of Theorem 2 in which \mathbf{Nf} has type $\square \text{exp} \rightarrow \text{exp}$. Only the first one can be typed in the refined signature of Example 3. Collecting information coming from different signatures we gain insight on the program \mathbf{Nf} .

7 From second order to first order representations

Using de Bruijn indexes we can pass from a second order to a first order representation of lambda terms (or formulas, or any other second order syntactic structure). Once a second order program is found, it is easy to transform it into a first order program by implementing the relevant procedures to handle variable substitutions. To illustrate this idea we modify the program of Definition 13 to obtain a program to reduce a lambda term to normal form in de Bruijn notation.

Definition 15. *(De Bruijn notation) We recall the **de Bruijn** notation for lambda-terms. In this notation variable occurrences are replaced by positive integers. For example $\lambda x. \lambda y. xy(\lambda z. x)$ becomes $\lambda \lambda 21(\lambda 3)$. The positive integer n*

indicates a variable which is bounded by the n -th occurrence of λ going upward in the parsing tree of the term. If such an occurrence of λ does not exist, then the integer indicates a free variable.

Note that closed terms which differ only for a renaming of bound variables have the same de Bruijn notation.

Definition 16. *To represent lambda terms in de Bruijn notation we use the following signature:*

Constructor symbols:

```

1   : nat;
S   : nat → nat;
var : nat → term;
abs : term → term;
app : term × term → term

```

For example $\lambda\lambda 21(\lambda 3)$ becomes $\text{abs}(\text{abs}(\text{app } v2 \ v1)(\text{abs } v3))$, where $2 = S(1)$, $3 = S(2)$, $vn = (\text{var } n)$. The de Bruijn terms correspond bijectively to the closed normal forms of type `term`.

The program `nf : term → term` defined below reduces a de Bruijn term to normal form. Unlike the higher order program `Nf` of Definition 13, it works well even when applied to representations of open terms (the free variables are represented by de Bruijn indexes which point “above” the root of the term). The reduction rules in the definition of `nf` are almost identical to those of `Nf`. The main difference is in equation (5) below, where the auxiliary program `sub` is used to simulate the single β -reduction which in the higher order program is built-in.

Definition 17. *(A program to reduce de Bruijn terms to normal form) We set $\text{nf} = \lambda x.\text{reduce } x \text{ nil}$ where `reduce` is defined as follows:*

Auxiliary constructor symbols:

```

nil : list;
cons : term × list → list

```

Program symbols:

```

reduce : term × list → term;
Rabs   : list × term → term;
Rvar   : list × term → term;
sub    : term × term → term;
subs   : term × term × nat → term
update : term × nat × nat → term

```

Reduction rules:

$$\text{reduce}(\text{app } x \ y) \ L := \text{reduce } x \ (\text{cons } y \ L); \quad (1)$$

$$\text{reduce}(\text{abs } f) \ L := \mathbf{Rabs} \ L \ f; \quad (2)$$

$$\mathbf{reduce}(\mathbf{var} \ n) \ L := \mathbf{Rvar} \ L \ (\mathbf{var} \ n); \quad (3)$$

$$\mathbf{Rabs} \ \mathbf{nil} \ f := \mathbf{abs}(\mathbf{reduce} \ f \ \mathbf{nil}); \quad (4)$$

$$\mathbf{Rabs}(\mathbf{cons} \ y \ L) \ f := \mathbf{reduce}(\mathbf{sub} \ f \ y) \ L; \quad (5)$$

$$\mathbf{Rvar} \ \mathbf{nil} \ u := u; \quad (6)$$

$$\mathbf{Rvar}(\mathbf{cons} \ y \ L) \ u := \mathbf{Rvar} \ L(\mathbf{app} \ u(\mathbf{reduce} \ y \ \mathbf{nil})) \quad (7)$$

The purpose of $\mathbf{sub} \ f \ y$ in equation (5) is to find the β -reduct of $\mathbf{abs} \ f$ applied to y . We set $\mathbf{sub} = \lambda f y. \mathbf{subs} \ f \ y \ 0$ where \mathbf{subs} is defined as follows:

$$\mathbf{subs}(\mathbf{app} \ u \ x) \ x \ m := \mathbf{app}(\mathbf{subs} \ u \ x \ m)(\mathbf{subs} \ x \ x \ m); \quad (8)$$

$$\mathbf{subs}(\mathbf{abs} \ u) \ x \ m := \mathbf{abs}(\mathbf{subs} \ u \ x \ (m + 1)); \quad (9)$$

$$\mathbf{subs}(\mathbf{var} \ n) \ x \ m := \begin{cases} (\mathbf{update} \ x \ n \ 1) & \text{if } m = n - 1, \\ \mathbf{var}(n - 1) & \text{if } m < n - 1, \\ (\mathbf{var} \ n) & \text{otherwise.} \end{cases} \quad (10)$$

$$\mathbf{update}(\mathbf{app} \ x \ y) \ m \ j := \mathbf{app}(\mathbf{update} \ x \ m \ j)(\mathbf{update} \ y \ m \ j); \quad (11)$$

$$\mathbf{update}(\mathbf{abs} \ x) \ m \ j := \mathbf{abs}(\mathbf{update} \ x \ m \ (j + 1)); \quad (12)$$

$$\mathbf{update}(\mathbf{var} \ n) \ m \ j := \begin{cases} \mathbf{var}(n + m - 1) & \text{if } n \geq j, \\ (\mathbf{var} \ n) & \text{otherwise.} \end{cases} \quad (13)$$

The program \mathbf{update} takes care of updating the de Bruijn indexes of the free variables after the substitution performed by \mathbf{subs} .

We do not enter in the details of the equations for the updating of the de Bruijn indexes since similar equations have already been used by various people, see for instance [1] and [18]. In these papers the authors consider a “lambda calculus with explicit substitutions” using a suitable notation based on de Bruijn indexes. What we do is different: we are not defining a lambda calculus, but rather a program to reduce lambda terms. So in our approach along with the normalization program, we can also define a wealth of other programs on lambda terms, for instance a program to count the variables. The lambda terms, represented in de Bruijn notation, do not reduce by themselves to normal form: they must be given as inputs to the normalization program.

8 Computability of all partial recursive functions

We define a **second order term algebra** as the set of closed normal forms of a given atomic type α over a given signature Σ of second order constructors. So such an algebra can be denoted by the pair (α, Σ) . We have seen that many interesting data structures can be represented by second order term algebras.

Theorem 3. *For every partial recursive function f between two second order term algebras (α_1, Σ_1) and (α_2, Σ_2) , there is a dichotomic set of reduction rules over a second order signature $\Sigma \supset \Sigma_1 \cup \Sigma_2$ which computes f . A similar result holds for functions of several arguments.*

Proof. (Sketch) We take for granted that the result is true for first order algebras (“folklore theorem”), namely when the constant symbols of the signatures have level 1. In the general case the idea is to show that to every second order term algebra we can associate a first order term algebra and a bijection between the two algebras, such that the bijection and its inverse are computable by a dichotomic set of reduction rules over a second order signature. For instance the second order term algebra of type `exp` that we have used to represent untyped lambda terms (Definition 12) can be associated bijectively to the first order term algebra of type `term` which represents lambda terms in de Bruijn notation (Definition 16). For the details of how to translate between the two representations using a dichotomic set of reduction rules see the appendix.

9 Interpretation in the pure untyped lambda calculus

Definition 18. *An interpretation of $\Lambda(\Sigma, P)$ into the untyped lambda calculus is a map ϕ which assigns to every closed term t of $\Lambda(\Sigma, P)$ a closed term t^ϕ of the untyped lambda calculus and has the following properties:*

1. $(tt_1 \dots t_n)^\phi \equiv t^\phi t_1^\phi \dots t_n^\phi$, $(\lambda x_1 \dots x_n. t)^\phi \equiv \lambda x_1 \dots x_n. (t^\phi)$. So ϕ is uniquely determined by its restriction $\phi|_\Sigma$ to the symbols of the signature.
2. If $t_1 \rightarrow t_2$ in $\Lambda(\Sigma, P)$, then $t_1^\phi \rightarrow t_2^\phi$ in the untyped lambda calculus.

The above definition admits many variants. A weaker notion is obtained by replacing the reduction relation by the convertibility relation in clause 2. A stronger version is obtained by requiring that the converse implication of clause 2 also holds. A reasonable compromise is to require that the interpretation is injective on the data structures. This can be formalized as follows:

Definition 19. *An interpretation ϕ of $\Lambda(\Sigma, P)$ into the untyped lambda calculus is injective on the data structures if whenever t_1 and t_2 are distinct normal forms of $\Lambda(\Sigma, P)$ having atomic type and not containing program symbols, then t_1^ϕ and t_2^ϕ have distinct normal forms.*

Using a technique introduced by Böhm and Piperno and studied in [9, 6] one can prove the following theorem:

Theorem 4. *$\Lambda(\Sigma, P)$ can be interpreted in the untyped lambda calculus by an interpretation that is injective on the data structures.*

Quite remarkably the interpretation described in [9, 6] does not make use of the fixed point combinator. Using this fact it is shown in [6] that in the first order case the interpretation “preserves strong normalization”.

A different interpretation of an higher order term algebra into the pure lambda calculus is obtained by replacing the constructors by variables and abstracting them, as in the definition of the Church numerals (see [7, 8, 17] for similar proposals).

Consider for instance the higher order algebra of type `exp` in Definition 12. The term $\lceil (\lambda x.xx)(\lambda x.xx) \rceil = \mathbf{App}(\mathbf{Abs}(\lambda x.\mathbf{App} \ x \ x))(\mathbf{Abs}(\lambda x.\mathbf{App} \ x \ x))$ is an element of that algebra. By replacing the constructors `App`, `Abs` by variables a, b and abstracting them, we obtain the lambda term $\lambda a b.a(b(\lambda x.a \ x \ x))(b(\lambda x.a \ x \ x))$ (which is a normal form).

In this way we have defined an embedding $t \mapsto \theta(t)$ of the pure lambda calculus into itself (e.g. θ sends $(\lambda x.xx)(\lambda x.xx)$ into $\lambda a b.a(b(\lambda x.a \ x \ x))(b(\lambda x.a \ x \ x))$) which is probably the simplest “gödelization” of the lambda calculus which has ever been considered (compare with [17, 6, 3]). The name “gödelization” is justified by the fact that there is a combinator which defines a bijection from the image of θ onto the Church numerals. This can be easily proved applying the interpretation of [9, 6] to the translations between first order and higher order term algebras given in the appendix. We also need the fact that all infinite first order algebras (suitably embedded in the lambda calculus) admit a lambda definable bijection onto the Church numerals. Note that the image of θ consists of typable terms of type $(\mathbf{exp} \times \mathbf{exp} \rightarrow \mathbf{exp}) \rightarrow ((\mathbf{exp} \rightarrow \mathbf{exp}) \rightarrow \mathbf{exp}) \rightarrow \mathbf{exp}$.

10 Appendix

The program $\mathbf{M} : \mathbf{term} \rightarrow \mathbf{exp}$ below, translates from de Bruijn notation as in Definition 16, to lambda terms represented as in Definition 12. We need an auxiliary program $\mathbf{ch} : \mathbf{term} \times \mathbf{term} \times \mathbf{nat} \rightarrow \mathbf{term}$ and an auxiliary constructor $\mathbf{Bx} : \mathbf{exp} \rightarrow \mathbf{term}$.

$$\mathbf{M}(\mathbf{abs} \ t) := \mathbf{Abs}(\lambda u.\mathbf{M}(\mathbf{ch} \ t(\mathbf{Bx} \ u)1)) \quad (1)$$

$$\mathbf{M}(\mathbf{app} \ x \ y) := \mathbf{App}(\mathbf{M} \ x)(\mathbf{M} \ y) \quad (2)$$

$$\mathbf{M}(\mathbf{Bx} \ u) := u \quad (3)$$

$$\mathbf{ch}(\mathbf{app} \ x \ y) \ u \ j := \mathbf{App}(\mathbf{ch} \ x \ u \ j)(\mathbf{ch} \ y \ u \ j) \quad (4)$$

$$\mathbf{ch}(\mathbf{abs} \ x) \ u \ j := \mathbf{Abs}(\mathbf{ch} \ x \ u(1 + j)) \quad (5)$$

$$\mathbf{ch}(\mathbf{var} \ m) \ u \ j := \text{if } m = j \text{ then } u \text{ else } (\mathbf{Var} \ m) \quad (6)$$

$$\mathbf{ch}(\mathbf{Bx} \ x) \ u \ j := \mathbf{Bx} \ x \quad (7)$$

We now define a term $\lambda x.\mathbf{db} \ x \ 0 : \mathbf{exp} \rightarrow \mathbf{term}$ which performs the inverse translation. The program $\mathbf{db} : \mathbf{exp} \times \mathbf{nat} \rightarrow \mathbf{term}$ uses the auxiliary constructor $\mathbf{Var} : \mathbf{nat} \rightarrow \mathbf{exp}$ (not to be confused with $\mathbf{var} : \mathbf{nat} \rightarrow \mathbf{term}$).

$$\mathbf{db}(\mathbf{Abs} \ t) \ n := \mathbf{abs}(\mathbf{db} \ (t(\mathbf{Var} \ n))(1 + n)) \quad (8)$$

$$\mathbf{db}(\mathbf{App} \ x \ y) \ n := \mathbf{app}(\mathbf{db} \ x \ n)(\mathbf{db} \ y \ n) \quad (9)$$

$$\mathbf{db}(\mathbf{Var} \ m) \ n := \mathbf{var}(n - m) \quad (10)$$

References

1. M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4): 375–416, 1991

2. A. Asperti and C. Laneve. Interaction Systems I, The Theory of Optimal Reductions. *Mathematical Structures in Computer Science*, 4(4): 457–504, 1995
3. H. Barendregt. Self-interpretation in lambda calculus. *Journal of Functional Programming*, 1(2): 229–233, 1991
4. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48: 931–940, 1983
5. S. Bellantoni and S. Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2: 97–110, 1992
6. A. Berarducci and C. Böhm. A self-interpreter of lambda calculus having a normal form. 6th Workshop, CSL '92, San Miniato, Italy, *E. Börger & al. eds. LNCS 702*: 85–99, Springer-Verlag, 1992
7. C. Böhm and A. Berarducci. Automatic synthesis of typed λ -programs on term algebras. *Theoretical Computer Science* 39: 135–154, 1985
8. C. Böhm. Fixed Point Equations Inside the Algebra of Normal Form. *Fundamenta Informaticae*, 37(4): 329–342, 1999
9. C. Böhm, A. Piperno and S. Guerrini. Lambda-definition of function(al)s by normal forms. In: *ESOP'94, LNCS 788*: 135–149, Springer-Verlag, 1994
10. S. Byun, R. Kennaway, R. Sleep. Lambda-definable term rewriting systems. Second Asian Computing Science Conference, ASIAN '96, Singapore, December 2-5, 1996, *LNCS 1179*:105–115, Springer-Verlag, 1996
11. R. L. Constable and S. F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121: 89–112, 1993
12. J. Despeyroux, F. Pfenning, C. Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. In: *R. Hindley, ed., Proc. TLCA '97 Conf., LNCS 1210*: 147–163, Springer-Verlag, 1997
13. M. Gabbay, A. Pitts. A New Approach to Abstract Syntax Involving Binders. In: *Proc. 14th Symp. Logic in Comp. Sci. (LICS) Trento, Italy*: 214–224, IEEE, Washington, 1999.
14. R. Harper, F. Honsell, G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1): 143–184, 1993
15. Y. Lafont. Interaction Combinators. *Information and Computation* 137 (1): 69–101, 1997
16. D. Leivant, J.-Y. Marion. Lambda calculus characterizations of polytime. *Fundamenta Informaticae*, 19: 167–184, 1993
17. T. Æ. Mogensen. Efficient Self-Interpretation in Lambda Calculus. *Journal of Functional Programming*, 2(3): 354–364, 1992
18. M. Ayala-Rincón, F. Kamareddine. Unification via λs_e -Style of Explicit Substitution. In: *International Conference on Principles and Practice of Declarative Programming, PPDP'00, ACM Publications*:163–174, 2000
19. S. Ronchi della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984