Abstract

We present here a generalization of $A$-translation to a class of pure type systems. We apply this translation to give a direct proof of the existence of a looping combinator in a large class of inconsistent type systems, a class which includes type systems with a type of all types. This is the first non-automated solution to this problem.

Capsule review

Consider a pure type system (PTS) extending $\lambda 2$ (see [Barendregt, 1991]). Under mild conditions concerning the PTS in question the paper gives an algorithm that transforms any proof $M$ of a contradiction (i.e. such that $\vdash M : \forall \alpha. \alpha$) into a looping combinator, i.e. a term $Y'$ such that $\vdash Y' : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha$ and $\text{BT}(Y') = \text{BT}(Y)$, where $Y$ is (the usual fixed point combinator, and $\text{BT}(P)$ is the Böhm tree of a lambda term $P$. This extends a previous result (Howe, 1987) in which a concrete proof of a contradiction in a particular PTS is translated with the help of a computer into a looping combinator.

1 Introduction

The term $A$-translation first appeared in a paper by Friedman (1978). It denotes there a technical tool used in a proof of closure under Markov’s rule of several intuitive systems. Combined with Gödel’s translation from classical arithmetic into intuitive arithmetic, this was used to give a new proof of the intuitive probability of classically provable $\Sigma_0^1$ formulas.

Leivant (1985) is a good reference on $A$-translation. Recently, connections between $A$-translation and continuation passing style have been investigated (see, for instance, Murthy’s (1990) PhD thesis).

We are going to generalize $A$-translation to a large class of pure type systems, introduced recently by Barendregt (1991) and Geuvers and Nederhof (1991). This

* This work has been supported by the CNRS “Programmation” Coordinated Research Program, and by the “Logical Framework” Esprit Basic Research Action.
generalization is motivated by the following problem: to extract constructive information from paradoxes in inconsistent type systems. More specifically, let us define a ‘looping combinator’ as being a term having the same Böhm tree as the fixed-point combinator $Y$. It has been shown by Howe (1987) that a type system with a tree of all types contain a looping combinator. We obtain this result as an application of $A$-translation for pure type systems.

The basic idea motivating this can be traced back to the earliest known translation from classical logic to intuitive logic due to Kolmogorov (1967). This translation was actually a translation of classical logic into minimal logic: the rule $ab falso quodlibet$ is never used, and the absurd proposition $\bot$ in Kolmogorov's paper can thus be replaced formally by any proposition $A$. Kolmogorov saw the use of his translation as the development of ‘pseudo-mathematics’, where, intuitively speaking, all notions and all lemmas occurring in a proof are defined and proved ‘relative to a fixed proposition $A$’.

This is this feature of $A$-translation that we essentially use here. In general, it is hard to see how to transform a paradox into a looping combinator. Howe's (1987) argument is rather involved, is done with computer assistance, and shows only how to extract a looping combinator out of one specific paradox. Our approach is more general. We show how to build a looping combinator from any given paradox. Indeed, when we apply $A$-translation to a paradox, we get a proof of $A$ where all notions and lemmas are defined and proved ‘relative to $A$’. This proof is then transformed without too many problems into a looping combinator.

The first section defines a class of ‘logical’ pure type systems in which we will define an $A$-translation. Section 3 describes the $A$-translation for logical pure type systems. We then state a significant property of proofs obtained from the $A$-translation in section 4. This property is exploited to show the existence of a looping combinator in inconsistent type systems. The last section gives some examples of type systems containing looping combinators. We end by raising some questions suggested by our work.

2 Logical pure type systems

We use here the standard definition of pure type systems from Barendregt (1991) and Geuvers–Nederhof (1991). In particular, we make fairly heavy implicit use of the general properties of pure types systems as presented in Geuvers–Nederhof (1991).

**Definition 1**

A pure type system $L$ is **logical** iff it is functional (see Geuvers and Nederhof, 1991) and contains two distinguished sorts Prop and Type, such that

- Prop: Type is an axiom of $L$
- $(\text{Prop}, \text{Prop}, \text{Prop})$ is a rule of $L$
- There are no sorts of type Prop

In a logical pure type system, the terms of type Prop are called **propositions**, and the terms of type a proposition are called **proofs**.
Definition 2
A logical pure type system is **inconsistent** iff there exists a proof of $A$ in the context $A: \text{Proof}$.

Definition 3
A logical pure type system is said to be **nondependent** iff the only rules concerning $\text{Prop}$ are of the form $(S, \text{Prop}, \text{Prop})$ where $S$ is some sort.

**Remark**
Simply-typed $\lambda$-calculus, system $F,F_m$ (see Geuvers and Nederhof, 1991) are non-dependent logical pure type systems. On the other hand, a type system with a type of all types, with $\text{Prop} = \text{Type}$ is not logical because $\text{Prop}$ is then a sort of type $\text{Prop}$. The calculus of constructions is logical, but is not non-dependent because it has the rule $(\text{Prop}, \text{Type}, \text{Type})$.

**Lemma 1**
In a non-dependent logical pure type system, if $X = (X_1 X_2)$ and $X_1$ or $X_2$ is a proof, then $X$ is a proof.

**Proof**
There exist $Y_1$, $Y_2$, $S_1$, $S_2$ and $S$ such that $X_1: (x_2: Y_2) Y$, $X_2: Y_2$, $Y: S$, $Y_2: S_2$ and $(S_2, S, S_1)$ is a rule. If $X_1$ is a proof, then $S_1 = \text{Prop} \Rightarrow \text{Prop}$ and so $S = \text{Prop}$. If $X_2$ is a proof, then $S_2 = \text{Prop}$, and so $S_1 = S = \text{Prop}$.

**Lemma 2**
In a non-dependent logical pure type system, if $Y$ is a subterm of $X$ and $Y$ is a proof, then $X$ is a proof.

**Proof**
By induction on the term $X$. We can also assume that $Y$ is a subterm of $X$ distinct from $X$.

In such a case, the term $X$ cannot be a variable, a constant:

- if $X$ is $\lambda x: X_1. X_2$ then, by induction hypothesis, since $X_1$ is not a proof, $Y$ is a subterm of $X_2$, and hence by induction hypothesis, $X_2$ is a proof. Hence $X$ is a proof.
- if $X$ is $(X_1 X_2)$ then by induction hypothesis, $X_1$ or $X_2$ is a proof. By lemma 1, this implies that $X$ is a proof.

The case where $X$ is a product is impossible by induction hypothesis.

**Remark**
This lemma implies that if $C: \text{Prop}$ in a context containing the declaration of a proof variable $h: B$, then $h$ is not a subterm of $C$. Thus, any product $\Pi h: B. C$ built from the rule $(\text{Prop}, \text{Prop}, \text{Prop})$ is non-dependent and can be written $B \rightarrow C$. 

https://doi.org/10.1017/S0956796800000952 Published online by Cambridge University Press
Lemma 3
Let $L$ be non-dependent logical pure type system and $p$ a proof in a context $\Gamma$. Then $p$ is either a variable of the context, a constant, or $\lambda x: X. q$ where $q$ is a proof in $\Gamma, x: X$, or $(q X)$ where $q$ is a proof in $\Gamma$. 

Proof
Direct by case analysis.

3 $A$-translation in non-dependent logical pure type systems

In all the sections we assume a fixed, non-dependent logical pure type system, and inside the context of $A: \text{Prop}.$

Notation
Let $B$ be a proposition. We write $[B]$ for the proposition $(B \rightarrow A) \rightarrow A$.

We now define a translation $+$ on terms which are not proofs. This translation depends on the type of the subterms, and it is defined relative to a context in which the term is well-formed. Notice that it is not clear a priori that $M^+$ is a well-formed term, so that a priori $M^+$ is defined only as a pseudo-term (see Geuvers and Nederhof, 1991). Proposition 1 will later show that $M^+$ is actually a well-formed term.

Definition
Let $X$ be a well-formed term in the context $\Gamma$, different from a proof:

- $X^+$ is $X$ if $X$ is a variable, a constant or a sort
- $(X_1, X_2)^+$ is $(X_1^+ \cdot X_2^+)$
- $(\lambda x: X_1. X_2)^+$ is $\lambda x: X_1^+ . X_2^+$
  where $X_1^+$ is defined in $\Gamma, x: X_1$
- the definition of $(\Pi x: X_1. X_2)^+$ depends on the type of $X_2$ and $X_1$:
  if $X_2$ is a proposition $B_2$ in $\Gamma$
  then if $X_1$ is a proposition $B_1$ in $\Gamma$
    then $(B_1 \rightarrow B_2)^+$ is $[B_1^+] \rightarrow [B_2^+]$
    else $(\Pi x: X_1. B_2)^+$ is $\Pi x: X_1^+ . [B_2^+]$,
      where $B_2^+$ is defined in $\Gamma, x: X_1$
  else $(\Pi x: X_1. X_2)^+$ is $\Pi x: X_1^+ . X_2^+$
      where $X_2^+$ is defined in $\Gamma, x: X_1$

Remark
Lemma 3 justifies the previous definition by cases.

Lemma 4
For any terms $X$ well-formed in $\Gamma, y: Y$ and $Z$ well-formed in $\Gamma$ different from proofs, then $(X[y:= Z])^+$ is identical to $X^+[y:= Z^+]$. 

https://doi.org/10.1017/S0956796800000952 Published online by Cambridge University Press
Proof
It is straightforward. □

Lemma 5
For any terms $X$ and $Y$ well-formed in $\Gamma$ different from proofs, $X =_\beta Y$ implies $X^+ =_\beta Y^+$.

Proof
It suffices to prove that $((\lambda z : Z. Z' Z'')^+)$ reduces to $(Z'[z := Z''])^+$. This follows from lemma 4. □

We now define a translation $\ast$ on propositions and contexts

Definitions
Let $B$ be a proposition in a certain context, $B^*$ is defined as $[B^+]$. Let $T$ be a well-formed context, $T^*$ is defined inductively as:

- if $T$ is the empty context then $T^*$ is the empty context
- if $T$ is $T', x : X$, where $X$ is not a proposition then $T^*$ is $T'^*, x : X^+$
- if $T$ is $T', h : B$, where $B$ is a proposition then $T^*$ is $T'^*, h : B^*$

Lemma 6
For any propositions $B$ and $C$ in $\Gamma$, $B =_\beta C$ implies $B^* =_\beta C^*$.

Proof
Straightforward by lemma 5. □

Proposition 1
If $T \vdash X : Y$ and $X$ is not a proof then $T'^* \vdash X^+ : Y^+$. If $T \vdash B : Prop$ then $T'^* \vdash B^* : Prop$.

Proof
We prove this simultaneously by induction on the structure of the derivation of $T \vdash X : Y$ (resp. $T \vdash B : Prop$). The case of conversion is done by lemma 5. Lemma 2 assures us that the derivation of $T \vdash X : Y$ (resp. $T \vdash B : Prop$) encounters no proofs. □

Lemma 7
For any propositions $B$ and $C$ in $\Gamma$, if $T^* \vdash p : B^*$ and $B =_\beta C$ then $T'^* \vdash p : C^*$.

Proof
By lemma 6 and the conversion rule in pure type systems. Proposition 1 assures that $T'^* \vdash C^* : Prop$. □

We now define translation $\ast$ on proofs. As for the translation $\ast$, it is defined relative to a context in which the term is a well-formed proof $p$, and it is not clear a priori
that $p^*$ is a well-formed term, so that $p^*$ is defined only as a pseudo-term. Theorem 1 will actually show that $p^*$ is indeed a well-formed term which is a proof.

**Definition**

Let $p$ be a proof in the context $\Gamma$:

- $p^*$ is $p$ if $p$ is a variable or a constant
- if $p$ is $\lambda h: B.q$, with $B$ a proposition, and $q: C$, then $p^*$ is $\lambda k: ((B^* \to C^*) \to A). (k \lambda h: B^*. \lambda k': (C^* \to A). (q^* k'))$ where $q^*$ is defined in $\Gamma$, $h: B$
- if $p$ is $\lambda k: Y.q$, with $Y$ not a proposition, and $q: C$, then $p^*$ is $\lambda k: ((\Pi x: Y. C^*) \to A). (k \lambda x: Y. \lambda k': (C^* \to A). (q^* k'))$ where $q^*$ is defined in $\Gamma$, $x: Y$
- if $p$ is $(p_1 p_2)$ and $p_1: B \to C$ then $p^*$ is $\lambda k: (C^* \to A). (p_1^* \lambda h_1: (B^* \to C^*). (h_1 p_2^* k))$
- if $p$ is $(p, X)$, when $X$ is not a proof, and $p_1: \Pi x: Y. C$, then $p^*$ is $\lambda k: (C[x := X]^* \to A). (p_1^* \lambda h_1: (\Pi x: Y^+. C^*). (h_1 X^* k))$

**Remark**

Lemma 3 justified the previous definition by cases.

**Theorem 1**

Let $B$ be a proposition in $\Gamma$. If $\Gamma \vdash p: B$ then $\Gamma^* \vdash p^*: B$

**Proof**

By induction on the structure of the derivation of $\Gamma \vdash p: B$. The case of proposition conversion is done by lemma 7. Proposition 1 treats the case of judgements $\Gamma \vdash X: Y$ with $X$ not a proof. □

**Remark 1**

$^*$ is a Kolmogorov-like $\Lambda$-translation. It generalizes an $\Lambda$-translation of Paulin–Mohring (1989) for the Calculus of Constructions with data types distinguished from propositions, and is inspired by a classical/intuitionistic translation of Girard (1972) for higher order $\lambda$-calculi.

**Remark 2**

If we assume Church–Rosser property for the pure type system we are considering, lemma 5 holds also for $\beta\eta$-conversion, and therefore proposition 1 and theorem 1 still hold in presence of $\beta\eta$-conversion. However, the Church–Rosser property for general pure type systems (not necessarily normalizable) with $\beta\eta$-conversion still seems to be an open problem.

## 4 Long $\Lambda$-applicativity

As we said in the introduction, the original motivation in using $\Lambda$-translation was the fact that, intuitively, proofs that get by $\Lambda$-translation 'proves only $\Lambda$'. Trying to make precise this remark leads to the following notion.
A-translation and looping combinators

Definition

The notion of long \( A \)-applicative proof in a context \( \Gamma \) is defined by the following cases:

- the variable \( h \) of type \( B \) with \( B : \text{Prop} \) is a long \( A \)-applicative proof if \( h : B \) is in \( \Gamma \)
- \( \lambda x_1 : Y_1 \ldots \lambda x_n : Y_n . p \) is a long \( A \)-applicative proof in \( \Gamma \) if \( p \) is a long \( A \)-applicative proof in \( \Gamma, x_1 : Y_1, \ldots, x_n : Y_n \) and if \( p \) is of type \( A \)
- \( (pq) \) is a long \( A \)-applicative proof in \( \Gamma \) if \( p \) and \( q \) are long \( A \)-applicative proofs in \( \Gamma \)
- \( (pX) \) where \( X \) is not a proof is a long \( A \)-applicative proof in \( \Gamma \) if \( p \) is a long \( A \)-applicative proof in \( \Gamma \).

Proposition 2

If \( p \) is a proof in \( \Gamma \) then \( p^* \) is long \( A \)-applicative in \( \Gamma^* \).

Proof

Direct from the definition of \( p^* \). \( \square \)

Lemma 8

If \( p \) is a long \( A \)-applicative proof in a context \( \Gamma, h : B \) and \( q \) is long \( A \)-applicative in \( \Gamma \) then \( p[h := q] \) is long \( A \)-applicative in \( \Gamma \).

If \( p \) is a long \( A \)-applicative proof in a context \( \Gamma, x : Y \) and \( X \) is not a proof in \( \Gamma \) then \( p[x := X] \) is long \( A \)-applicative in \( \Gamma \).

Proof

By induction on the structure of \( p \). \( \square \)

5 Looping combinators

The idea of Meyer and Reinhold (1986) to obtain a recursion combinator in the inconsistent system \( \text{Type}: \text{Type} \) was to exploit the non-normalizability of the proof of the inconsistency by inserting some ‘\( f \)’ in it to obtain a term \( p_0 \), such that \( p_0 \) reduces to \( (fp_1) \) and then \( p_1 \) to \( (fp_2) \), and so on. From such a sequence, it is straightforward to build a family of terms \( Y_n : \Pi A : \text{Type}.(A \to A) \to A \) such that \( (Y_nAf) = f(Y_{n+1}Af) \).

Definition

Let \( T \) be a pure type system and \( S \) a sort of \( T \). A looping combinator of sort \( S \) in \( T \) is a term \( Y : \Pi A : S.(A \to A) \to A \) such that there exists a sequence of terms \( Y_0 = Y, Y_1, \ldots, Y_n, \ldots, \) of type \( \Pi A : S.(A \to A) \to A \) such that for any \( A : S, f : A \to A \)

\[
(Y_nAf) = \beta f(Y_{n+1}Af)
\]

Howe (1987) applied the same idea to transform the paradox of Girard (1972) into a looping combinator by a direct mechanical analysis of the term corresponding to this paradox.

We now show how to build a looping combinator in any inconsistent nondependent logical pure type system. The last section will show that this in particular implies the existence of a looping combinator also for \( \text{Type}: \text{Type} \).
From now we assume a fixed, inconsistent, nondependent, logical pure type system inside the context $A : \text{Prop}$.

Proposition 3
There exists a long $A$-applicative proof of $A$.

Proof
Since the type system is inconsistent, there exists a proof $q$ of $A$ in the context $A : \text{Prop}$. By theorem 1, $(q_A)^*$ is a proof of $A^*$ in the context $A : \text{Prop}$, and by proposition 2, this proof is long $A$-applicative. But $A^*$ is $(A \rightarrow A) \rightarrow A$, and $p_A = (q_A)^* \lambda x : A . x$ is a long $A$-applicative proof of $A$. □

We now precise what kind of term is a long $A$-applicative proof of $A$:

Lemma 9
A long $A$-applicative proof of $A$ is of the following form:

$$(\lambda \lambda^1 : Y^1 \ldots \lambda \lambda^m : Y^m . q) X^1 \ldots X^m$$

with $m \geq 1$, $q : A$ and each $X^i$ is either long $A$-applicative or not a proof.

Proof
Let $p$ be a long $A$-applicative proof of $A$ in the context $A : \text{Prop}$. Since $A$ is atomic, $A$ cannot be convertible to a product by Church–Rosser. Hence, by uniqueness of type, $p$ does not begin with an abstraction.

Therefore, it is of the following form:

$$(p' X^1 \ldots X^m)$$

with $m \geq 0$ and $p'$ either a variable or an abstraction.

Since we are in the context $A : \text{Prop}$, the term $p'$ cannot be a variable, $m \geq 1$ and $p'$ begins with an abstraction. And since $p$ is long $A$-applicative, $p'$ is of the following form:

$$\lambda \lambda^1 : Y^1 \ldots \lambda \lambda^m : Y^m . q$$

with $m' \geq 1$ and $q : A$.

The type of $q$ remains $A$ by instantiation, hence $m$ cannot be greater than $m'$. And since $p$ proves $A$, $m'$ cannot be greater than $m$. Hence, we have $m = m'$, i.e. $p$ has the desired form. □

We now define a strategy of reduction applicable to long $A$-applicative proofs of type $A$.

Definition
Let $p$ be a long $A$-applicative proofs of type $A$. By lemma 9, $p$ is

$$(\lambda \lambda x_1 : Y_1 \ldots \lambda x_n : Y_n . q) X_1 \ldots X_n)$$

with $n \geq 1$ and $q : A$,

$\text{red}(p)$ is then the following term of type $A$

$$q[x_1 := X_1] \ldots [x_n := X_n].$$
Lemma 10
For any long A-applicative proof p of A in A: Prop, red(p) is a long A-applicative proof of A in A: Prop.

Proof
By lemma 8. □

We now define the transformation p' which inserts ‘marks’ inside long A-applicative proofs p in such a way that for any long A-applicative proofs p of A red(p') is (f(red(p))').

Definition
Let p be a long A-applicative proof in a context Γ, p' is defined inductively in the context Γ, f: A → A as follows:

1. If p is a variable h in Γ then p' is h in Γ,
2. If p is λ x₁: Y₁...λ xₙ: Yₙ.q in Γ then p' is λ x₁: Y₁...λ xₙ: Yₙ.(f q') in Γ where q' is defined in Γ, f: A → A, x₁: Y₁,...,xₙ: Yₙ,
3. If p is (p₁,p₂) then p' is (p₁',p₂'),
4. If p is (p₁ M) with M not a proof, then p' is (p₁,M).

Remark
p' is of same type as p and is also long A-applicative.

Lemma 11
If p is an A-applicative proof in the context Γ, h: B and q an A-applicative proof of B in Γ then p'[h ← q'] is (p[h ← q])'.
If p is an A-applicative proof in the context Γ, R: T and M: T not a proof then p'[R ← M] is (p[R ← M])'.

Proof
By structural induction on p and by lemma 8. □

Lemma 12
For any long A-applicative proof p of A, red(p') is (f(red(p))').

Proof
p is of the form
((λ x₁: Y₁...λ xₙ: Yₙ.q) X₁...Xₙ),
and then p' is
((λ x₁: Y₁...λ xₙ: Yₙ.(f q'))(X₁)'...((Xₙ)')), which reduces by lemma 11 to (f(red(p))'). □

Lemma 13
There exists a sequence of terms M₀,M₁,...,Mₙ,...defined in the context A: Prop, f: A → A such that Mₙ = β (f Mₙ₊₁).
Proof
We define a sequence of terms $p_n$ as follows. First, we define $p_0$ to be any long $A$-applicative proof of $A$ in the context $A : \text{Prop}$, using proposition 3. We then define $p_{n+1}$ to be $\text{red}(p_n)$. Each proof term $p_n$ is long $A$-applicative proof of $A$ in $A : \text{Prop}$ by lemma 10.

Let $M_n$ be $p_n^\prime$. The sequence $M_0, \ldots, M_n, \ldots$ satisfies lemma 13 by lemma 12.

Theorem 2
In any inconsistent non-dependent logical pure type system, there exists a looping combinator of type $\text{Prop}$.

Proof
Direct from lemma 13.

Remark
The proof given here is constructive. We can effectively transform any proof of $A$ in the context $A : \text{Prop}$ into a looping combinator.

6 Applications
We describe here the systems $U^-, U$ and $\text{Type} : \text{Type}$ as pure type systems.

The system $U^-$ is the pure type system defined by the sorts $\text{Prop}$, $\text{Type}$ and $\text{Class}$, the axioms $\text{Prop} : \text{Type}$ and $\text{Type} : \text{Class}$, and the rules:

$(\text{Prop}, \text{Prop}, \text{Prop})$
$(\text{Type}, \text{Prop}, \text{Prop})$
$(\text{Type}, \text{Type}, \text{Type})$
$(\text{Class}, \text{Type}, \text{Type}).$

System $U$ is the same as system $U^-$, plus the following rule:

$(\text{Class}, \text{Prop}, \text{Prop}).$

The system $\text{Type} : \text{Type}$ is the pure type system with only the sort $\text{Type}$, only the axiom $\text{Type} : \text{Type}$, and only the rule $(\text{Type}, \text{Type}, \text{Type})$.

Both $U$ and $U^-$ systems are non-dependent logical pure type system. They are both inconsistent, as shown in Girard (1972) and Coquand (1991). Hence, by theorem 2, they contain a looping combinator of the sort $\text{Prop}$. It is clear that a looping combinator for one of these systems translates directly in a looping combinator of sort $\text{Type}$ for $\text{Type} : \text{Type}$.

Here is a direct application. Call a non-dependent logical type system impredicative iff it contains the rule $(\text{Type}, \text{Prop}, \text{Prop})$.

Theorem 3
Convertibility is undecidable for inconsistent impredicative logical pure type system. Furthermore, convertibility and type-checking is undecidable for $\text{Type} : \text{Type}$. 

https://doi.org/10.1017/S0956796800000952 Published online by Cambridge University Press
The arguments of Meyer and Reinhold (1986), which assumed the existence of a fixed-point combinator, apply directly using a looping combinator instead.

For the sake of completeness, we include a sketch of these arguments. First, it is standard (Girard, 1972) how to represent primitive recursive functions as terms of type $N \rightarrow N$, where $N$ is the proposition $\Pi X. X \rightarrow (X \rightarrow X) \rightarrow X$, and the number $n$ is represented by the term $\lambda X. \lambda x. \lambda f. (f^n x)$. A looping combinator family allows the numeralwise representation of any partial recursive function $\phi$ by a term $\Phi$: namely $\Phi t_n = \beta t_k$ iff $\phi(n) = k$. This entails the undecidability of convertibility in any inconsistent impredicative logical pure types system.

The same reasoning will apply to $Type: Type$ by taking $N$ to be the type $\Pi X. X \rightarrow (X \rightarrow X) \rightarrow X$. Furthermore, in this case the problem of whether $\phi(n) = 0$ reduces to the question whether $(f x)$ is typable in the context $P: N \rightarrow Type, f: P(t_0) \rightarrow N, x: P(\Phi(t_n))$. Likewise, checking specific type judgements is undecidable, since $\phi(n) = 0$ reduces to the question whether $x$ has type $P(\Phi(t_n))$ in the context $P: N \rightarrow Type, x: P(t_0)$. □

Notice, however, that the normalization theorem for system $F$ (Girard, 1972) directly implies the decidability of type-checking for the $U^-$ and $U$ systems.

7 Conclusion

We would like to highlight some problems:

- The problem of the existence of a fixed-point combinator for the $Type: Type$ system still exists.
- Is it possible to derive the existence of a looping combinator from the existence of a paradox in a more direct way than by using $A$-translation?
- For the $U^-$ system it is possible to define a 'stripping' operation that associates to any proof term the untyped $\lambda$-term we get by forgetting the type information. We conjecture that the usual direct proof of non-typability of the term $(\lambda x(x x)) \lambda x(x x)$ in system $F$ extends to show that this term is not typable in the $U^-$ system.

Acknowledgements

The authors want to thank Herman Geuvers, Chet Murthy, Erik Palmgren and Benjamin Werner for their remarks about the paper, and for enjoyable discussions about $A$-translation and pure type systems. Thanks also to Henk Barendregt for his definition of looping combinators.

References


