Under consideration for publication in J. Functional Programming

THEORETICAL PEARL
The non-typability of some fixed-point combinators in Pure Type Systems

HERMAN GEUVERS and JOEP VERKOELEN
Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands
(*e-mail:* herman@cs.ru.nl)

Abstract

The type theories λU and λU^- are known to be logically inconsistent. For λU, this is known as Girard's paradox (Girard, 1972); for λU^- the inconsistency was proved by Coquand (Coquand, 1994). It is also known that the inconsistency gives rise to a so called *looping combinator*: a family of terms L_n such that $L_n f$ is convertible with $f(L_{n+1}f)$. It is unclear whether a fixed point combinator exists in these systems. Hurkens (Hurkens, 1995) has given a simpler version of the paradox in λU^-, giving rise to an actual proof term that can be analyzed, and which is proved not to be a fixed point combinator in (Barthe & Coquand, 2006). However, the underlying untyped term is a real fixed point combinator.

In the present paper we analyze the possibility of typing a fixed point combinator in λU and we prove that the Curry and Turing fixed point combinators Y and Θ cannot be typed in λU.

1 Introduction

This paper deals with the subject of fixed point and looping combinators in typed λ-calculi. We are mainly interested in the systems λU^-, λU and $\lambda\star$, which arose in the early 70s as inconsistent extensions of (typed) higher order logic, following the Curry-Howard formulas-as-types embedding. In a sense, the simplest system is $\lambda\star$, where 'type is a type' and therefore many constructions that are forbidden in other type theories are possible. The system is inconsistent in the sense that there are closed inhabitants of all types, also the 'bottom' type $\Pi\alpha:\star.\alpha$. This makes the system logically inconsistent. Asking whether all untyped terms can be typed is the reverse question: Given an untyped λ-term M, can one add type-information to M, to obtain some term M' that is typable? We show that for λU, the answer no: there are terms, e.g. Ω, that cannot be typed in λU.

Although systems like $\lambda\star$ and λU are logically inconsistent, computationally they are still interesting, because not all terms are β-convertible. The first to study the computational power of these inconsistent systems was (Howe, 1987), going back to earlier (unpublished) work of (Reinhold, 1986). Howe coined the terminology *looping combinator* for a family of terms $\{L_n\}_{n\in\mathbb{N}}$ such that $L_n\, f =_\beta f(L_{n+1}\, f)$, and he showed that a looping combinator can be defined in $\lambda\star$. Using a looping combinator, it can be shown that the equational theory (the theory of β-conversion) is undecidable and that the theory is Turing complete. We have not been able to find a proof of Turing completeness in the published literature, so we outline it briefly in this paper.

When Girard (Girard, 1972) proved the paradox in 1972, he did that for λU, an extension of higher order logic with polymorphic domains and quantification over all domains. This system allows less type constructions than $\lambda\star$, but that has the advantage that it is somewhat easier to see what is going on. By that time, it was unclear whether λU^-: higher order logic with polymorphic domains (but no quantification over all domains) was inconsistent.

In 1994, Coquand (Coquand, 1994) proved that λU^- is inconsistent as well, by encoding Reynold's result (Reynolds, 1984), stating that no set-theoretic model of polymorphic lambda calculus exists, into λU^-. Later, Hurkens gave a considerably shorter proof (Hurkens, 1995), which is based on interpreting Russell's paradox in λU^-. The intuition of the proof given in (Hurkens, 1995) is difficult to follow, because the proof is optimized a lot in order to get a small proof term. A more intuitive proof of the inconsistency of λU^- has been given by Miquel in (Miquel, 2000). It is based on representing sets as pointed graphs in λU^- and then interpreting Russell's paradox.

In the present paper we analyze the paradox in λU syntactically. (For a semantic analysis, relating the paradox to models of higher order logic, see (Geuvers, 2007).) The main question we are interested in is whether there exists a fixed-point combinator in λU. We give a partial answer by showing that the well-known Turing and Curry fixed-point combinators (Θ and Y) cannot be typed in λU. We prove our results for λU, but they also immediately apply to λU^- without a change.

In this article we assume that the reader is familiar with the lambda calculus, both in its untyped form and typed versions for the remainder of the article. For details and background we refer to (Barendregt, 1981; Barendregt, 1992).

2 Untyped Lambda Calculus

In this section we study the expressive power of looping combinators. We will not yet be specific about the type theory, because the expressive power deals with the computation (β-reduction) and not with the typing. So, basically the results in this section can be cast in an untyped setting. First a precise definition of the notion of "looping combinator".

Definition 2.1

Given a type A in our type system, a *fixed point combinator of type* A is a term $Y : (A \to A) \to A$ such that for all $f : A \to A$ we have

$$Y\, f =_\beta f\,(Y\, f).$$

a *looping combinator of type* A is a family of terms $L_n : (A \to A) \to A$ for all natural numbers n, such that for all $f : A \to A$ we have

$$L_n\, f =_\beta f\,(L_{n+1}\, f).$$

Remark 1

We will refer to L_0 as 'the looping combinator', not mentioning the whole family. If L_0 is a looping combinator then for all natural numbers n, the term L_n (in the family $\{L_n\}_{n\in\mathbb{N}}$) is also a looping combinator. Every fixed point combinator is a looping combinator.

In order to represent recursive functions in our type theory, we must be able to represent natural numbers, with a zero element Z, a successor function S and a predecessor function. Furthermore, we must be able to represent booleans with a test-for-zero and an if-then-else construction. (This could also be achieved without booleans by using the natural numbers and Z for true and S Z for false.)

Definition 2.2
A type theory *contains a data type for natural numbers and for booleans* in case the type theory has

- types nat and bool,
- terms $\mathsf{Z} : \mathsf{nat}$, $\mathsf{S} : \mathsf{nat} \to \mathsf{nat}$, $\mathsf{P}^- : \mathsf{nat} \to \mathsf{nat}$ with $\mathsf{P}^-(\mathsf{S}^{n+1}(\mathsf{Z})) =_\beta \mathsf{S}^n(\mathsf{Z})$
- terms $\mathsf{tt}, \mathsf{ff} : \mathsf{bool}$ and $\mathsf{Zero?} : \mathsf{nat} \to \mathsf{bool}$ with $\mathsf{Zero?Z} =_\beta \mathsf{tt}$, $\mathsf{Zero?}(\mathsf{S}x) =_\beta \mathsf{ff}$,
- for $b : \mathsf{bool}$ and $e_1, e_2 : \mathsf{nat}$, a term $\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \mathsf{nat}$ with $\mathsf{if\,tt\,then}\, e_1\, \mathsf{else}\, e_2 =_\beta e_1$ and $\mathsf{if\,ff\,then}\, e_1\, \mathsf{else}\, e_2 =_\beta e_2$.

The *untyped* λ-calculus is *Turing complete*: all recursive functions are definable as λ-terms. The power of the untyped λ calculus lies in the fact that one can *solve recursive equations*, that is, one can positively answer questions of the following kind:

- Is there a term M such that $M\,x =_\beta x\,M\,M$?
- Is there a term N such that $N\,x =_\beta \mathsf{if}\,(\mathsf{Zero?}x)\,\mathsf{then}\,1\,\mathsf{else}\,\mathsf{mult}\,x\,(N\,(\mathsf{P}^-\,x))$?

(Here, mult is some term that defines multiplication.)

In the untyped λ-calculus, these questions can be answered affirmatively because we have a fixed point combinator. If Y is a fixed-point combinator, $M := Y(\lambda m\,\lambda x.x\,m\,m)$ and $N := Y(\lambda n\,\lambda x.\mathsf{if}\,(\mathsf{Zero?}x)\,\mathsf{then}\,1\,\mathsf{else}\,\mathsf{mult}\,x\,(n\,(\mathsf{P}^-\,x)))$ do the job. So, a solution has the form $Y\,F$, where F is the functional that we want to apply repeatedly:

$$N(\mathsf{S}\,p) =_\beta (\lambda n\,\lambda x.\mathsf{if}\,(\mathsf{Zero?}x)\,\mathsf{then}\,1\,\mathsf{else}\,\mathsf{mult}\,x\,(n\,(\mathsf{P}^-\,x)))N\,(\mathsf{S}\,p) =_\beta \mathsf{mult}\,(\mathsf{S}\,p)\,(N\,p).$$

A looping combinator does a similar thing: it allows the repeated application of a functional: $Y_0\,F =_\beta F(Y_1\,F) =_\beta F(F((Y_2\,F)) =_\beta \ldots$. So, using a looping combinator we should also be able to define all recursive functions. However, a looping combinator does not provide a solution to a recursive equation, but an 'almost solution'[1]. Let us make the proof precise that all recursive functions are λ-definable in a type theory with a looping combinator. The original proof of Turing completeness of the untyped lambda calculus can be found in (Kleene, 1936); see also (Barendregt, 1981). The proof below basically appears in the unpublished manuscript (Reinhold, 1986).

Theorem 2.1

[1] The notion of 'almost solution' can be made more precise by observing that a fixed-point combinator and a looping combinator have the same Böhm tree and therefore also $Y_i F$ and $F\,(Y_j\,F)$ have the same Böhm tree for all i, j

In a typed λ-calculus with data types for natural numbers and booleans and a looping combinator L of type nat→nat, all (partial) recursive functions are λ-definable.

The only interesting part is to show that the class of λ-definable functions is closed under primitive recursion and minimization. For $n \in \mathbb{N}$, we use the notation $\overline{n}$ to denote the representation of n as a λ-term. (It may be that $\overline{n}$ is the Church numeral c_n, but we are not committed to a specific representation.)

Lemma 2.1
If we have a looping combinator L of type nat→nat, the λ-definable functions are closed under primitive recursion.

Proof
Let φ be defined by primitive recursion from χ and ψ:

$$\begin{array}{rcl} \varphi(\vec{x},0) & = & \chi(\vec{x}) \\ \varphi(\vec{x},n+1) & = & \psi(\vec{x},n,\varphi(\vec{x},n)) \end{array}$$

and suppose that χ, ψ are lambda-defined by G,H respectively. Define

$$\Phi := \lambda f\,\vec{x}\,n.\mathsf{if}\ (\mathsf{Zero?}\ n)\ \mathsf{then}\ (G\,\vec{x})\ \mathsf{else}\ (H\,\vec{x}\,(P^-\ n)\ (f\,\vec{x}\,(P^-\ n)))$$

We claim that the term $i\ L_i\Phi$ lambda-defines φ for all $i \in \mathbb{N}$. As the computation of $L_0\Phi$ may result in computing $L_1\Phi$, which again may result in computing $L_2\Phi$ etc, it will not work to prove $\forall n(L_i\,\Phi\,\overline{n} =_\beta \overline{\varphi(\vec{x},n)})$ separately for every i. Instead we prove $\forall n \forall i(L_i\,\Phi\,\overline{n} =_\beta \overline{\varphi(\vec{x},n)})$ by induction on n.

Basis: Assume $n = 0$. Given a natural number i we have $L_i\,\Phi\,\vec{x}\,\overline{0} =_\beta \Phi\,(L_{i+1}\,\Phi)\,\vec{x}\,\overline{0} \twoheadrightarrow_\beta$ $\mathsf{if}\ (\mathsf{Zero?}\ \overline{0})\ \mathsf{then}\ (G\,\vec{x})\ \mathsf{else}\ (H\,\vec{x}\,(P^-\ \overline{0})\ ((L_{i+1}\ \Phi)\,\vec{x}\,(P^-\ \overline{0}))) =_\beta G\,\vec{x}$

Induction: Assume that for all j, $L_j\,\Phi\,\vec{x}\,\overline{n} =_\beta \overline{\varphi(\vec{x},n)}$ (IH). Given a natural number i, we have to prove that $L_i\,\Phi\,\vec{x}\,\overline{n+1} =_\beta \overline{\varphi(\vec{x},n+1)}$.

$$\begin{array}{rcl} L_i\,\Phi\,\vec{x}\,\overline{n+1} & =_\beta & \Phi\,(L_{i+1}\,\Phi)\,\vec{x}\,\overline{n+1} \\ & \twoheadrightarrow_\beta & \mathsf{if}\ (\mathsf{Zero?}\ \overline{n+1})\ \mathsf{then}\ (G\,\vec{x})\ \mathsf{else}\ (H\,\vec{x}\,(P^-\ \overline{n+1})\ (L_{i+1}\,\Phi\,\vec{x}\,(P^-\ \overline{n+1}))) \\ & =_\beta & H\,\vec{x}\,\overline{n}\ (L_{i+1}\,\Phi\,\vec{x}\,\overline{n}) \\ & \stackrel{\text{IH}}{=}_\beta & H\,\vec{x}\,\overline{n}\ \overline{\varphi(\vec{x},n)} \\ & =_\beta & \overline{\psi(\vec{x},n,\varphi(\vec{x},n))} \end{array}$$

So we conclude that for all i, $L_i\ \Phi\ \lambda$-defines φ. □

Lemma 2.2
If we have a looping combinator L of type nat→nat, the λ-definable functions are closed under minimization.

Proof
Let φ be defined by $\varphi(\vec{x}) := \mu z[\chi(\vec{x},z) = 0]$, where χ is total and lambda-defined by G. We now need to prove that there is a lambda term F such that

$$\begin{array}{rclll} F\,\vec{x} & =_\beta & \overline{n} & \text{if} & G\,\vec{x}\,\overline{n} =_\beta \overline{0} \text{ and } \forall p < n(G\,\vec{x}\,\overline{p} \neq_\beta \overline{0}) \\ F\,\vec{x} & = & \uparrow & \text{if} & \forall p(G\,\vec{x}\,\overline{p} \neq_\beta \overline{0}) \end{array}$$

Define

$$\begin{array}{rcl} \Phi & \equiv & \lambda h\,\vec{x}\,z.\mathsf{if}\ (\mathsf{Zero?}\ (G\,\vec{x}\,z))\ \mathsf{then}\ z\ \mathsf{else}\ (h\,\vec{x}\,(\mathsf{S}z)) \\ H_i & \equiv & \lambda\vec{x}.\lambda z.L_i\,\Phi\,\vec{x}\,z \end{array}$$

Now we have

$$\begin{array}{rcl} H_i\,\vec{x}\,z & =_\beta & L_i\,\Phi\,\vec{x}\,z \\ & =_\beta & \Phi\,(L_{i+1}\,\Phi)\,\vec{x}\,z \\ & =_\beta & \mathsf{if}\ (\mathsf{Zero?}\ (G\,\vec{x}\,z))\ \mathsf{then}\ z\ \mathsf{else}\ (L_{i+1}\,\Phi\,\vec{x}\,(\mathsf{S}\,z)) \\ & =_\beta & \mathsf{if}\ (\mathsf{Zero?}\ (G\,\vec{x}\,z))\ \mathsf{then}\ z\ \mathsf{else}\ H_{i+1}\,\vec{x}\,(\mathsf{S}\,z) \end{array}$$

We now consider the value of $H_i\vec{x}\overline{n}$:

- If $\forall p \geq n(G\,\vec{x}\,\overline{p} \neq_\beta \overline{0})$, then $H_i\,\vec{x}\,\overline{n} =_\beta H_{i+k}\,\vec{x}\,\overline{n+k}$ (for all k) and we can prove that $H_i\,\vec{x}\,\overline{n}$ has no normal form.
- If $\forall p(n \leq p < m \to G\,\vec{x}\,\overline{p} \neq_\beta \overline{0})$ and $G\,\vec{x}\,\overline{m} =_\beta \overline{0}$, then $\forall i(H_i\,\vec{x}\,\overline{n} =_\beta m)$.

So, we can take the following term F to λ-define φ: $F := \lambda\vec{x}.H_0\,\vec{x}\,\overline{0}$. (As a matter of fact, one can take any of the terms $\lambda\vec{x}.H_i\,\vec{x}\,\overline{0}$ to λ-define φ.) □

3 Pure Type Systems

The systems we study can all be interpreted as Pure Type Systems (PTS). For a thorough explanation on PTS's see (Barendregt, 1992; Geuvers, 1993; Barendregt & Geuvers, 2001).

Definition 3.1

A *Pure Type System* $\lambda(\mathscr{S},\mathscr{A},\mathscr{R})$ is given by a set $\mathscr{S}$ (of *sorts*), a set $\mathscr{A} \subset \mathscr{S}\times\mathscr{S}$ (of *axioms*), and a set $\mathscr{R} \subset \mathscr{S}\times\mathscr{S}\times\mathscr{S}$ (of *rules*), and is the typed lambda calculus with the reduction rules presented in Fig. 1. We assume $s \in \mathscr{S}$. The elements of $\mathscr{A}$ are written as $s_1 : s_2$ with $s_1, s_2 \in \mathscr{S}$. The elements of $\mathscr{R}$ are written as (s_1,s_2,s_3) with $s_1,s_2,s_3 \in \mathscr{S}$. If $s_2 = s_3$, we write (s_1,s_2) instead.

The expressions in the reduction rules are taken from the set of pseudo-terms $\mathscr{T}$ defined by

$$\mathscr{T} := \mathscr{S} \mid \mathscr{V} \mid (\Pi\mathscr{V}{:}\mathscr{T}.\mathscr{T}) \mid (\lambda\mathscr{V}{:}\mathscr{T}.\mathscr{T}) \mid \mathscr{T}\,\mathscr{T}$$

where $\mathscr{V}$ is the collection of variables.

(sort)	$\vdash s_1 : s_2$	if $s_1 : s_2 \in \mathscr{A}$
(var)	$\dfrac{\Gamma \vdash T : s}{\Gamma, x{:}T \vdash x : T}$	if $x \notin \Gamma$
(weak)	$\dfrac{\Gamma \vdash T : s \qquad \Gamma \vdash M : U}{\Gamma, x{:}T \vdash M : U}$	if $x \notin \Gamma$
(Π)	$\dfrac{\Gamma \vdash T : s_1 \qquad \Gamma, x{:}T \vdash U : s_2}{\Gamma \vdash \Pi x{:}T.U : s_3}$	if $(s_1, s_2, s_3) \in \mathscr{R}$
(λ)	$\dfrac{\Gamma, x{:}T \vdash M : U \qquad \Gamma \vdash \Pi x{:}T.U : s}{\Gamma \vdash \lambda x{:}T.M : \Pi x{:}T.U}$	
(app)	$\dfrac{\Gamma \vdash M : \Pi x{:}T.U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U[N/x]}$	
(conv$_\beta$)	$\dfrac{\Gamma \vdash M : T \qquad \Gamma \vdash U : s}{\Gamma \vdash M : U}$	$T =_\beta U$

Fig. 1. The derivation rules of PTS

We can define a number of well known type systems as Pure Type Systems. We give the PTS definitions of the type systems that are mentioned in this article.

$\lambda 2$ (System F)

$\mathscr{S}$	$\star, \square$
$\mathscr{A}$	$\star : \square$
$\mathscr{R}$	$(\star,\star), (\square,\star)$

λU^-

$\mathscr{S}$	$\star, \square, \triangle$
$\mathscr{A}$	$\star : \square, \square : \triangle$
$\mathscr{R}$	$(\star,\star), (\square,\star), (\square,\square), (\triangle,\square)$

λU

$\mathscr{S}$	$\star, \square, \triangle$
$\mathscr{A}$	$\star : \square, \square : \triangle$
$\mathscr{R}$	$(\star,\star), (\square,\star), (\square,\square), (\triangle,\square), (\triangle,\star)$

$\lambda\star$ (`Type:Type`)

$\mathscr{S}$	$\star$
$\mathscr{A}$	$\star : \star$
$\mathscr{R}$	$(\star,\star)$

We remind the notation $\Gamma \vdash M : A : K$ that we use as an abbreviation for the conjunction of $\Gamma \vdash M : A$ and $\Gamma \vdash A : K$.

3.1 Looping combinators in PTS's

The notion of looping combinators for a PTS occurs in (Coquand & Herbelin, 1994). Similarly we can define the notion of a fixed point combinator for a PTS. These combinators are of a polymorphic type and therefore connected to the *sort* they can take types from.

Definition 3.2
Given a Pure Type System $T = (\mathscr{S}, \mathscr{A}, \mathscr{R})$ and a *sort* $s \in \mathscr{S}$. A *fixed point combinator of sort s* in T is a closed term $Y : \Pi A{:}s.(A \to A) \to A$ such that for all natural numbers n, $A : s$ and $f : A \to A$ holds

$$(Y\ A\ f) =_\beta f(Y\ A\ f)$$

A *looping combinator of sort s* in T is a closed term $L_0 : \Pi A{:}s.(A \to A) \to A$ such that there exists a sequence of terms $L \equiv L_0, L_1, L_2, \ldots, L_n, \ldots$ of type $\Pi A{:}s.(A \to A) \to A$ such that for all natural numbers n, $A : s$ and $f : A \to A$ holds

$$(L_n\ A\ f) =_\beta f(L_{n+1}\ A\ f)$$

3.2 The system λU

We now further study λU as a PTS, and present an erasure map from λU terms to untyped lambda terms. In λU, one can define all partial recursive functions, because there is a looping combinator and one can define data types for booleans and natural numbers in the standard polymorphic way, so the results of Section 2 apply. To be precise, the kinds for natural numbers and booleans in λU are as follows. (They are actually *polymorphic kinds*.)

$$\begin{array}{rcl} \mathsf{nat} & := & \Pi k{:}\Box.k \to (k \to k) \to k \\ \mathsf{bool} & := & \Pi k{:}\Box.k \to k \to k \end{array}$$

with the usual polymorphic definitions for the numerals, the booleans, test-for-zero and if-then-else.

We now first give a "layered definition" of pseudo-terms of λU, which is implicit in (Geuvers, 1993) and occurs explicitly in (Miquel, 2000).

Definition 3.3
We divide the set of variables $\mathscr{V}$ into three disjoint sets $\mathrm{var}^\triangle$, $\mathrm{var}^\Box$ and $\mathrm{var}^\star$ that we use in the (var) rule of Fig. 1. We use the following notation for the variables from these three sets.

$$\begin{array}{rcl} \mathrm{var}^\triangle & = & \{k_1, k_2, k_3, \ldots\} \\ \mathrm{var}^\Box & = & \{\alpha, \beta, \gamma, \ldots\} \\ \mathrm{var}^\star & = & \{x, y, z, \ldots\} \end{array}$$

Definition 3.4
We define the syntactical categories *Kinds*, *Constructors* and *Proof terms*. as follows (where $k \in \mathrm{var}^\triangle$, $\alpha \in \mathrm{var}^\Box$ and $x \in \mathrm{var}^\star$). Along with the definition we fix notation to range over

these categories.

Kinds	K	$::=$	$k \mid \star \mid K \to K \mid \Pi k\!:\!\Box.K$
notation			$K_1, K_2, K_3, \ldots$
Constructors	P	$::=$	$\alpha \mid \lambda\alpha\!:\!K.P \mid PP \mid P \to P$
			$\mid \lambda k\!:\!\Box.P \mid PK$
			$\mid \Pi\alpha\!:\!K.P$
notation			$P, Q, R, \ldots$
Proof terms	t	$::=$	$x \mid \lambda x\!:\!P.t \mid tt$
			$\mid \lambda\alpha\!:\!K.t \mid tP$
			$\mid \lambda k\!:\!\Box.p \mid pK$
notation			$t, p, q, \ldots$

Apart from $\Box$ and $\triangle$, each λU-term is in one of the syntactical categories of Definition 3.4. We have the following.

Proposition 3.1

1. If $\Gamma \vdash M : U : \Box$ then $U \in$ Kinds and $M \in$ Constructors
2. If $\Gamma \vdash M : U : \star$ then $U \in$ Constructors and $M \in$ Proof terms

Definition 3.5

We call a term U a *type* if $\Gamma \vdash U : \star$ for some Γ. Using Proposition 3.1 we see that the category of *types* is a subset of the constructors.

Using these definitions we define a meaningful erasure function on terms of λU that maps proof terms to untyped lambda calculus terms.

Definition 3.6

For t a proof term of λU, we define the *erasure* of t, denoted by $|t|$, as follows, by induction on the construction of proof terms.

$$
\begin{array}{lcll}
|x| & = & x & \\
|\lambda x\!:\!P.p| & = & \lambda x.|p| & \text{if } P \in \text{Constructors} \\
|pq| & = & |p||q| & \text{if } p, q \in \text{Proof terms} \\
|\lambda\alpha\!:\!K.p| & = & |p| & \text{if } K \in \text{Kinds} \\
|pP| & = & |p| & \text{if } P \in \text{Constructors} \\
|\lambda k\!:\!\Box.p| & = & |p| & \\
|pK| & = & |p| & \text{if } K \in \text{Kinds}
\end{array}
$$

All general results of PTSs apply to λU. The main additional result that we need, which is kind of ‘folklore’, even though no written proof of it exists to our knowledge, is that constructors and kinds of λU are (strongly) normalizing. We only need them to be normalizing, but the proof below gives a strong normalization result.

Definition 3.7

Assume a context $\Sigma := A : \star, c : (\star{\to}A){\to}A, d : \Pi\alpha{:}\star.(\alpha{\to}A){\to}A, e : \star{\to}\star{\to}\star$. We define the map $[-]$ from Kinds and Constructors of λU to $\lambda 2$ as follows.

Kinds			Constructors		
$[k]$	$=$	k	$[\alpha]$	$=$	α
$[\star]$	$=$	A	$[P\,K]$	$=$	$[P]\,[K]$
$[K_1{\to}K_2]$	$=$	$[K_1]{\to}[K_2]$	$[P\,Q]$	$=$	$[P]\,[Q]$
$[\Pi k{:}\Box.K]$	$=$	$\Pi k{:}\star.[K]$	$[\lambda k{:}\Box.P]$	$=$	$\lambda k{:}\star.[P]$
			$[\lambda\alpha{:}K.P]$	$=$	$\lambda\alpha{:}[K].[P]$
			$[\Pi k{:}\Box.P]$	$=$	$c\,(\lambda k{:}\star.[P])$
			$[\Pi\alpha{:}K.P]$	$=$	$d\,[K](\lambda\alpha{:}[K].[P])$
			$[P{\to}Q]$	$=$	$e\,[P]\,[Q]$

The mapping $[-]$ is extended to contexts by extending it to variable declarations as follows: $[k : \Box] := k : \star$, $[\alpha : K] := \alpha : [K]$ (if $K : \Box$) and $[x : \sigma] := x : [\sigma]$ (if $\sigma : \star$).

We can prove by induction on the derivation that $[-]$ is a sound embedding of constructors and kinds of λU to proof-terms and types of system F. Moreover, it is immediate that the mapping $[-]$ preserves reductions. So we have the following Lemma, and SN as an immediate Corollary because system F is SN.

Lemma 3.1
Suppose $\Gamma \vdash P : K : \Box$ in λU. Then $\Sigma, [\Gamma] \vdash [P] : [K] : \star$ in system F. If also $P \to_\beta Q$, then $[P] \to_\beta^+ [Q]$.

Corollary 3.1
All kinds and constructors of λU are strongly normalizing.

3.3 Untypability of Ω*,* Y*,* Θ

Definition 3.8
An untyped lambda term M is *typable in* λU iff there exist Γ, t, P such that $\Gamma \vdash t : P : \star$ and $|t| = M$.

We prove that the terms Ω, Y and Θ are not typable in λU. The result we prove is more general and the non-typability of Ω, Y ad Θ is an immediate consequence of it. We first give two definitions that help phrase the general Theorem.

It should be pointed out that both the general Theorem and its instantiation to the non-typability of Ω, Y and Θ immediately go through for λU^-, but not for $\lambda\star$. The crucial difference is that in λU and λU^-, the types are normalizing, and hence we can make a crucial case distinction. (See Remark 2 below.)

Definition 3.9
Let t be a proof-term t and x a variable.

- We say that t *contains a self-application of* x if $|t|$ contains the sub-term $x\,x$.
- We say that t *contains* Ω if $|t|$ contains a sub-term of the shape $(\lambda x.N)(\lambda y.P)$ where N contains a sub-term $x\,x$ and P contains a sub-term $y\,y$

Theorem 3.2
If the proof-term t contains Ω, t is not typable in λU.

This Theorem is the main result of this section. The remainder of this section is devoted to its proof.

Remark 2
Because types are SN in λU, we can safely assume types to be in normal form at all times.

We use the abbreviation $\Pi v : V.\sigma$ to denote an abstraction over a kind-variable *or* a constructor variable. So, this covers bot the cases $\Pi k : \Box.\sigma$ and $\Pi\alpha : K.\sigma$.

A type σ in normal form is of one of the following two forms

- $\Pi\vec{v} : \vec{V}.\tau \to \rho$ (for τ and ρ types in normal form),
- $\Pi\vec{v} : \vec{V}.\alpha\vec{T}$ (for α a constructor variable).

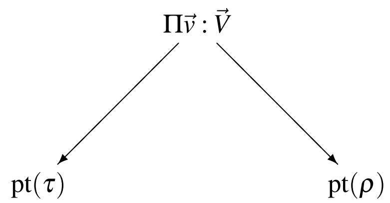
In both these cases, $\vec{v}$ and $\vec{V}$ or $\vec{T}$ may be empty.

We extend the notion of *parse tree* of a type σ, known from (Wells, 1999) for system F and extended to $F\omega$ in (Urzyczyn, 1997).

Definition 3.10
Given $\Gamma \vdash \sigma : \star$ in λU we define the *parse tree* of σ (written $\mathrm{pt}(\sigma)$) as follows. We use Remark 2

- If $\sigma \equiv \Pi\vec{v} : \vec{V}.\tau \to \rho$ then

$\mathrm{pt}(\Pi\vec{v} : \vec{V}.\tau \to \rho) =$



- If $\sigma \equiv \Pi\vec{v} : \vec{V}.\alpha\vec{T}$ then

$$\mathrm{pt}(\Pi\vec{v} : \vec{V}.\alpha\vec{T}) = \Pi\vec{v} : \vec{V}.\alpha\vec{T}$$

Definition 3.11
We denote a *path* by a finite sequence $X \in \{L, R\}^*$. Let σ be a type.

1. We say that X *is a path in* σ if we stay inside $\mathrm{pt}(\sigma)$ when following X through the parse tree of σ. (Of course, at every node in $\mathrm{pt}(\sigma)$ we move left when reading an L and right when reading an R.)
2. A *left-going path in* σ is a path in σ that consists only of Ls. (So the path has no branches to the right in $\mathrm{pt}(\sigma)$.)
3. The *left-terminal path* in σ is the unique left-going path from the root of $\mathrm{pt}(\sigma)$ to a leaf. We will write $\mathrm{ltp}(\sigma)$ for the left-terminal path of σ.

Note: $\mathrm{length}(\mathrm{ltp}(\sigma \to \tau)) = \mathrm{length}(\mathrm{ltp}(\sigma)) + 1$ and $\mathrm{length}(\mathrm{ltp}(\Pi v : V.\sigma)) = \mathrm{length}(\mathrm{ltp}(\sigma))$.

Definition 3.12
Given a variable α, a type σ and a path X, we define the notion α *owns the path* X *in* σ by induction on σ (using Remark2) as follows.

- If $\sigma = \Pi\vec{v}:\vec{V}.\tau \to \rho$, then α *owns* LX' *in* σ if α owns X' in τ and α *owns* RX' *in* σ if α owns X' in ρ,
- If $\sigma = \Pi\vec{v}:\vec{V}.\alpha\vec{T}$, then α *owns* X *in* σ if X is the empty sequence.

In the last case, α may be one of the variables v.

The intuition is that α owns X in σ precisely when we arrive at a leaf with annotation $\Pi\vec{v}:\vec{V}.\alpha T_1 T_2 \ldots T_n$ (for some $T_1,\ldots,T_n$) in case we follow the path X through pt(σ), starting from the root.

We also define the *containment* relation ($\preceq$) between types of λU, as an extension of that notion for Fω, as it occurs in (Urzyczyn, 1997).

Definition 3.13
Given two well-formed types σ and τ (so $\Gamma \vdash \sigma : \star$ and $\Gamma' \vdash \tau : \star$ in λU form some Γ, Γ'), we say that σ *is contained in* τ, notation $\sigma \preceq \tau$, if

- $\sigma = \Pi\vec{v}{:}\vec{V}.\rho$, for some (possibly empty) vector $\vec{v}$ and type ρ such that there are no quantifiers at the root of ρ,
- $\tau = \Pi\vec{w}{:}\vec{W}.\rho[\vec{T}/\vec{v}]$, where the variables in $\vec{w}$ do not occur free in σ.

As in (Urzyczyn, 1997), it is easy to see that this containment relation is reflexive and transitive, so it is a quasi-order.

Lemma 3.2
If σ and τ are types with $\sigma \preceq \tau$, then length(ltp(σ)) $\leq$ length(ltp(τ)).

Proof
This follows directly from the fact that the only parts of pt(σ) that are affected by a substitution are the leaves, which can only expand. □

By the same reasoning as in the proof of Lemma 3.2, when $\sigma \preceq \tau$, the entire tree structure of σ remains present in τ. So we have the following Corollary.

Corollary 3.2
For every path X in σ, there is a path X' in τ such that X is a prefix of X'.

Lemma 3.3
If $\sigma \preceq \tau$ and ltp(σ) is not owned by a variable quantified at the root of σ, then ltp(σ) = ltp(τ).

Proof
Let σ and τ be types such that $\sigma \preceq \tau$ and suppose ltp(σ) is not owned by a variable quantified at the root of σ. Then by Definition 3.13 there are $\vec{\alpha}, \vec{\beta}, \vec{\rho}, \sigma'$ such that $\sigma = \Pi\vec{\alpha}.\sigma'$ and $\tau = \Pi\vec{\beta}.\sigma'[\vec{\rho}/\vec{\alpha}]$. The variable at the leaf at the end of the left-terminal path is not replaced by the substitution $[\vec{\rho}/\vec{\alpha}]$, so the left-terminal path of σ is the same as that of τ.
□

Lemma 3.4
If $\Gamma \vdash t : \sigma : \star$ and t contains a self application of x, with $x : \tau$, then ltp(τ) is owned by a variable that is quantified at the root of τ.

Proof
Suppose $\Gamma \vdash t : \sigma$ and $|t|$ contains the sub-term xx with $x : \tau$. (The type τ may be given as a declaration in Γ or $\lambda x : \tau.q$ may be a sub-term of t.) The general form of the self-application of x in t is

$$x\vec{T}(\lambda\vec{v} : \vec{V}.x\vec{R}).$$

Suppose that $x\vec{T} : \rho_1$ and $\lambda\vec{v} : \vec{V}.x\vec{R} : \rho_2$. We know that $\rho_1 = \rho_2 \to \rho_3$ for some ρ_3, so $\text{length}(\text{ltp}(\rho_1)) = \text{length}(\text{ltp}(\rho_2)) + 1$.

Also $\tau \preceq \rho_1$ and $\tau \preceq \rho_2$. If $\text{ltp}(\tau)$ is not owned by a variable at the root of τ, then $\text{ltp}(\rho_1) = \text{ltp}(\tau) = \text{ltp}(\rho_2)$ as a consequence of Lemma 3.3. Contradiction, so $\text{ltp}(\tau)$ is owned by a variable quantified at the root of τ. □

Proof of Theorem 3.2 Suppose that a proof-term t contains Ω. Then $|t|$ contains a sub-term $(\lambda x.Q)(\lambda y.P)$ such that Q contains a sub-term xx and P contains a sub-term yy. So there are λU proof-terms q and p such that qp is a sub-term of t, $|q| = \lambda x.Q$ and $|p| = \lambda y.P$. There are types σ and τ such that $\Gamma \vdash q : \sigma \to \tau$ and $\Gamma \vdash p : \sigma$. Because $|q| = \lambda x.Q$, we know that $x : \sigma$. Because xx is a sub-term of Q, we know that $\text{ltp}(\sigma)$ is owned by a variable quantified at the root of σ (Lemma 3.4). Because $|p| = \lambda y.P$, we know that $\sigma = \rho_1 \to \rho_2$ for some ρ_1, ρ_2 with $y : \rho_1$. Because yy is a sub-term of P, we know that $\text{ltp}(\rho_1)$ is owned by a variable quantified at the root of ρ_1. However, this means that $\text{ltp}(\sigma)$ is *not* owned by a variable quantified at the root, which is a contradiction. Thus, t is not typable. □

The theorem implies that many untyped λ-terms M are not *typable in* λU, that is, there is no well-typed term t in λU such that $|t| \equiv M$.

Corollary 3.3
The following well-known untyped λ-terms are not typable in λU:

$$\begin{aligned} \Omega &= (\lambda x.xx)(\lambda x.xx), \\ Y &= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)), \\ \Theta &= (\lambda xy.y(xxy)(\lambda xy.y(xxy)). \end{aligned}$$

4 Conclusion

We have given a general proof that the well-known fixed-point combinators Y, Ω and Θ are not typable in λU. For $\lambda\star$, the situation is very much open. The techniques that we have applied here immediately go through for λU^-, but don't work for $\lambda\star$, because types are not SN in $\lambda\star$. So whether Y, Ω and Θ are typable in $\lambda\star$ remains open.

Another interesting question that remains is whether a fixed-point combinator exists at all in λU. One can study the looping combinator L_0 in λU (or λU^-) that can be created using the inconsistency proof of Hurkens (Hurkens, 1995). If we erase all type information, we obtain the following term.

$$|L_i| = L = \lambda f.(\lambda x.x(\lambda pq.f(qpq))x)(\lambda y.yy)$$

In the untyped λ-calculus, this is a fixed-point combinator and an interesting one, because it contains no *double self-application*, as Ω, Y and Θ do. Our Theorem 3.2 proves that double self-application is non-typable in λU. It should be noted that if we erase from L_0

just the "domains", the obtained term is also a fixed-point combinator. This has been shown in (Barthe & Coquand, 2006). This is called the *domain free* version of the term L_0, and it is obtained by changing the erasure of Definition 3.6 to: $|\lambda v : V.M| = \lambda v.|M|$ and all the other cases by structural recursion. (One just strips away all type information in the λ-abstractions.) So, the fact that L_0 is not a fixed-point combinator is just due to the fact that the types in the lambda-abstractions expand.

We conjecture that the term L cannot be typed as a fixed point combinator in λU, and more generally we conjecture that no fixed point combinator exists in λU. The work here shows that seeing types as trees isolates a lot of useful structure from them. The branches of the tree often remain unchanged when the type is manipulated. As the tree structure is a graphic representation of the $\star$ level of types, the definition of trees does not change much from System F to Fω to λU.

Acknowledgments

We would like to thank Hugo Herbelin and Joe Wells for discussions on the topic of non typability of Ω in System F.

References

Barendregt, H. (1992). Lambda calculi with types. *Pages 117–309 of:* Abramsky, S., Gabbay, D. M., & Maibaum, T. S. E. (eds), *Handbook of logic in computer science*. Oxford University Press.

Barendregt, H., & Geuvers, H. (2001). Proof-assistants using dependent type systems. *Pages 1149–1238 of:* Robinson, J.A. & Voronkov, A. (eds), *Handbook of automated reasoning*. Elsevier.

Barendregt, H.P. (1981). *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logics and the Foundations of Mathematics, vol. 103. North Holland, Amsterdam, The Netherlands.

Barthe, G., & Coquand, Th. (2006). Remarks on the equational theory of non-normalizing pure type systems. *Journal of functional programming*, **16**(2), 137–155.

Coquand, Th. (1994). A new paradox in type theory. *Pages 7–14 of: Logic, methodology and philosophy of science ix: Proc. ninth int. congress of logic, methodology, and philosophy of science*. Elsevier.

Coquand, Th., & Herbelin, H. (1994). A-translation and looping combinators in pure type systems. *Journal of functional programming*, **4**, 77–88.

Geuvers, H. (1993). *Logics and type systems*. Ph.D. thesis, Radboud University, Nijmegen.

Geuvers, H. (2007). (In)consistency of extensions of higher order logic and type theory. *Pages 140–159 of:* Altenkirch, Th. & McBride, C. (eds), *Types for proofs and programs int. workshop, nottingham, uk, april 18-21, 2006, revised selected papers*. LNCS, vol. 4502. Springer.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.

Howe, D. J. (1987). The computational behaviour of Girard's paradox. *Pages 205–214 of: Proceedings of the 2nd symposium on logic in computer science*. IEEE.

Hurkens, A.J.C. (1995). A simplification of Girard's paradox. *Pages 266–278 of: TLCA '95: Proceedings of the 2nd int. conf. on typed lambda calculi and applications*. London, UK: Springer.

Kleene, S.C. (1936). Lambda-definability and recursiveness. *Duke mathematical journal*, **2**, 340–353.

Miquel, A. 2000 (11). *Russell's Paradox in System U^-*. Notes of a talk at TYPES 2000, Durham UK.

Reinhold, M.B. (1986). *Typechecking is undecidable when 'type' is a type*.

Reynolds, J.C. (1984). Polymorphism is not set-theoretic. *Pages 145–156 of:* Kahn, G. MacQueen, D.B. & Plotkin, G.D. (eds), *Semantics of data types, international symposium, sophia-antipolis, france, june 27-29, 1984, proceedings*. Lecture Notes in Computer Science, vol. 173. Springer.

Urzyczyn, P. (1997). Type reconstruction in Fω. *Mathematical structures in comp. sci.*, **7**(4), 329–358.

Wells, J.B. (1999). Typability and type checking in system f are equivalent and undecidable. *Annals of pure and applied logic*, **98**, 111–156.