On Fixed point and Looping Combinators in Type Theory

Herman Geuvers and Joep Verkoelen

Institute for Computing and Information Sciences
Radboud University Nijmegen
Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands

Abstract. The type theories λU and λU^- are known to be logically inconsistent. For λU, this is known as Girard's paradox [Gir72]; for λU^- the inconsistency was proved by Coquand [Coq94]. 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 was unclear whether a fixed point combinator exists in these systems. Later, Hurkens [Hur95] has given a simpler version of the paradox in λU^-, giving rise to an actual proof term that can be analyzed.

In the present paper we analyze the proof of Hurkens and we study the looping combinator that arises from it: it is a real looping combinator (not a fixed point combinator) but in the Curry version of λU^- it is a fixed-point combinator. We also analyze the possibility of typing a fixed point combinator in λU^- and we prove that the Church and Turing fixed point combinators 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. However, the system is computationally still interesting, because not all terms are β-convertible.

The first one to study the computational power of these inconsistent systems was [How87], going back to earlier (unpublished) work of [Rei86]. 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 $\lambda\star$. With then use of a looping combinator, it can be shown that the equational theory is undecidable and that the theory is Turing complete. The proof of this last fact, we have not been able to find in the published literature, so we outline it briefly in this paper.

When Girard 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 proved that λU^- is inconsistent as well, the proof of which was later considerably shortened by Hurkens. In the present paper we analyze the paradox in λU^- (but we believe that our results will apply to λU without a change). 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 Church fixed-point combinators (Θ and Y) cannot be typed in λU^-.

Before giving the negative result, we exhibit the proof of inconsistency of [Hur95] and we extract the looping combinator from the proof (and show that it is a looping combinator indeed). This analysis immediately shows that in the Curry version of λU^-, this looping combinator is 'just' a fixed point combinator. So all the "extra structure" is in the types.

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 [BB98].

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 mainly 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 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 usually refer to L_0 as 'the looping combinator', not mentioning the whole family. Then, 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. Finally, every fixed point combinator is a looping combinator.

In order to represent the 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 can also be achieved by using the natural numbers and Z for true and $S\,Z$ for false.) So, we assume a type nat with $Z : \text{nat}$, $S : \text{nat} \to \text{nat}$, $P^- : \text{nat} \to \text{nat}$ such that $P^-(S^{n+1}(Z)) =_\beta S^n(Z))$ and a type bool with $\mathsf{tt} : \text{bool}$ and $\mathsf{ff} : \text{bool}$ and $Z? : \text{nat} \to \text{bool}$ and, for $b : \text{bool}$, $e_1, e_2 : \text{nat}$, $\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \text{nat}$ such that $Z?Z =_\beta \mathsf{tt}$, $Z?(S\,x) =_\beta \mathsf{ff}$, $\mathsf{if}\,\mathsf{tt}\,\mathsf{then}\, e_1\, \mathsf{else}\, e_2 =_\beta e_1$ and $\mathsf{if}\,\mathsf{ff}\,\mathsf{then}\, e_1\, \mathsf{else}\, e_2 =_\beta e_2$.

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 solve questions of the following kind:

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

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

A looping combinator does the same 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'. So let us make the proof that all recursive functions are λ-definable in a type theory with a looping combinator precise here. The original proof can be found in [S.C36]. Our proof follows the proof given in [BB98]. It basically appears in the unpublished manuscript [Rei86].

Theorem 1. *In a typed λ-calculus with a data type for natural numbers and for booleans and a looping combinator L, the set of recursive functions is λ-definable.*

We need to prove that the basic functions are λ-definable and that the class of λ-definable functions is closed under composition, primitive recursion and minimization. We will only show that the λ-definable functions are closed under primitive recursion and minimization, because the other cases are immediate. Consider the looping combinator $L \equiv L_0, L_1, \ldots L_n, \ldots$

Notation 1 *We use the notation $\overline{}$ to denote the embedding of the natural number to the λ-term that λ-defines it. (So, $\overline{n}$ may be the Church numeral c_n, but we are not committed to a specific representation.)*

Lemma 1. *Given a looping combinator L the λ-definable functions are closed under primitive recursion.*

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

$$\begin{aligned} \varphi(\boldsymbol{x}, 0) &= \chi(\boldsymbol{x}) \\ \varphi(\boldsymbol{x}, n+1) &= \psi(\boldsymbol{x}, n, \varphi(\boldsymbol{x}, n)) \end{aligned}$$

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

$$\Phi := \lambda f\boldsymbol{x}n.\textsf{if } (Z?\ n) \textsf{ then } (G\boldsymbol{x}) \textsf{ else } (H\boldsymbol{x}(P^{-}n)(f\boldsymbol{x}(P^{-}n)))$$

We claim that for all natural numbers i

$$L_i\Phi \text{ lambda-defines } \varphi$$

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 $\forall n L_i\Phi\overline{n} =_\beta \overline{\phi(\boldsymbol{x}, n)}$ separately for every i. Instead we prove $\forall n\forall i(L_i\Phi\overline{n} =_\beta \overline{\phi(\boldsymbol{x}, n)})$ by induction on n.

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

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

$$\begin{aligned}
L_i\Phi\boldsymbol{x}\overline{n+1} &=_\beta \Phi(L_{i+1}\Phi)\boldsymbol{x}\overline{n+1}\\
&\twoheadrightarrow_\beta \textsf{if } (Z\ \overline{n+1}) \textsf{ then } (G\boldsymbol{x}) \textsf{ else } (H\boldsymbol{x}(P^{-}\overline{n+1})((L_{i+1}\Phi)\boldsymbol{x}(P^{-}\overline{n+1})))\\
&=_\beta H\boldsymbol{x}\overline{n}((L_{i+1}\Phi)\boldsymbol{x}\overline{n})\\
&=_\beta \psi(\boldsymbol{x}, n, \phi(\boldsymbol{x}, n))
\end{aligned}$$

The last equation uses the fact that H lambda-defines ψ and that $(L_{i+1}\Phi)\boldsymbol{x}\overline{n} =_\beta \overline{\varphi(\boldsymbol{x}, n)}$ (by IH). Thus for all i: $L_i\Phi$ λ-defines φ.

Lemma 2. *Given a looping combinator L the λ-definable functions are closed under minimization.*

Proof. Let φ be defined by

$$\varphi(\boldsymbol{x}) = \mu z[\chi(\boldsymbol{x}, z) = 0]$$

Where χ is total and lambda-defined by G. We now need to prove that there is a lambda term F without using a fixed point combinator such that

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

Define

$$\begin{aligned}
\Phi &\equiv (\lambda h\boldsymbol{x}z.\textsf{if } (Z?\ (G\boldsymbol{x}z)) \textsf{ then } z \textsf{ else } (h\boldsymbol{x}(Sz)))\\
H_i &\equiv \lambda\boldsymbol{y}.\lambda z.L_i\Phi\boldsymbol{y}z
\end{aligned}$$

Now we have

$$\begin{aligned}
H_i\boldsymbol{y}z &=_\beta L_i\Phi\boldsymbol{y}z\\
&=_\beta \Phi(L_{i+1}\Phi)\boldsymbol{y}z\\
&=_\beta \textsf{if } (Z?\ (G\boldsymbol{y}z)) \textsf{ then } z \textsf{ else } (L_{i+1}\Phi\boldsymbol{y}(Sz))
\end{aligned}$$

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

So, we can take any of the following terms F_i to λ-define ϕ: $F_i := H_i\boldsymbol{y}\overline{0}$. $\square$

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 [BAG$^+$92] and [Geu93].

Definition 2. *A* **Pure Type System** $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ *is given by a set* $\mathcal{S}$ *(of* sorts*), a set* $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ *(of* axioms*), and a set* $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ *(of* rules*), and is the typed lambda calculus with the reduction rules presented below.*

We assume $s \in \mathcal{S}$*. The elements of* $\mathcal{A}$ *are written as* $s_1 : s_2$ *with* $s_1, s_2 \in \mathcal{S}$*. The elements of* $\mathcal{R}$ *are written as* (s_1, s_2, s_3) *with* $s_1, s_2, s_3 \in \mathcal{S}$*. If* $s_2 = s_3$*, we write* (s_1, s_2) *instead.*

(sort)	$\vdash s_1 : s_2$	*if* $s_1 : s_2 \in \mathcal{A})$
(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* $(s_1, s_2, s_3) \in \mathcal{R}$
(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$_\beta$)	$\dfrac{\Gamma \vdash M : T \qquad \Gamma \vdash U : s}{\Gamma \vdash M : U}$	$T =_\beta U$

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

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

Where $\mathcal{V}$ *is the collection of variables.*

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.

$$\lambda U^- \quad \boxed{\begin{array}{l} \mathcal{S}\ \star, \Box, \triangle \\ \mathcal{A}\ \star : \Box, \Box : \triangle \\ \mathcal{R}\ (\star,\star), (\Box,\star), (\Box,\Box), (\triangle,\Box) \end{array}}$$

$$\lambda U \boxed{\begin{array}{l} \mathcal{S}\ \star, \Box, \triangle \\ \mathcal{A}\ \star : \Box, \Box : \triangle \\ \mathcal{R}\ (\star,\star), (\Box,\star), (\Box,\Box), (\triangle,\Box), (\triangle,\star) \end{array}}$$

$$\begin{array}{c} \lambda\star \\ (\texttt{Type:Type}) \end{array} \qquad \boxed{\begin{array}{l} \mathcal{S}\ \star \\ \mathcal{A}\ \star : \star \\ \mathcal{R}\ (\star,\star) \end{array}}$$

Throughout this article, we will be using these definitions.

3.1 Looping combinators in PTS's

We can find a definition of looping combinators for a PTS in [CH94]. We can define a fixed point combinator in the same way. Both will be defined below. These combinators are of a polymorphic type and therefore connected to the *sort* they can take types from.

Definition 3. *Given a Pure Type System* $T = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ *and a* sort $s \in \mathcal{S}$. *A* Fixed Point Combinator of sort s *in* T *is a term* $Y : \mathit{\Pi} A : s.(A \to A) \to A$ *such that for all natural numbers* n, $A : s$ *and* $f : A \to A$ *holds*

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

Definition 4. *Given a Pure Type System* $T = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ *and a* sort $s \in \mathcal{S}$. *A* **Looping Combinator of sort** s *in* T *is a term* $L_0 : \mathit{\Pi} A : s.(A \to A) \to A$ *such that there exists a sequence of terms* $L \equiv L_0, L_1, L_2, \ldots, L_n, \ldots$ *of type* $\mathit{\Pi} A : s(A \to A) \to A$ *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 $\boldsymbol{\lambda U^-}$

We now further study λU^- as a PTS, and present an erasure map from λU^- terms to untyped lambda terms.

[Miq00] gives a nice layered definition of (pseudo-)terms of λU^-, which we will copy and expand here:

Definition 5. *We define three sets of variables* $var^\triangle$, $var^\Box$ *and* $var^\star$ *as follows*

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

Definition 6. *We define the syntactical categories* **Kinds**, **Constructors** *and* **Proof terms** *as follows (where $k \in var^{\triangle}$, $\alpha \in var^{\Box}$ and $x \in var^{\star}$)*

$$\begin{array}{lll}
\textit{Kinds} & K & ::= k \mid \star \mid K \to K \mid \Pi k{:}\Box.K \\
\\
\textit{Constructors} & P & ::= \alpha \mid \lambda\alpha{:}K.P \mid PP \mid P \to P \\
 & & \mid \lambda k{:}\Box.P \mid PK \\
 & & \mid \Pi\alpha{:}K.P \\
\\
\textit{Proof terms} & t & ::= x \mid \lambda x{:}P.t \mid tt \\
 & & \mid \lambda\alpha{:}K.t \mid tP
\end{array}$$

Remark 2. Apart from $\Box$ and $\triangle$, all λU^{-} terms are part of one of the syntactical categories as defined in definition 6.

Notation 2 *We will use the following notation for terms and variables*

	variables	*terms*
Kinds	$k_1, k_2, k_3, \ldots$	$K_1, K_2, K_3, \ldots$
Constructors	$\alpha, \beta, \gamma, \ldots$	$P, Q, R, \ldots$
Proof terms	$x, y, z, \ldots$	$t, p, q, \ldots$

To see that this definition indeed gives us merely pseudo-terms, we only need to look at the application rule for two proof terms. The rule $t ::= tt$ does not demand that the types of the two proof terms being applied match in any way.

Proposition 1. *We have the following.*

1. *If there is a derivation of the form $\Gamma \vdash M : U : \Box$ then $U \in$ Kinds and $M \in$ Constructors*
2. *If there is a derivation of the form $\Gamma \vdash M : U : \star$ then $U \in$ Constructors and $M \in$ Proof terms*

Definition 7. *With proposition 1 we can define the category* **Types** *as a subset of the Constructors. A term U is a Type iff we can make a derivation of the form $\Gamma \vdash U : \star$.*

Using these definitions it is easy to define a meaningful erasure function on terms of λU^{-} that maps proof terms onto untyped lambda calculus terms.

Definition 8. *Given a pseudo-term of λU^{-}, t, we define the* **erasure** *of t, $|t|$, with induction on the construction of proof terms as given above.*

$$\begin{array}{lll}
|x| & = x & \\
|\lambda x{:}P.p| & = \lambda x.|p| & \textit{if } P \in \textit{Constructors} \\
|pq| & = |p||q| & \textit{if } p, q \in \textit{Proof terms} \\
|\lambda\alpha{:}K_1.p| & = |p| & \textit{if } K_1 \in \textit{Kinds} \\
|pP| & = |p| & \textit{if } P \in \textit{Constructors} \\
|\lambda k{:}\Box.p| & = |p| & \\
|pK_1| & = |p| & \textit{if } K_1 \in \textit{Kinds}
\end{array}$$

4 Looping combinators in λU^-

In this section we will take a look at looping combinators in λU^-. Coquand and Herbelin [CH94] have shown that in any inconsistent logical Pure Type System, a looping combinator can be derived from any term of type $\bot$. In addition, [Geu07] has given a concrete looping combinator based on the proof of the inconsistency of λU^- as presented in [Hur95]. We take a look at the proof that Hurkens presented, follow Geuvers' formalization of that proof in Coq and show that this yields a looping combinator. Also, given the erasure of this looping combinator, we obtain a fixed point combinator in the untyped λ-calculus.

4.1 Inconsistency of λU^-

The main example that we will study in this section is the Lego formalization Geuvers and Pollack have made of Hurkens' proof of the inconsistency of λU^-. We will extract lambda terms from the Lego code and analyze them. Lego uses $\lambda\star$ as its logical system, where you have Type : Type, but we can read this as λU^- code with little effort. Following is a copy of the code as it appeared in [Geu07] with the suggestion for the looping combinator applied.

```
[V = {A|Type}((A->Type)->(A->Type))->A->Type];
[U = V->Type];
[sb [A|Type][r:(A->Type)->(A->Type)][a:A] = [z:V]r (z r) a : U];
[le [i:U->Type][x:U] =
       x ([A|Type][r:(A->Type)->(A->Type)][a:A]i (sb r a)) :
Type];
[induct [i:U->Type] = {x:U}(le i x)->i x : Type];
[WF = [z:V]induct (z le) : U];
[B:Type];
[F:B->B];
[I [x:U] = ({i:U->Type}(le i x)->i (sb le x))-> B :Type];

Goal i:U->Type(induct i)-> i WF;
intros i y;
Refine y WF ([x:U]y (sb le x));
Save omega;

Goal induct I;
intros x p q;
Refine F (q I p ([i:U->Type]q ([y:U]i (sb le y))));
Save lemma;

Goal ({i:U->Type}(induct i)->i WF)->B;
intros x;
Refine x I lemma ([i:U->Type]x ([y:U]i (sb le y)));
Save lemma2;
```

```
Goal B; Refine lemma2 omega;
Save paradox;
```

In terms of type theory, `{x:U}` denotes a $\mathit{\Pi}$-abstraction, `[z:V]` denotes a λ-abstraction, and `{A|Type}` `[A|Type]` denote implicit arguments. For Coq users: `Refine` is basically the `apply` tactic. It is important to note that if you read this as λU^-, then `Type` denotes $\star$ in all but three cases. These three cases are the first occurences of the word `Type` in the definitions of `V`, `sb` and `le`, in which case it should be read as $\Box$[1]

We will use U to denote the term that lambda-defines `U` in the Lego code. The term omega thus becomes:

$$\begin{aligned} \text{omega} &\equiv \lambda i : U \to \star.\lambda y : (\text{induct } i).y \text{ WF } (\lambda x : U.y\ (sb\ le\ x)) \\ &\quad :\ \mathit{\Pi} i : U \to \star.(\text{induct } i) \to i \text{ WF} \end{aligned}$$

If we take $\beta : \star$ for `B:Type`, we get the following terms:

$$\begin{aligned} \text{lemma} &\equiv \lambda x : U.\lambda p : (le\ I\ x).\lambda q : (\mathit{\Pi} i : U \to \star.(le\ i\ x) \to i(sb\ le\ x)). \\ &\quad f\ (q\ I\ p\ (\lambda i : U \to \star.q\ (\lambda y : U.i\ (sb\ le\ y)))) \\ &\quad :\ \text{induct } I \\ \text{lemma2} &\equiv \lambda x : (\mathit{\Pi} i : U \to \star.(\text{induct } i) \to i \text{ WF}). \\ &\quad x\ I \text{ lemma } (\lambda i : U \to \star.x\ (\lambda y : U.i\ (sb\ le\ y))) \\ &\quad :\ (\mathit{\Pi} i : U \to \star.(\text{induct } i) \to i \text{ WF}) \to \beta \\ \text{paradox} &\equiv \text{lemma2 omega}\ :\ \beta \end{aligned}$$

We see here that paradox gives us a proof term for any proposition β[2], thus proving the inconsistency of λU^-. In addition, we can now define a looping combinator for the function $f : \beta \to \beta$. The proof of this follows in the section 4.2.

The terms we generated have a notation close to the Lego code, but which does not make clear distinction between proof terms and constructors. Therefore, we will first rewrite the variable names to adhere to notation 2. In addition, we will write $G \equiv (sb\ le)$.

$$\begin{aligned} \text{omega} &\equiv \lambda \alpha : U \to \star.\lambda y : (\text{induct } \alpha).y \text{ WF } (\lambda \gamma : U.y\ (G\ \gamma)) \\ &\quad :\ \mathit{\Pi} \alpha : U \to \star.(\text{induct } \alpha) \to \alpha \text{ WF} \\ \text{lemma} &\equiv \lambda \gamma : U.\lambda p : (le\ I\ \gamma).\lambda q : (\mathit{\Pi} \alpha : U \to \star.(le\ \alpha\ \gamma) \to \gamma(G\ \alpha)). \\ &\quad f\ (q\ I\ p\ (\lambda \alpha : U \to \star.q\ (\lambda \delta : U.\alpha\ (G\ \delta)))) \\ &\quad :\ \text{induct } I \\ \text{lemma2} &\equiv \lambda x : (\mathit{\Pi} \alpha : U \to \star.(\text{induct } \alpha) \to \alpha \text{ WF}). \\ &\quad x\ I \text{ lemma } (\lambda \alpha : U \to \star.x\ (\lambda \gamma : U.\alpha\ (G\ \gamma))) \\ &\quad :\ (\mathit{\Pi} \alpha : U \to \star.(\text{induct } \alpha) \to \alpha \text{ WF}) \to \beta \\ \text{paradox} &\equiv \text{lemma2 omega}\ :\ \beta \end{aligned}$$

---

[1] These are exactly the three instances where `A` is an implicit argument.

[2] The term contains a free variable $f : \beta \to \beta$. One can leave this free variable out in the proof term of β as its only purpose is to make a looping combinator out of the proof term.

4.2 The looping combinator and its erasure

We now take a look at the looping combinator generated from the inconsistency proof of λU^-. First, we define *domain free* versions of terms in order to make them easier to read [Br95]. This removes the type information in the λ-abstraction. We introduce the abbreviation $f \circ g$ to denote $\lambda z.f\,(g\,z)$.

$$\begin{aligned}
\text{omega} &\equiv \lambda \alpha y.y \text{ WF } (y \circ G) \\
\text{lemma}^f &\equiv \lambda \gamma p q.f\ (q\ I\ p\ (\lambda \alpha.q\ (\alpha \circ G))) \\
\text{lemma2}^f &\equiv \lambda x.x\ I \text{ lemma}^f\ (\lambda \alpha.x\ (\lambda \gamma.\alpha \circ G)) \\
\text{paradox}^f &\equiv \text{lemma2}^f \text{ omega}
\end{aligned}$$

$$\begin{aligned}
\text{WF}_1 &\equiv \text{WF} \\
\text{WF}_{n+1} &\equiv (G \text{ WF}_n)
\end{aligned}$$

$$\begin{aligned}
P_1^f &\equiv \text{lemma}^f \circ G \\
P_{n+1}^f &\equiv P_n^f \circ G \equiv \text{lemma}^f \circ \underbrace{G \circ \ldots \circ G}_{n+1} \\
Q_1 &\equiv \lambda \alpha.\text{omega } (\alpha \circ G) \\
Q_{n+1} &\equiv \lambda \alpha.Q_n\ (\alpha \circ G) \equiv \lambda \alpha.\text{omega}(\underbrace{\alpha \circ \ldots \circ \alpha}_{n+1} \circ G)
\end{aligned}$$

The proof of the paradox centers around lemma omega:
$\text{paradox}^f \equiv \text{lemma2}^f \text{ omega} \twoheadrightarrow_\beta \text{omega } I \text{ lemma}^f\ (\lambda \alpha.\text{omega } (\alpha \circ G)) \twoheadrightarrow_\beta \text{lemma}^f \text{ WF } P_1^f\ Q_1$.

We have the following two lemmas that basically give the looping combinator.

Lemma 3. *For all natural numbers n we have $Q_n\ I\ P_n^f =_\beta \mathit{lemma}^f\ \mathit{WF}_{n+1}\ P_{n+1}^f$ and*

$$\mathit{lemma}^f\ \mathit{WF}_n\ P_n^f\ Q_n =_\beta f(\mathit{lemma}^f\ \mathit{WF}_{n+1}\ P_{n+1}^f\ Q_{n+1})$$

Proof. Given a natural number n we have

$$\begin{aligned}
Q_n\ I\ P_n^f &=_\beta \text{omega}(\underbrace{I \circ \ldots \circ I}_{n+1} \circ G)\ P_n^f \\
&=_\beta P_n^f \text{ WF } (P_n^f \circ G) \\
&=_\beta \text{lemma}^f\ (G^n\,(\text{WF}))P_{n+1}^f \\
&=_\beta \text{lemma}^f \text{WF}_{n+1} P_{n+1}^f.
\end{aligned}$$

For the second part of the lemma, we have (using the first part of the lemma for the last equation):

$$\begin{aligned}
\text{lemma}^f \text{ WF}_n\ P_n^f\ Q_n &\equiv (\lambda \gamma p q.f\ (q\ I\ p\ (\lambda \alpha.q\ (\alpha \circ G)))) \text{ WF}_n\ P_n^f\ Q_n \\
&=_\beta f(Q_n\ I\ P_n^f\ (\lambda \alpha.Q_n\ (\alpha \circ G))) \\
&=_\beta f(Q_n\ I\ P_n^f\ Q_{n+1}) \\
&=_\beta f(\text{lemma}^f \text{ WF}_{n+1}\ P_{n+1}^f\ Q_{n+1})
\end{aligned}$$

Corollary 1. *The term* $\lambda\beta : \star.\lambda f : \beta \to \beta.paradox^f$ *is a looping combinator of sort* $\star$.

We now look at the erasure (definition 8) of the looping combinator ($|\lambda\alpha : \star.\lambda f : \alpha \to \alpha.\text{paradox}^f|$). For this we need to isolate the terms of type *a proposition* and erase the type information.

$$\begin{aligned}|omega| &= |\lambda\alpha : U \to \star.\lambda y : (\text{induct } \alpha).y \text{ WF } (\lambda\gamma : U.y\ (G\ \gamma))| \\ &= \lambda y.yy\end{aligned}$$

We see that the erasure of *omega* is $\lambda y.yy \equiv \omega$. In the same way we find that

$$\begin{aligned}|\text{lemma}^f| &\equiv \lambda pq.f(qpq) \\ |\text{lemma2}^f| &\equiv \lambda x.x|\text{lemma}^f|x \\ &\equiv \lambda x.x(\lambda pq.f(qpq))x \\ |\text{paradox}^f| &\equiv |\text{lemma2}^f|\ |\text{omega}| \\ &\equiv (\lambda x.x(\lambda pq.f(qpq))x)\ \omega\end{aligned}$$

The term $|\lambda\alpha.\lambda f.\text{paradox}^f|$ is a fixed point combinator in untyped lambda calculus. For ease of presentation, we define the term $M^f \equiv |\text{lemma}^f| \equiv \lambda pq.f(qpq)$ and we reduce $|\lambda\alpha.\lambda f.\text{paradox}^f|$ once obtaining the following easy to verify result.

Lemma 4. $\lambda f.(\omega\,(\lambda pq.f\,(q\,p\,q))\,\omega)$ *is a fixed point combinator in the untyped lambda calculus.*

Corollary 2. *There is a fixed point combinator in the Curry version of* λU^-.

Proof. In the Curry version of λU^-, the only abstractions are the first order abstractions that remain after the erasure. A term M is typable in the Curry version of λU^- iff there is a term N, typable in the Church version of λU^- with $|N| \equiv M$. □

4.3 Untypability of Ω

We now explore the claim of [CH94] that the usual direct proof of Ω's untypability for System F can be applied to λU^-.

Definition 9. *An untyped lambda term* M *is* **typable in** λU^- *iff there exist* Γ, t, P *such that* $\Gamma \vdash t : P$ *and* $|t| = M$.

We prove that the term Ω is not typable in λU^-. The result we obtain is even a bit stronger.

Theorem 2. *If the untyped term* M *contains a subterm* $(\lambda x.N)(\lambda y.P)$ *such that* N *contains a subterm* xx *and* P *contains a subterm* yy, *then* M *is untypable in* λU^-.

Remark 3. Note that types in λU^- (Kinds and Constructors) are SN, which means that we can safely assume types to be in normal form at all times.

We are going to extend the notion of *parse tree* for a type, known from [Joe99] for system F and extended to $F\omega$ in [Urz97].

Definition 10. *Given $\Gamma \vdash A : \star$, we define the* **parse tree** *of A (written $\mathrm{pt}(A)$) below. By remark 3 we may assume that A is in normal form.*

- *If $A \equiv Q \to R$ then*

$$\mathrm{pt}(Q \to R) = \begin{tikzcd} & \to \arrow[dl] \arrow[dr] & \\ \mathrm{pt}(Q) & & \mathrm{pt}(R) \end{tikzcd}$$

- *If $A \equiv \Pi\alpha : K_1.Q$ with α a constructor variable, then*

$$\mathrm{pt}(\Pi\alpha : K_1.Q) = \Pi\alpha : K_1 \;\; \mathrm{pt}(Q)$$

- *In all other cases ($A \equiv \alpha$, $A \equiv QR$ or $A \equiv QK_1$)*

$$\mathrm{pt}(A) = A$$

Definition 11. *A* **left-going path** *of a type is a path that has no branches to the right.*

Definition 12. *The* **left-most path** *of a type is the unique left-going path from the root of the type to a leaf. We will write* $\mathrm{lmp}(A)$ *for the left-most path of a type A*

Definition 13. *A variable α* **owns** *a path $X \in \{L, R\}^*$ in a type A if one of the following holds:*

- *$A = \alpha T_1 T_2 \ldots T_n$ with $n \geq 0$ and X is the empty sequence.*
- *$A = Q_1 \to Q_2$, and either $X = LX'$ and α owns X' in Q_1, or $X = RX'$ and α owns X' in Q_2.*
- *$A = \Pi\beta : K_1.Q$ and α owns X in Q.*

In the last case, α and β may be the same variable.

Remark 4. It is a consequence of the so called 'Stripping Lemma' (see e.g. [Geu93]) that, if $\Gamma \vdash t\,p : C$ with $\Gamma \vdash C : \star$, then $\Gamma \vdash t : A \to B$ and $\Gamma \vdash p : A$ for some types A and B.
Note that $\mathrm{length}(\mathrm{lmp}(A \to B)) = \mathrm{length}(\mathrm{lmp}(A)) + 1$.

We also define the *containment* relation ($\preceq$) for λU^-, as an extension of the notion for Fω in [Urz97].

Definition 14. *Given two types σ and τ, the relation $\sigma \preceq \tau$ holds iff $\sigma = \Pi\boldsymbol{\alpha}.\sigma'$, for some (possibly empty) vector $\boldsymbol{\alpha}$ and type σ' such that there are no quantifiers at the root of σ', and $\tau = \Pi\boldsymbol{\beta}.\sigma'[\boldsymbol{\rho}/\boldsymbol{\alpha}]$ for $\boldsymbol{\rho}$ of appropriate kinds and the variables in $\boldsymbol{\beta}$ do not occur free in σ.*

As in [Urz97], it is easy to see that this definition is a *quasi-order*, thus it's *reflexive* and *transitive.*

Lemma 5. *Given two types* σ, τ *with* $\sigma \preceq \tau$*, then* $\mathrm{length}(\mathrm{lmp}(\sigma)) \leq \mathrm{length}(\mathrm{lmp}(\tau))$.

Proof. This follows directly from the fact that the only parts of a type that are affected by type-application are the leaves, which can only expand. (The only way to reduce a tree is by proof term application.) $\square$

Corollary 3. *By the same reasoning, the entire tree structure of* σ *remains present in* τ*, thus for every path* $X \in \{R, L\}^*$ *in* σ*, there is a path* X' *in* τ *such that* X *is a prefix of* X'.

Lemma 6. *If* $\sigma \preceq \tau$ *and* $\mathrm{lmp}(\sigma)$ *is not owned by a variable quantified at the root of* σ*, then* $\mathrm{length}(\mathrm{lmp}(\sigma)) = \mathrm{length}(\mathrm{lmp}(\tau))$.

Proof. Given types τ, σ, such that $\sigma \preceq \tau$ and $\mathrm{lmp}(\sigma)$ is not owned by a variable quantified at the root of σ. Then by definition 14 there are $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\rho}, \sigma'$ such that $\sigma = \Pi \boldsymbol{\alpha}.\sigma'$ and $\tau = \Pi \boldsymbol{\beta}.\sigma'[\boldsymbol{\rho}/\boldsymbol{\alpha}]$. The variable at the leaf on the end of the left-most path is not replaced by the substitution $[\boldsymbol{\rho}/\boldsymbol{\alpha}]$, so the left most path has the same length. $\square$

Lemma 7. *If the proof term* $M : A$ *contains a proof term variable* $x : Q$ *which is used in a self application (i.e.* $|M|$ *contains the subterm* xx*), then* $\mathrm{lmp}(()Q)$ *is owned by a variable that is quantified at the root of* $\mathrm{pt}(()Q)$.

Proof. Given a proof term M, and a typing $\Gamma \vdash M : A$ such that $|M|$ contains the subterm xx. There is a type Q such that $x : Q$: this may be as a declaration in Γ or $\lambda x : Q.N$ is a subterm of M. The general form of the subterm xx in M is $x\boldsymbol{T}(\lambda \boldsymbol{\beta} : \boldsymbol{K}.x\boldsymbol{R})$. Say that $x\boldsymbol{T} : S_1$ and $\lambda \boldsymbol{\beta} : K_i.x\boldsymbol{R} : S_2$ then we know that the $\mathrm{length}(\mathrm{lmp}(S_1)) = \mathrm{length}(\mathrm{lmp}(S_2)) + 1$ (by remark 4).

Also $Q \preceq S_1$ and $Q \preceq S_2$ and if the $\mathrm{lmp}(()Q)$is not owned at the root, then $\mathrm{length}(()\mathrm{lmp}(()S_1)) = \mathrm{length}(()\mathrm{lmp}(()Q)) = \mathrm{length}(()\mathrm{lmp}(()S_2))$ as a consequence of Lemma!!. Contradiction, so $\mathrm{lmp}(Q)$ is owned by a variable quantified at the root of $\mathrm{pt}(()Q)$. $\square$

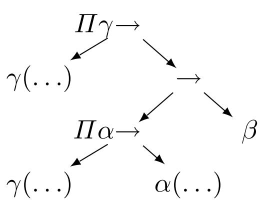
Proof. of the Theorem Given an untyped term M containing a subterm $(\lambda x.N)(\lambda y.P)$ such that N contains a subterm xx and P contains a subterm yy. There are Γ, t, A such that $\Gamma \vdash t : A$ and $|t| = (\lambda x.N)(\lambda y.P)$. Thus, there are λU^- terms p, q such that pq is a subterm of t, $|p| = (\lambda x.N)$ and $|q| = (\lambda y.P)$. There are types Q, R such that $\Gamma \vdash p : Q \to R$ and $\Gamma \vdash q : Q$. Because p was created by a lambda abstraction on the variable x, we know that $x : Q$. Because xx is a subterm of N, we know that $\mathrm{lmp}(Q)$ is owned by a variable quantified at the root of Q by lemma 7. Because q is created by a lambda abstraction on the variable y, we know that Q is a tree with an arrow type of which the left hand side is the type of $y : S$ for some S. Because yy is a subterm of P, we know that $\mathrm{lmp}(S)$ is owned by a variable quantified at the root of S. However, this means that $\mathrm{lmp}(Q)$ is owned by a variable quantified at the left hand side of the root arrow, and not at the root, which is a contradiction. Thus, M is not typable. $\square$

Corollary 4. *The well-known untyped* λ*-terms* Ω, Y *and* Θ *are not typable in* λU^-.

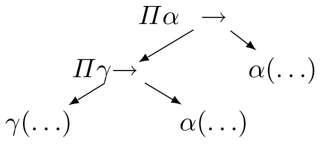
One may try to use the definition of parse trees for types (Definition 10) to show that it is not possible to type $L = \lambda f.(\lambda x.x(\lambda pq.f(qpq))x)(\lambda y.yy)$ in λU^- as a fixed-point combinator (So it can only be typed as a real looping combinator.) For example by showing that the parse trees of the subterms expand, and that therefore the types get larger and larger. However, we were not able to obtain such a result through parse tree analysis. The reason for this is that for the looping combinator we have analyzed in section 4, the trees of the types don't change in size, but only the information in the leafs.

When analyzing the possible typings of $L = \lambda f.(\omega(\lambda pq.f(qpq))\omega)$, we can leave out f and ask ourselves the question whether the typed version of this term $L' := \omega(\lambda pq.qpq)\omega$ is loops or not. We will write $M = \lambda pq.qpq$, which gives us $L' = \omega M \omega$. When analyzing the infinite reduction of the typed version of L' in section 4, we see that every time that M and ω come to the head of the term, their type has a similar parse tree:

M :



ω :



5 Conclusion

We are confident that our results hold also for λU, so also there, Y, Ω and Θ are not typable. For $\lambda\star$, the situation is very much open. The techniques that we have applied here don't work, because types are not SN in $\lambda\star$.

Another interesting question that remains is whether a fixed-point combinator exists at all in λU^- and whether the term L can be typed as a fixed point combinator in λU^-. We conjecture that no fixed point combinator exists in λU^-. Although we have not been able to prove this, the work here shows that seeing types as trees isolates a lot of useful structure from them. This makes the types appear less wild, as the branches of the tree often remain unchanged when the type is manipulated. As the tree structure is a graphic representation of the prop 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 the discussion on the syntactical proof of non typability of Ω in System F.

References

[BAG$^+$92] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[BB98] Henk Barendregt and Erik Barendsen. Introduction to Lambda Calculus. 1998.

[Br95] Gilles Barthe and Morten Heine Sørensen. Domain-free pure type systems. In *Proceedings of TLCA'95, volume 902 of Lecture Notes in Computer Science*, pages 9–20. Springer-Verlag, 1995.

[CH94] Thierry Coquand and Hugo Herbelin. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, 4:77–88, 1994.

[Coq94] Thierry Coquand. A new paradox in type theory. In *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.

[Geu93] Herman Geuvers. *Logics and Type Systems.* PhD thesis, Radboud University, Nijmegen, 1993.

[Geu07] Herman Geuvers. Inconsistency of classical logic in type theory, 2007.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[How87] D. J. Howe. The computational behaviour of Girard's paradox. In *Proceedings of the 2nd Symposium on Logic in Computer Science*, pages 205–214. IEEE, 1987.

[Hur95] Antonius J. C. Hurkens. A simplification of girard's paradox. In *TLCA '95: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 266–278, London, UK, 1995. Springer-Verlag.

[Joe99] Joe B. Wells. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1999.

[Miq00] Alexandre Miquel. Russell's Paradox in System U^- minus. TYPES 2000 - Durham, 11 2000.

[Rei86] Mark B. Reinhold. Typechecking is undecidable when 'type ' is a type, 1986.

[S.C36] S.C.Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

[Urz97] Paweł Urzyczyn. Type reconstruction in Fω. *Mathematical. Structures in Comp. Sci.*, 7(4):329–358, 1997.