The Problem with "Type in Type" and a resolution thereof

Abstract

In "A Tutorial Implementation of a Dependently Typed Lambda Calculus" (A. Löh et al., 2010) a simple dependently typed lambda calculus and a straightforward implementation of type inference and checking for it are presented. In keeping the type rules and implementation simple, they made the deliberate choice to include the rule "type in type", which is commonly known to be inconsistent. Here, this inconsistency will be explored and an extension to the type rules and it's implementation will be presented which resolve it.

Overview

The type system for which A. Löh et al. [Löh 2010] presented their straightforward implementation in Haskell, which we shall call $\lambda_{\Pi}^{\tau\tau}$ and which will be taken as the basis of the following discussion, is a direct extension of the simply typed lambda calculus (STLC), with function types, $\tau \to \tau'$, extended to dependent function types, $\forall x : \rho.\rho'$, and the distinction between ordinary terms and type terms being dissolved. From this results the necessity for a term which is the type of all types, $*$, which in turn of course also requires a type itself. The perhaps most straightforward choice to be made here is to consider the type of $*$ to be $*$ itself ("type in type"). Just like in Martin-Löf's 1971 "A Theory of Types" [Martin-Löf 1971] this is in fact the choice that A. Löh et al. made. However, contrary to Martin-Löf in 1971, they did so in full knowledge that this results in an inconsistent type system, as was shown by Girard in 1972 [Girard 1972], to keep the type system and it's implementation simple.

The inconsistency arising from $* : *$ will be the focus of the first part of this paper. However, instead of Girard's original proof, a much simplified construction due to Hurkens [Hurkens 1995] will be discussed, which in the following shall be called "Hurkens' paradox". From this paradox we will arrive at (relatively) compact concrete term of type $\forall A : *.A$, which should be impossible in a consistent type system, given that by applying this term to any possible type we receive a term of that type, including any definitionally empty types. So, looked at through the Curry-Howard Correspondence, where we take types to represent propositions and terms to represent proofs, this entails that we can provide a proof for every possible proposition, which clearly would make such an implementation of little use as the basis for a proof assistant.

While there are different ways of resolving the inconsistency arrising from $* : *$, in the second part of this paper, one possible solution, namely a "hierarchy of sorts" [Coquand 1986] will be introduced, and an extended version of the lambda calculus and typing implementation presented by A. Löh et al., which we shall call λ_{Π}^{ω}, will be presented.

Hurkens' Paradox

In 1995, Antonius J.C. Hurkens derived, based upon work by Girard [Girard 1972] and Coquand [Coquand 1991], a relatively compact term of type $\bot$ in λU^- [Hurkens 1995]. While the type system of λU^- goes beyond the type system of $\lambda_{\Pi}^{\tau\tau}$, his construction can be followed one-to-one, giving us a term of type $\bot$ in $\lambda_{\Pi}^{\tau\tau}$, proving the type system's inconsistency, which we shall do in the following.

Though Hurkens showed two different approaches to simplifying Girard's paradox, the one for which he provided a complete term of type $\bot$ is based upon the concept of "powerful universes", and will be the one explored here.

While the goal in the end will be to implement the paradox in the mentioned implementation of $\lambda_{\Pi}^{\tau\tau}$, for readability and convenience, in the following, the syntax of the dependently typed programming language Agda [?] will be used in the explanation of the paradox.

For a complete and annotated Agda source file implementing Hurkens' paradox, a translation thereof into the abstract syntax of our implementation of $\lambda_{\Pi}^{\tau\tau}$ in Haskell, and also the fully expanded term of type $\bot$ in Haskell, refer to the code repository associated with this paper [?].

In the following, each type theoretic proof, given in Agda syntax, will be accompanied with a corresponding elaborated set theoretic proof which shall aid in understanding of the proof term and it's derivation.

Basic Definitions

In absence of record types, we will, as is common, define the empty type $\bot$ and negation $\neg$ as follows:

```
⊥ : Set
⊥ = (A : Set) → A

¬ : Set → Set
¬ P = P → ⊥
```

A term of type $\bot$ would have to be a function that could produce a term for any possible type, i.e. a proof of every possible proposition. Therefore this type has to be empty for the type system to be consistent.

A term of type $\neg P$ for some proposition P is simply a function which, if a proof of P were given, would produce a term of type $\bot$. Therefore, if we can construct a term of type $\neg P$, then this means we can not possibly produce a term of type P, meaning that the proposition P must not be true. At least that is the case in a consistent type system.

This gives us the first hint as to how we could derive a term of type $\bot$. If we can come up with some proposition P for which we can both derive a term of type P and of type $\neg P$, then we simply have to apply $\neg P$ to P and we will have our term of type $\bot$ in hand. In fact, this will be exactly what we shall do in the end, however, first we have to come up with such a proposition.

For ease of readability and conceptual understanding, we shall also define a function for the type of all propositions over some type A. From a set-theoretic perspective, this is to be understood as the set of all subsets for some set A, i.e. it's powerset:

```
℘ : Set → Set
℘ A = A → Set
```

This powerset function will be made extensive use of in the following to allow us to build up our paradox.

A Powerful Universe

The first significant definition for Hurkens' Paradox is an instance of a *powerful universe*, which we shall consider essentially plucked out of thin air, in the knowledge that it will allow us to derive our contradiction:

```
U : Set
U = (A : Set) → (℘ (℘ A) → A) → ℘ (℘ A)

τ : ℘ (℘ U) → U
τ ppU A ppA→A pA = ppU (λ u → pA (ppA→A (u A ppA→A)))

σ : U → ℘ (℘ U)
σ u = u U τ
```

This triple of (U, σ, τ) we consider to be *powerful*, since it satsifies, in set theoretic terms, the following property:

$$\forall C \in \wp(\wp U) : \sigma(\tau C) = \{X | \, \{y|\tau(\sigma y) \in X\} \in C\}$$

We will not concern ourselves with translating this property into type theory or proving that this property holds for our (U, σ, τ) (see Hurkens' original derivation [Hurkens 1995] for some elaboration on the definition of a powerful universe), since such a proof term will not be necessary in constructing our paradox. Rather we will implicitly use this property as it arises from the behaviour of τ and σ as defined above.

Inductive Subsets and *Well Founded* Elements

For subsets of U we define the following proposition:

```
inductive' : ℘ (℘ U)
inductive' pU = ((u : U) → σ u pU → pU u)
```

In set theoretic terms this means that for some subset pU of U, we consider pU to be *inductive* iff the following property holds:

$$\forall u \in U : (pU \in \sigma u \Rightarrow u \in pU)$$

Using this property over subsets of U, we define a proposition over elements of U:

```
well-founded : ℘ U
well-founded u = (pU : ℘ U) → inductive' pU → pU u
```

In set theoretic terms this means that we consider some element u of U to be *well founded* iff it is in every inductive subset of U.

A Paradoxical Element

With τ from our definition of U as a powerful universe, we pick out a specific element in U for which we can show that it simultaneously is well founded, and isn't well founded, which will give us the contradiction we seek:

```
Ω : U
Ω = τ inductive'
```

This means that in set theoretic terms, we consider Ω to be $\tau\ \{pU \in \wp U | pU\ is\ inductive\}$.

The Paradoxical Element is well founded

The proof that Ω is well founded is relatively straightforward:

```
well-founded-Ω : well-founded Ω
well-founded-Ω pU ind-pU = ind-pU Ω (λ u → ind-pU (τ (σ u)))
```

In set theoretic terms, we need to show that for any inductive subset pU of U, Ω is in pU. Knowing that pU is inductive, this means that we need to show that pU is in $\sigma\Omega$.

Since U is powerful, $\sigma\Omega = \{pU \in \wp U | \ \{u \in U | \tau(\sigma u) \in pU\}\ \ is\ inductive\}$. So to show that our pU is in $\sigma\Omega$, we need to show that $\{u \in U | \tau(\sigma u) \in pU\}$ is inductive.

Since pU is inductive, we known that for any $u \in U$, if pU is in $\sigma(\tau(\sigma u))$, then $\tau(\sigma u)$ is in pU. But this already is exactly what we need to show to prove that $\{u \in U | \tau(\sigma u) \in pU\}$ is inductive, therefore our proof is complete.

But also not well founded

To construct the proof that Ω is at the same time not well founded, we define one more concept:

```
_<_ : U → U → Set
v < u = (pU : ℘ U) → σ u pU → pU v
```

For some $u \in U$, we consider some $v \in U$ to be a *predecessor* of u iff for every subset pU of U, pU being in σu implies that v is in pU.

With this concept, we define ourselves one specific subset of U, which will turn out to be inductive:

```
Δ : ℘ U
Δ u = ¬ (τ (σ u) < u)
```

So in set theoretic terms, $\Delta = \{u \in U | \tau(\sigma u) \not< u\}$.

We prove that Δ is inductive:

```
inductive-Δ : inductive' Δ
inductive-Δ u σuΔ τσu<u = τσu<u Δ σuΔ (λ pU → τσu<u λ w → pU (τ (σ w)))
```

To show that Δ is inductive, we need to show that for any $u \in U$, if Δ is in σu then u is in Δ, so $\tau(\sigma u) \not< u$.

So for any $u \in U$ with $\Delta \in \sigma u$, we assume $\tau(\sigma u) < u$ and arrive at a contradiction from this assumption as follows:

$\tau(\sigma u) < u$ means that for any pU in σu, $\tau(\sigma u)$ is in pU. If we take for pU Δ itself, this means that $\tau(\sigma u)$ is in Δ, so $\tau(\sigma(\tau(\sigma u))) \not< \tau(\sigma u)$.

On the other hand we can show that $\tau(\sigma(\tau(\sigma u))) < \tau(\sigma u)$ as follows:

For any subset pU of U we have to show that if pU is in $\sigma(\tau(\sigma u))$, then $\tau(\sigma(\tau(\sigma u)))$ is in pU. However, since U is powerful, this simplifies to having to show that if $\{w|\tau(\sigma w) \in pU\}$ is in σu, then $\tau(\sigma u)$ is in $\{w|\tau(\sigma w) \in pU\}$. But that follows directly from our initial assumption that $\tau(\sigma u) < u$. Therefore our proof is complete.

With Δ and the knowledge that it is inductive in hand, we can at last prove that Ω is not well founded, which will complete the contradiction we seek:

```
¬well-founded-Ω : ¬ (well-founded Ω)
¬well-founded-Ω wfΩ = wfΩ Δ inductive-Δ (λ pU → wfΩ (λ w → pU (τ (σ w))))
```

To show that Ω is not well founded, we assume that it is, and will from this derive a contradiction:

Since Δ is inductive and Ω well founded, this means that Ω is in Δ, and therefore $\tau(\sigma\Omega) \not< \Omega$. On the other hand, we can show that $\tau(\sigma\Omega) < \Omega$ as follows:

For any subset pU of U we have to show that if pU is in $\sigma\Omega$, then $\tau(\sigma\Omega)$ is in pU. However, since U is powerful and Ω was defined as $\tau\ \{pU \in \wp U|pU\ is\ inductive\}$, this simplifies to having to show that if $\{w|\tau(\sigma w) \in pU\}$ is inductive, then Ω is in $\{w|\tau(\sigma w) \in pU\}$. But that follows directly from our initial assumption that Ω is well founded. Therefore our proof is complete.

A term of the empty type

With both a proof that Ω is well founded and a proof that Ω is not well founded in hand, we can at last construct the term of type $\bot$:

```
false : ⊥
false = ¬well-founded-Ω well-founded-Ω
```

This concludes the proof that λ_{Π}^{π} is inconsistent.

A Hierarchy of Sorts

As evident from the construction of a contradiction in λ_{Π}^{π} presented above, it is necessary for the expressiveness of our dependently typed lambda calculus to be weakened in some way for it to be consistent. However, we simultaneously do not want to give up the ability to express propositions of practical interest and their proofs in our lambda calculus.

Luckily, a rather simple modification to the type system of λ_{Π}^{π} is sufficient to make it consistent again [Martin-Löf 1973][Coquand 1986], replacing the problematic type rule $* : *$ by a *hierarchy of sorts*:

$$\begin{gathered} * : *_1 \\ *_1 : *_2 \\ *_2 : *_3 \\ *_3 : *_4 \\ \dots \end{gathered}$$

So just like in λ_Π^{π} every object term, like $true$, has some type, like $Bool$, and every type term has the kind $*$. However, the term $*$ itself does not have the kind $*$, but the *sort* $*_1$, the term $*_1$ has the sort $*_2$, the term $*_2$ has the sort $*_3$ and so on.

In the following, we will use for consistency instead of $*$ as the sort of types $*_0$.

This lambda calculus we shall call λ_Π^ω. The grammar and type rules for it are as follows:

$$\begin{array}{rl} e, \rho ::= & e : \rho \\ | & x \\ | & e\ e' \\ | & \lambda x.e \\ | & *_\ell \\ | & \forall x : \rho.\rho' \end{array}$$

$$\frac{\Gamma \vdash e :_\uparrow \tau}{\Gamma \vdash e :_\downarrow \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}$$

$$\frac{\Gamma, x : \tau \vdash e :_\downarrow \tau'}{\Gamma \vdash \lambda x.e :_\downarrow \forall x : \tau.\tau'}$$

$$\frac{\Gamma \vdash e :_\uparrow \forall x : \tau.\tau' \quad \Gamma \vdash e' :_\downarrow \tau}{\Gamma \vdash e\ e' :_\uparrow \tau'[\,x \mapsto e'\,]}$$

$$\frac{\Gamma \vdash \rho :_\uparrow *_\ell \quad \rho \Downarrow \tau \quad \Gamma \vdash e :_\uparrow \tau}{\Gamma \vdash (\,e : \rho\,) :_\uparrow \tau}$$

$$\frac{\Gamma \vdash \rho :_\uparrow *_\ell \quad \rho \Downarrow \tau \quad \Gamma, x : \tau \vdash \rho' :_\uparrow *_{\ell'}}{\Gamma \vdash \forall x : \rho.\rho' :_\uparrow *_{max(\ell,\ell')}}$$

$$\frac{}{\Gamma \vdash *_\ell :_\uparrow *_{\ell+1}}$$

Only the last three rules differ from λ_Π^{π}, and only the last two do so substantially. Also of note is that some type judgements have turned from type checking to type inference due to the need to know the *level* ℓ for a sort $*_\ell$, which will in turn necessicate corresponding adaptations in the implementation.

The last type rule, $*_\ell : *_{\ell+1}$, is a direct implementation of the hierarchy of sorts as explained above.

However of course it is also necessary to reconsider type judgements over terms $\forall x : \rho.\rho'$ in the second to last type rule.

One option in defining the type rule for $\forall x : \rho.\rho'$ would be to simply directly keep the rule from λ_Π^{π} in our new type system:

$$\frac{\Gamma \vdash \rho :_\downarrow *_0 \quad \rho \Downarrow \tau \quad \Gamma, x : \tau \vdash \rho' :_\downarrow *_0}{\Gamma \vdash \forall x : \rho.\rho' :_\uparrow *_0}$$

While this would be consistent (since it is simply a strictly less powerful type system than that of λ_Π^ω), and would have the advantage of that we would only have to check that the kind of ρ and ρ' is $*_0$, it would greatly restrict the expressiveness of our language, since it would mean that a function could not take a type as an argument or return a type as it's result, it could only go from objects to objects.

Instead we will allow for functions to go from any sort to any sort, from objects to types, from types to objects, from kinds to objects, from objects to kinds, etc. :

$$\frac{\Gamma \vdash \rho :_{\uparrow} *_{\ell} \quad \rho \Downarrow \tau \quad \Gamma, x : \tau \vdash \rho' :_{\uparrow} *_{\ell'}}{\Gamma \vdash \forall x : \rho.\rho' :_{\uparrow} *_{max(\ell,\ell')}}$$

So the sort of some term $\forall x : \rho.\rho'$ is simply the higher of the sorts of ρ and ρ'.

Taking $\rho \to \rho'$ as a shorthand for $\forall x : \rho.\rho'$ with x not apeparing in ρ', this means for example that a term $*_7 \to *_3$ is of the sort $*_8$, since $*_7 : *_8$ and $*_3 : *_4$.

As a more practical example, the "powerset" function $\wp$ we defined above for our construction of Hurkens' paradox, $\wp A := A \to *_0$, would have the sort $*_0 \to *_1$.

This might already hint towards how the construction of Hurkens' paradox presented above will no longer work in λ_{Π}^{ω}, given that it heavily relies on the fact that in $\lambda_{\Pi}^{\tau\tau}$, $\wp : * \to *$.

However, the definition of the function $\wp$ above might also raise the question what to do if we do not want to look at the type of all propositions over some type, but rather over some kind, or over some higher sort. Do we define $\wp_1 K := K \to *_0$ of sort $*_1 \to *_2$, $\wp_2 S := S \to *_0$ of sort $*_2 \to *_3$, and so on?

In the type system presented here, we do not have any other choice but to go the above route. However, there are possible extensions to alleviate this redundancy of definitions, namely *universe polymorphism* [Sozeau et al. 2014]. However not only does this require levels to become terms inside the lambda calculus itself, but also brings with it the need to introduce a second hierarchy of sorts for level polymorphic function types. We will not implement this here.

Implementation

With the introduction of hierarchy of sorts having necessitated that some judgements in our type rules have turned from type checking to type inference, the implementation of $\lambda_{\Pi}^{\tau\tau}$ has to be changed in quite a few places. However, most of these changes are not terribly significant. Rather, we will focus here on the changes to the implementation due to the new type rules for $e : \rho$, $\forall x : \rho.\rho'$, and $*_{\ell}$.

The Abstract Syntax Tree Data Type

```haskell
data TermInfer
  = Ann TermCheck TermInfer
  | Star Int
  | Pi TermInfer TermInfer
  | Bound Int
  | Free Name
  | TermInfer :@: TermCheck
  deriving (Show, Eq)
```

`Star` now no longer simply is a constant constructor, but has an argument, it's level. And both `Ann` and `Pi` now have in places a `TermCheck` replaced with `TermInfer` due to the necessary changes to the type rules.

typeInfer for Ann

```
typeInfer i g (Ann e r) =
  do
    s <- typeInfer i g r
    case s of
      (VStar l) -> do
        let t = evalInfer [] r
        typeCheck i g e t
        return t
      _ -> failure ":("
```

We infer the type of `r`, which has to be some `VStar l` or we fail. Otherwise we proceed as in the implementation of $\lambda_{\Pi}^{\tau\tau}$, evaluating `r` to `t` and checking `e` against `t`.

typeInfer for Pi

```
typeInfer i g (Pi r r') =
  do
    s <- typeInfer i g r
    case s of
      (VStar l) -> do
        let t = evalInfer [] r
        s' <-
          typeInfer
            (i + 1)
            ((Local i, t) : g)
            (substInfer 0 (Free (Local i)) r')
        case s' of
          (VStar l') -> return (VStar (max l l'))
          _ -> failure ":("
      _ -> failure ":("
```

We infer the type for `r`, which has to be some `VStar l` or we fail. Otherwise, we evaluate `r` to `t` and infer the type of `r'` in the extended context, which again has to be some `VStar l'`. With both the sort level `l` of `r` and `l'` of `r'` determined, we can return the sort of our `Pi` term, `VStar (max l l')`.

And that is all.