# Homework 10
# Pure Type Systems

98-317: Hype for Types

Due: 30 Apr 2019 at 6:30 PM

## 1 Introduction

For the past two weeks, we have been discussing high-powered type systems. Beginning at the simply typed $\lambda$-calculus, $\boldsymbol{\lambda}_{\rightarrow}$, we have introduced the following ideas:

- *Quantification* of types on other types;
- *Higher-order* types, acting like functions on types;
- *Dependency* of types on values.

The lambda cube captured the idea of these three dimensions being orthogonal to each other. Some well-known type theories based on these ideas include System $\mathbf{F}$, with universally quantified types; System $\mathbf{F}_{\boldsymbol{\omega}}$, with quantification and type operators; and the Calculus of Constructions, with all three features.

In this assignment, we will examine a $\lambda$-calculus with the following features: *dependent types* and *first-order type operators*. The type operators being first order implies that this system lies somewhere between $\mathbf{F}$ and $\mathbf{F}_{\boldsymbol{\omega}}$ along that dimension; it is possible here to express *functions on types*, but not higher-order functions, and that is why we are merely first-order.

The name of this type theory is $\boldsymbol{\lambda\Pi}$. It is the metatheory of the Edinburgh Logical Framework, and for that reason it is also commonly known by the name LF. LF is a powerful theory useful for logic programming and computer-assisted proof as well as automated theorem proving. The Twelf (`http://twelf.org`) project is an implementation of LF from CMU.

Though LF lacks quantification, do not be fooled into thinking that it is not a very powerful type system. The presence of dependent types means that LF is capable of expressing many interesting things, including entire other type systems. In this assignment, we will be exploring an embedding of the simply typed $\lambda$-calculus into LF. In essence, a system like LF can act as a metatheory, in which the rules of another type system like $\boldsymbol{\lambda}_{\rightarrow}$ may be embedded. This enables rapid prototyping of type systems and makes testing and proving properties about them easier. Such a framework means we can reuse syntax handling and

type checking features, bringing us well on the way to having a prototype interpreter very quickly.

LF is one of many type systems that are good at serving as a type metatheory. The type system with quantification, type operators, and dependent types is known as the Calculus of Constructions. It sits at the top-right corner of the lambda cube. It, and its extension the Calculus of Inductive Constructions, are the basis of many logical tools. The popular theorem prover Coq, for example, is based on and was named after the Calculus of Inductive Constructions.

**Source** The material in this assignment is heavily derived from *Advanced Topics in Types and Programming Languages* by Benjamin Pierce, chapter 2. You may find reading that chapter to be helpful, as it thoroughly explains this type system.

**Turning in the Homework** You should submit your code solutions in the file `terms.sml` to Autolab.

# 2 The λΠ Type Theory

**λΠ** has four important concepts: *terms*, *types*, *kinds*, and *contexts*. Essentially, terms and types correspond to the same concepts as in the simply-typed $\lambda$-calculus. Recall that kinds are the "types of types", and characterize type-level operators and functions (type families). We have seen contexts all semester, and here they will link not only term variables to their types, but also type variables to their kinds. The type system involves dependent products ($\Pi$) at the type and kind level. Because of this, typechecking can be difficult, and type inference is in fact undecidable. We will only go through the basics of the type system so you understand what we will be working with.

Here is the syntax:

| | | | | |
|---|---|---|---|---|
| Terms | $e$ | $::=$ | $x$ | variables |
| | | $\mid$ | $\lambda\,(x : \tau)\,e$ | abstractions |
| | | $\mid$ | $e_1\ e_2$ | applications |
| Types | $\tau$ | $::=$ | $t$ | type variables |
| | | $\mid$ | $\Pi\,(x : \tau)\,\tau'$ | dependent product type |
| | | $\mid$ | $\tau\ e$ | type family application |
| Kinds | $\kappa$ | $::=$ | $*$ | kind of proper types |
| | | $\mid$ | $\Pi\,(x : \tau)\,\kappa$ | kind of type families |
| Contexts | $\Gamma$ | $::=$ | $\cdot$ | empty |
| | | $\mid$ | $\Gamma, x : \tau$ | term binding |
| | | $\mid$ | $\Gamma, t : \kappa$ | type binding |

In addition, when a dependent product is redundant, we use a more compact notation. $\alpha \to \beta$ is shorthand for $\Pi\,(x : \alpha)\,\beta$ when $x$ does not appear in $\beta$.

The judgments and rules of the type system are complex and you do not need to understand them. However, here are some important characteristics:

- Term typing proceeds as in $\boldsymbol{\lambda}_{\boldsymbol{\to}}$. Abstractions (functions) have dependent type.
- Type variables and type families are not generated during the typechecking of a program. Instead, the programmer specifies an *initial context* that declares a set of type variables, each of which is associated with a kind. They are "baked in" to the system. This initial context may also include term variables associated with types.
- Type family application works in the same way as function application, except that the argument is a term that is substituted for the term variable in a dependent type.
- The dependent $\Pi$ type has kind $*$. It does *not* have dependent kind. This is because a $\Pi$ type is a proper type. The only things that have dependent kind are *type family constructors*. What is such a constructor? In SML, `list` is an example of a type family constructor, which builds a type from another type. We could say that `list`

has kind $* \to *$.[1] Therefore, perhaps "type" is not the ideal name for all $\tau$. Only $\tau : *$ are proper types; the remainder might be better called "type constructors". These constructors with kind other than $*$ show up only in the initial context.

You might be wondering why, even in the presence of dependent types, typechecking is decidable. The reason is that terms, types, and kinds are *strongly normalizing*, i.e. there are no non-terminating computations and every $\lambda$-term can be reduced to a normal form. This is a consequence of the fact that $\boldsymbol{\lambda}_{\to}$ is strongly normalizing, and does not hold for systems with dependent types in general.

# 3 Propositions as Types

The Curry-Howard correspondence relates types to propositions and programs to proofs. In the presence of dependent types, we gain even more power to write logical propositions and prove them simply by writing functional programs that typecheck.

Previously, we have equated propositions such as $A \implies B$ with types such as $\alpha \to \beta$. Terms inhabiting the type, in this case functions taking $\alpha$ and returning $\beta$, serve to prove the proposition. A function is a transformation from a proof of $A$ into a proof of $B$. This perfectly captures propositional, or zeroth-order, logic.

However, we may want to work in a logic with predicates like $P(A)$, and quantification over propositions. With dependent types, we may encode quantification using the dependent product. The proposition

$$\forall\,(x : A)\,P(x)$$

becomes the dependent product

$$\Pi\,(x : \alpha)\,P(x)$$

The ability to treat propositions as types has impact on programming. We may make logical statements by writing them as types and prove them (or request a proof) by passing around terms inhabiting the types. An example application is in the definition of dependently-typed vectors with a fixed length $n$ encoded into the type. This type might be denoted `Vector` $t\ n$ where $n \in \mathbb{N}$, and we could define a safe indexing operation `sub` with the type:

$$\Pi\,(n : \mathbb{N})\ \texttt{Vector}\ t\ n \to\ \Pi\,(l : \mathbb{N})\ \texttt{Lt}\ l\ n \to t$$

As you can see, we make use of a new dependent type `Lt` $l\ n$ which is the proposition $l < n$. The indexing operator is dependent on the length $n$ of the vector and the index $l$, requests a proof that $l < n$, and then extracts an element from the vector. In a dependently typed system, such a type would be extremely powerful and statically encode requirements that hold on functions and data structures. Unfortunately, few practical programming languages have support for dependent types, due to the lurking danger of undecidability and the difficulty in checking dependent types in general.

[1] Though SML relies on $\mathbf{F}_{\boldsymbol{\omega}}$, which is a different type system, the idea is similar here. In particular, in $\boldsymbol{\lambda\Pi}$ the left side of the $\to$ should be a type, not a kind, because we only have first-order operators.

# 4 $\boldsymbol{\lambda_{\to}}$ in LF

We will now describe an embedding of $\boldsymbol{\lambda_{\to}}$ into the LF type system. Since the $\lambda$-calculus has types and terms, we will represent each "meta-type" as a type in LF.

$$\begin{array}{lcl} \mathsf{Ty} & : & * \\ \mathsf{Tm} & : & \mathsf{Ty} \to * \end{array}$$

$\mathsf{Ty}$ is the meta-type of $\lambda$-calculus types. For a lambda calculus type $\alpha$, $\mathsf{Tm}\ \alpha$ is the meta-type of $\lambda$-calculus terms with that type.

Now we define the types of $\lambda$-calculus.

$$\begin{array}{lcl} \mathsf{base} & : & \mathsf{Ty} \\ \mathsf{arrow} & : & \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \end{array}$$

The simply typed $\lambda$-calculus normally only has only one type constructor, $\to$. However this poses a problem since then there is no base case, so we add a type called $\mathsf{base}$.

Finally we define the two term constructors.

$$\begin{array}{lcl} \mathsf{app} & : & \Pi\,(A : \mathsf{Ty})\,\Pi\,(B : \mathsf{Ty})\,\mathsf{Tm}\ (\mathsf{arrow}\ A\ B) \to \mathsf{Tm}\ A \to \mathsf{Tm}\ B \\ \mathsf{lam} & : & \Pi\,(A : \mathsf{Ty})\,\Pi\,(B : \mathsf{Ty})\,(\mathsf{Tm}\ A \to \mathsf{Tm}\ B) \to \mathsf{Tm}\ (\mathsf{arrow}\ A\ B) \end{array}$$

Both of these term constructors are dependent on the function type $A \to B$, and take $A$ and $B$ as dependent parameters. Then, the difference between applications and abstractions is made clear in the symmetry of the remainder of the type. $\mathsf{Tm}\ (\mathsf{arrow}\ A\ B)$ is the type of terms representing functions from $A$ to $B$, $\mathsf{Tm}\ A$ is the type of terms of type $A$, and $\mathsf{Tm}\ B$ is the type of terms of type $B$. So applications take a function and apply it to a term within the codomain, yielding a term within the domain. Abstractions take a *mapping* from the codomain to the domain, and build a function term representing that mapping. As Pierce describes, this technique of lifting the $\to$ in LF into the $\mathsf{arrow}$ constructor in the $\boldsymbol{\lambda_{\to}}$ implementation is called *higher-order abstract syntax*. We now have a way of seeing how abstraction and application are intimately related to each other.

In the code handout, you are provided with an implementation of LF, its typechecker, and the embedding of $\boldsymbol{\lambda_{\to}}$. You can see the abstract syntax of LF in `check-lf.sig`, and the signature of its typechecker in `check-lf.sig`. The implementation of the typechecker is `impl/check-lf.sig`. The signature of the $\boldsymbol{\lambda_{\to}}$ embedding is in `stlc.sig`, and its implementation is in `impl/stlc.sml`. You are encouraged to take a peek at these implementations to see how the type theory is implemented.

Your task will be to write some code in the embedded $\lambda$-calculus. In `terms.sml` you will see the descriptions of the terms that you should implement. In `main.sml` you will see the routine which compares the terms to their expected types. To run the checker, load `sources.cm` and run `Main.check ()` in the REPL.

Specifically, you can see in `terms.sml` the implementation of the polymorphic identity function as well as the Church encoding of the number 1. These are the encodings:

$$\begin{array}{rcl} \mathsf{id} & : & \alpha \to \alpha \\ \mathsf{id} & \triangleq & \lambda\,(x : \alpha)\,x \\ \mathsf{one} & : & \alpha \to (\alpha \to \alpha) \to \alpha \\ \mathsf{one} & \triangleq & \lambda\,(x : \alpha)\,\lambda\,(f : \alpha \to \alpha)\,f\ x \end{array}$$

We must translate the encoding into the embedded $\boldsymbol{\lambda}_{\to}$ in LF. These are the translations:

$$\begin{array}{rcl} \mathsf{id} & \triangleq & \mathsf{lam}\ A\ A\ (\lambda\,(x : \mathsf{Tm}\ A)\,x) \\ \mathsf{one} & \triangleq & \mathsf{lam}\ A\ (\mathsf{arrow}\ (\mathsf{arrow}\ A\ A)\ A) \\ & & \quad (\lambda\,(x : \mathsf{Tm}\ A)\,\mathsf{lam}\ (\mathsf{arrow}\ A\ A)\ A\ (\lambda\,(f : \mathsf{Tm}\ (\mathsf{arrow}\ A\ A))\,\mathsf{app}\ A\ A\ f\ x)) \end{array}$$

Since we do not have real polymorphism in $\boldsymbol{\lambda}_{\to}$, we simply concoct some type variables $A, B, C, \ldots$ and instruct the typechecker to accept the variables in the initial context. The identity function is implemented as an abstraction from $A$ to $A$ whose implementation maps a term of type $\mathsf{Tm}\ A$ to itself. Try to match the implementation of $\mathsf{one}$ with the definitions of $\mathsf{app}$ and $\mathsf{lam}$ to see how its translation works and typechecks at the levels of LF and the embedded $\boldsymbol{\lambda}_{\to}$.

**Task.** You are asked to implement the Church encodings of Boolean true and false, as well as the constructor for pairs.

Here are the encodings. You must translate them into LF:

$$\begin{array}{rcl} \mathsf{btrue} & : & \alpha \to \alpha \to \alpha \\ \mathsf{btrue} & \triangleq & \lambda\,(x : \alpha)\,\lambda\,(y : \alpha)\,x \\ \mathsf{bfalse} & : & \alpha \to \alpha \to \alpha \\ \mathsf{bfalse} & \triangleq & \lambda\,(x : \alpha)\,\lambda\,(y : \alpha)\,y \\ \mathsf{pair} & : & \alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma \\ \mathsf{pair} & \triangleq & \lambda\,(x : \alpha)\,\lambda\,(y : \beta)\,\lambda\,(f : \alpha \to \beta \to \gamma)\,f\ x\ y \end{array}$$

*Hint:* The `Lam` and `App` constructors always first take two arguments: the domain type and the codomain type. Then, `Lam` takes a "higher-order function" corresponding to $\lambda$ in LF (and to \ in the code).

*Second Hint:* There are three "lambdas" in the code. The first is `Lam`. This says "construct a $\lambda$-function in $\boldsymbol{\lambda}_{\to}$". The second is \. This says "construct a term-level abstraction in LF". The final is `fn`. This is just a keyword in SML. It doesn't build any LF syntax trees!

*Third Hint:* Don't forget the correct uses of `Tm`. You shouldn't need `Ty`, or the base type.

*Fourth Hint*: The reference solution is 10 lines in total.