Lecture Notes on Parametric Polymorphism

15-814: Types and Programming Languages
Frank Pfenning and Jan Hoffmann

Lecture 10
Thursday, September 24, 2026

1 Introduction

Polymorphism refers to the possibility for an expression to have multiple types. System T does not feature polymorphism and as a result we have to, for example, implement the identity function for each argument type we want to use it with.

$$\begin{array}{lcl} id_1 & \triangleq & \lambda\,(x : \texttt{nat})\,x \\ id_2 & \triangleq & \lambda\,(x : \texttt{nat} \to \texttt{nat})\,x \end{array}$$

The function id_1 can be applied to arguments of type `nat` and the function id_2 can be applied to arguments of type `nat` $\to$ `nat`. However, the implementations of both functions are identical, that is, id_1 and id_2 have the same function body. This is why it is beneficial to just have one implementation of the identity function and assign it a polymorphic type

$$id : \forall(t.t \to t)$$

to express all possible argument types in a single form.

Christopher Strachey [Str00] distinguished two forms of polymorphism: *ad hoc polymorphism* and *parametric polymorphism*. Ad hoc polymorphism refers to multiple types possessed by a given expression or function which has different implementations for different types. For example, *plus* might have type $\mathit{int} \to \mathit{int} \to \mathit{int}$ but also $\mathit{float} \to \mathit{float} \to \mathit{float}$ with different implementations at these two types. Similarly, a function $\mathit{show} : \forall(t.\, t \to \mathit{string})$ might convert an argument of any type into a string, but the conversion function

itself will of course have to depend on the type of the argument: printing Booleans, integers, floating point numbers, pairs, etc. are all very different operations. Even though it is an important concept in programming languages, in this lecture we will not be concerned with ad hoc polymorphism.

In contrast, *parametric polymorphism*, which we introduce in this lecture, refers to a function that behaves the same at all possible types. The identity function, for example, is parametrically polymorphic because it just returns its argument, regardless of its type. The essence of "parametricity" wasn't rigorously captured until the beautiful analysis by John Reynolds [Rey83], which we will sketch in a later lecture on parametricity.

Slightly different systems for parametric polymorphism were discovered independently by Jean-Yves Girard [Gir71] and John Reynolds [Rey74]. Girard worked in the context of *logic* and developed *System F*, while Reynolds worked directly on *type systems* for programming languages and designed the polymorphic λ-calculus. With minor syntactic changes, we will follow Reynolds's presentation.

2 Universally Quantified Types

We would like to add types of the form $\forall(t.\,\tau)$ to express parametric polymorphism. The fundamental idea is that an expression of type $\forall(t.\,\tau)$ is a *function* that takes a *type* as an argument.

This is a rather radical change of attitude. So far, our expressions only contained type annotations to make types unique, and now types become embedded in expressions and are actually passed to functions. Let's see where it leads us. Now we *could* write

$$\lambda(t)\,\lambda(x:t).\,x : \forall(t.\,t \to t)$$

but abstraction over a type seems so different from abstraction over an expression that we make up a new notation and instead write

$$\Lambda(t)\,\lambda(x:t).\,x : \forall(t.\,t \to t)$$

using a capital lambda (Λ). To express the typing rules, our contexts carry two different forms of declarations: $x : \tau$ (as we had so far) and now also $t\ \mathit{type}$, expressing that t is a type variable. The typing judgment then has the form

$$\Delta\ \Gamma \vdash e : \tau$$

without repeated variables or type variables in Γ and Δ. There will be some further presuppositions mentioned later. For type abstractions, we have the rule

$$\frac{\Delta, t\ \mathit{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(t)\ e : \forall(t.\tau)}\ T_{\text{t-abs}}$$

Here, t is a bound variable in $\Lambda(t)\ e$ and $\forall(t.\tau)$ so we allow it to be silently renamed if it conflicts with any variable already declared in Γ or Δ. We implicitly use α-equivalence by requiring that we pick the same name t for both bound variables: the t in $\Lambda(t)\ e$ and the t in $\forall(t.\tau)$.

We haven't yet seen how t can actually appear in e, but we can already verify:

$$\dfrac{\dfrac{\dfrac{}{t\ \mathit{type}\ x : t \vdash x : t}\ T_{\text{var}}}{t\ \mathit{type}\ \cdot \vdash \lambda(x : t)\, x : t \to t}\ T_{\text{lam}}}{\cdot \vdash \Lambda(t)\, \lambda(x : t)\, x : \forall(t.\, t \to t)}\ T_{\text{t-abs}}$$

The next question is how we *apply* such a polymorphic function to a type? Again, we *could* just write $e(\tau)$ for the application of a polymorphic function e to a type τ, but we would like it to be more syntactically apparent so we write $e\,[\tau]$.

Assuming we have a type `nat` for natural numbers and numerals $\overline{n}$ in System F (we actually have that), we may want to apply our polymorphic identity to the numeral $\overline{8128}$. In this case, we want to apply the polymorphic identity to the type `nat` to get a term of type $\texttt{nat} \to \texttt{nat}$:

$$(\Lambda(t)\, \lambda(x : t)\, x)\,[\texttt{nat}] : \texttt{nat} \to \texttt{nat}$$

These considerations lead us to a rule where we *substitute into the type*:

$$\frac{\Delta\ \Gamma \vdash e : \forall(t.\tau) \quad \Delta \vdash \tau'\ \mathit{type}}{\Delta\ \Gamma \vdash e\,[\tau'] : [\tau'/t]\tau}\ T_{\text{t-app}}$$

The second premise is there to check that the type τ' which is part of the expression $e\ [\tau']$ is *valid*. This just means that all the type variables occurring in τ' are declared in Δ (just like all the expression variables in e must be declared in Γ).

Here is a small sample derivation, assuming we have defined

$$\begin{array}{lcl} \mathit{id} & : & \forall t.\, t \to t \\ \mathit{id} & = & \Lambda(t)\, \lambda(x : t)\, x \end{array}$$

Then we can typecheck:

$$\dfrac{\dfrac{\vdots \qquad \vdots}{\cdot \vdash \mathit{id} : \forall(t.\, t \to t) \quad \cdot \vdash \mathtt{nat}\ \mathit{type}}\, T_{\text{t-app}} \qquad \vdots}{\cdot \vdash \mathit{id}\,[\mathtt{nat}] : \mathtt{nat} \to \mathtt{nat} \qquad \cdot \vdash \overline{3} : \mathtt{nat}}\, T_{\text{app}}$$
$$\cdot \vdash \mathit{id}\,[\mathtt{nat}]\,\overline{3} : \mathtt{nat}$$

where we need some rules to verify that `nat` is a closed type (that is, has no free type variables). Fortunately, that's easy: we just check all the components of a type.

$$\dfrac{\Delta \vdash \tau_1\ \mathit{type} \quad \Delta \vdash \tau_2\ \mathit{type}}{\Delta \vdash \tau_1 \to \tau_2\ \mathit{type}}\, U_{\text{arr}} \qquad \dfrac{}{\Delta, t\ \mathit{type} \vdash t\ \mathit{type}}\, U_{\text{var}}$$

$$\dfrac{\Delta, t\ \mathit{type} \vdash \tau\ \mathit{type}}{\Delta \vdash \forall(t.\,\tau)\ \mathit{type}}\, U_{\text{all}}$$

3 Summary: Syntax and Typing Rules

Here is the summary of the language of the polymorphic λ-calculus:

			Abstract	Concrete	
Typ	τ	::=	t	t	type variable
			$\mathtt{fun}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$	universal type
Exp	e	::=	x	x	variable
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\,(x:\tau)\,e$	fun. abstraction
			$\mathtt{app}(e_1;e_2)$	$e_1(e_2)$	fun. application
			$\mathtt{Lam}(t.e)$	$\Lambda(t)\ e$	type abstraction
			$\mathtt{App}(e;\tau)$	$e[\tau]$	type application

Expression Contexts	Γ	::=	$\cdot \mid \Gamma, x:\tau$
Type Contexts	Δ	::=	$\cdot \mid \Delta, t\ \mathit{type}$

We assume that all variables and type variables in a context are distinct, and rename bound variables or type variables to maintain this invariant.

In a judgment $\Delta\ \Gamma \vdash e : \tau'$, we also assume that for all $x : \tau \in \Gamma$ $\Delta \vdash \tau\ \mathit{type}$. We maintain this property in the type rules, that is, we can prove the following lemma.

Lemma 1 *If* $\Delta\ \Gamma \vdash e : \tau'$ *and* $\Delta \vdash \tau$ *type for all* $x : \tau \in \Gamma$ *then* $\Delta \vdash \tau'$.

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau}\ T_{\text{var}} \qquad \frac{\Delta\ \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau\ \mathit{type}}{\Delta\ \Gamma \vdash \lambda(x : \tau)\, e : \tau \to \tau'}\ T_{\text{lam}}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta\ \Gamma \vdash e_2 : \tau}{\Delta\ \Gamma \vdash e_1(e_2) : \tau'}\ T_{\text{app}}$$

$$\frac{\Delta, t\ \mathit{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(t)\ e : \forall(t.\tau)}\ T_{\text{t-abs}} \qquad \frac{\Delta\ \Gamma \vdash e : \forall(t.\tau) \quad \Delta \vdash \tau'\ \mathit{type}}{\Delta\ \Gamma \vdash e\,[\tau'] : [\tau'/t]\tau}\ T_{\text{t-app}}$$

4 Dynamic Semantics

We now define the dynamic semantics of System F. Here, we present the eager (or call-by-value) version of the rules. Similarly as for System T, the choice of the evaluation order is inconsequential for System F since it is a total language.

Both function and type abstraction are values.

$$\frac{}{\lambda\,(x : \tau)\, e\ \texttt{val}}\ V_{\text{lam}} \qquad \frac{}{\Lambda(t)\ e\ \texttt{val}}\ V_{\text{t-abs}}$$

In the definition of the stepping relation, we have familiar rules for function application. For type instantiation, we substitute the argument type τ into the body of the type abstraction. Intuitively, this leads to a step that conforms to preservation since for instance $(\Lambda(t)\,\lambda(x : t)\,x)[\texttt{nat}] \longmapsto \lambda(x : \texttt{nat})\,x$.

$$\frac{e_1 \longmapsto e_1'}{e_1(e_2) \longmapsto e_1'(e_2)}\ E_{\text{ap1}} \qquad \frac{e_1\ \texttt{val} \quad e_2 \longmapsto e_2'}{e_1(e_2) \longmapsto e_1(e_2')}\ E_{\text{ap2}}$$

$$\frac{e_2\ \texttt{val}}{(\lambda\,(x : \tau)\, e)(e_2) \longmapsto [e_2/x]e}\ E_{\text{ap3}}$$

$$\frac{e \longmapsto e'}{e[\tau] \longmapsto e'[\tau]}\ E_{\text{t-ap1}} \qquad \frac{}{(\Lambda(t)\ e)[\tau] \longmapsto [\tau/t]e}\ E_{\text{t-ap2}}$$

We can prove type safety.

Theorem 2 (Progress) *If* $e : \tau$ *then either* e `val` *or there exists an* e' *such that* $e \longmapsto e'$.

Theorem 3 (Preservation) *If $e : \tau$ and $e \longmapsto e'$ then $e' : \tau$.*

One remarkable fact about the polymorphic λ-calculus (which is quite difficult to prove) is that every program still evaluates to a value.

Theorem 4 (Normalization) *If $e : \tau$ then $e \longmapsto^* v$ for a v such that v `val`.*

We can prove normalization with a similar *logical relations* (or *candidates*) method as for System T. However, the idea to go by induction on types breaks down when we come to the case of the type abstraction $\Lambda(t)\, e$. The natural idea would be to define that $\Lambda(t)\, e[\tau]$ terminates for all types τ. However, because of impredicativity, τ could for instance be the type of $\Lambda(t)\, e$, which means we won't be able to apply the induction hypothesis. The solution is to use a technique developed by John Reynolds that we will study in the context of equality for System F in a few weeks.

5 Typing Self-Application Polymorphically

As an exercise in building a typing derivation, we provide a polymorphic type for self-application $\lambda x.\, x\ x$. We accomplish this by allowing x to have a polymorphic type $\forall(t.\, t \to t)$. We call this type u (or $unit$) because there is exactly one normal term of this type: the polymorphic identity function. Applying the identity to itself seems plausible in any case. So we claim:

$$\begin{array}{lcl} u & \triangleq & \forall(t.\, t \to t) \\ \omega & : & u \to u \\ \omega & \triangleq & \lambda(x : u)\, x\, [u]\, x \end{array}$$

This is established by the following typing derivation. When you want to build such a derivation yourself, you should always build it "bottom-up", starting with the final conclusion. The fact that the rules are syntax-directed means you have no choice which rule to choose, but some parts of the type may be unknown and may need to be filled in later.

$$\dfrac{\dfrac{x : u \vdash x\ [u] : \boxed{} \qquad x : u \vdash x : \boxed{}}{x : u \vdash x\ [u]\ x : u}\, T_{\text{app}}}{\cdot \vdash \lambda(x : u)\, x\, [u]\, x : u \to u}\, T_{\text{lam}}$$

As a rule of thumb, it seems to work best to first fill in the first premise of an application (rule T_{app}) and then the second. Continuing in the left branch of the derivation (and remembering that $u = \forall(t.\, t \to t)$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : u \vdash x : u}\, T_{\text{var}} \quad \dfrac{t\ \mathit{type} \vdash t \to t\ \mathit{type}}{\cdot \vdash u\ \mathit{type}}\, U_{\text{all}}}{x : u \vdash x\ [u] : u \to u}\, T_{\text{t-app}} \qquad \dfrac{}{x : u \vdash x : u}\, T_{\text{var}}}{x : u \vdash x\ [u]\ x : u}\, T_{\text{app}}}{\cdot \vdash \lambda(x : u)\, x\ [u]\ x : u \to u}\, T_{\text{lam}}$$

The fact that $t \to t$ is a valid type follows quickly by the rules U_{arr} and U_{var}. There are more types that work for self-application (see Exercise 3).

Crucial in this example is that we can instantiate the quantifier in $u = \forall(t.\, t \to t)$ with u itself. This "self-referential" nature of the type quantifier is called *impredicativity* because it quantifies not only over types already defined, but also itself. Some systems of type theory reject impredicative quantification because the meaning of the quantified type is not constructed from the meaning of types we previously understand. Impredicativity was also seen as a source of paradoxes, although Girard did give a syntactic argument for the consistency of System F [Gir71] with impredicative quantification.

You may wonder why we cannot type a diverging expression like

$$\Omega \triangleq (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)\ .$$

In System F, we could start by replacing one of the two self-applications with the previously typed term $\omega = \lambda(x : u)\, x\ [u]\ x$. However, $\omega : u \to u$ and therefore we cannot derive a type for $\omega(\omega)$. The problem is that $u \neq u \to u$. In general, we can only pass a term with a smaller type (the argument type) in an application. Therefore, a "recursive call" has to be performed with an expression of smaller type, which prohibits infinite loops.

6 Church Encodings

The type system of System F is too restrictive to enable divergence but expressive enough for Church encodings.

Products We can define unit as follows.

$$\begin{array}{rcl} \texttt{unit} & \triangleq & \forall(t.t \to t) \\ \langle\rangle & \triangleq & \Lambda(t)\,\lambda\,(x:t)\,x \end{array}$$

The expression $\langle\rangle$ is not the only value of type `unit` but it is *essentially* the only value. More precisely, every value of type `unit` is an implementation of the identity function. We will prove that fact later in the course.

We define pairs as follows.

$$\begin{array}{rcl} \tau_1 \times \tau_2 & \triangleq & \forall(t.(\tau_1 \to \tau_2 \to t) \to t) \\ \langle e_1, e_2\rangle & \triangleq & \Lambda(t)\,\lambda\,(p:\tau_1 \to \tau_2 \to t)\,p(e_1)(e_2) \\ e \cdot i & \triangleq & e[\tau_i](\lambda\,(x_1:\tau_1)\,\lambda\,(x_2:\tau_2)\,x_i) \end{array}$$

Sums We define

$$\begin{array}{rcl} \texttt{void} & \triangleq & \forall(t.t) \\ \texttt{absurd}\{\rho\}(e) & \triangleq & e[\rho] \\ \tau_1 + \tau_2 & \triangleq & \forall(t.(\tau_1 \to t) \to (\tau_2 \to t) \to t) \\ i \cdot e & \triangleq & \Lambda(t)\,\lambda\,(x_1:\tau_1 \to t)\,\lambda\,(x_2:\tau_2 \to t)\,x_i(e) \\ \texttt{case}\ e\,\{\,x_1.e_1 \mid x_2.e_2\,\} & \triangleq & e[\tau](\lambda\,(x_1:\tau_1)\,e_1)(\lambda\,(x_2:\tau_2)\,e_2) \end{array}$$

In the last definition, we assume that $\texttt{case}\ e\,\{\,x_1.e_1 \mid x_2.e_2\,\}$ has type τ.

Natural Numbers Let's return to Church's representation of natural numbers. We define

$$\texttt{nat} \triangleq \forall(t.\,(t \to t) \to t \to t)$$

Then we can verify with typing derivations as above:

$$\begin{array}{rcl} \mathit{zero} & : & \texttt{nat} \\ \mathit{zero} & \triangleq & \Lambda(t)\,\lambda(s:t \to t)\,\lambda(z:t)\,z \end{array}$$

We also expect the successor function to have type $\texttt{nat} \to \texttt{nat}$, but there is one slightly tricky spot. We start:

$$\begin{array}{rcl} \mathit{succ} & : & \texttt{nat} \to \texttt{nat} \\ \mathit{succ} & \triangleq & \lambda n.\,\Lambda(t)\,\lambda(s:t \to t)\,\lambda(z:t)\,s\,(n\ \boxed{}) \end{array}$$

Before, we just applied n to s and z, but now $n : \texttt{nat}$, which means that it expects a *type* as its first argument. At this point (in a hypothetical typing derivation we did not write out), we have the context

$$t\ \mathit{type}\ \ n:\texttt{nat}, s:t \to t, z:t$$

so we need to instantiate the quantifier with t, which next requires arguments of type $t \to t$ and t (which we have at hand with s and z).

$$\begin{array}{lcl} \mathit{succ} & : & \mathtt{nat} \to \mathtt{nat} \\ \mathit{succ} & \triangleq & \lambda n.\, \Lambda(t)\, \lambda(s : t \to t)\, \lambda(z : t).\, s\, (n\, [t]\, s\, z) \end{array}$$

It becomes more interesting with the addition function. Recall that in the untyped setting we had

$$\mathit{plus} \triangleq \lambda n.\, \lambda k.\, n\ \mathit{succ}\ k$$

iterating the successor function n times on argument k. The start of the typed version is again relatively straightforward: the only difference is that we need to apply n first to a type.

$$\begin{array}{lcl} \mathit{plus} & : & \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \\ \mathit{plus} & \triangleq & \lambda n.\, \lambda k.\, n\ [\boxed{}]\ \mathit{succ}\ k \end{array}$$

But what type do we need? We have that the next argument has type $\mathtt{nat} \to \mathtt{nat}$ and the following one $\mathtt{nat}$, so that we need to instantiate t with $\mathtt{nat}$!

$$\begin{array}{lcl} \mathit{plus} & : & \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \\ \mathit{plus} & \triangleq & \lambda n.\, \lambda k.\, n\ [\mathtt{nat}]\ \mathit{succ}\ k \end{array}$$

So we need that

$$n : \forall(t.\, (t \to t) \to t \to t)$$

and then

$$n\ [\mathtt{nat}] : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat}$$

We should point out that this definition of addition cannot be typed in System T. In that setting, n can only be applied to functions s of one fixed type $\tau \to \tau$ to iterate starting from $z : \tau$. This means that very few functions are actually definable—essentially only functions like successor and addition, but not exponentiation or predecessor.

A significant aspect of this is that we instantiate the quantifier in $\mathtt{nat} = \forall(t.\, (t \to t) \to t \to t)$ with $\mathtt{nat}$ itself.

7 Implementation in Lambda

We can express Church encodings in the language LAMBDA, which uses polymorphic types when files have extension `.poly` or given the command

line argument `-l poly`. We use `!a` as concrete syntax for $\forall t$, and `/\a` for Λt. Type definitions are preceded by the keyword `type`, and type declarations for variable definitions are preceded by the keyword `decl`.

```
1  type nat = !a. (a -> a) -> a -> a
2
3  decl zero : nat
4  decl succ : nat -> nat
5
6  defn zero = /\a. \s. \z. z
7  defn succ = \n. /\a. \s. \z. s (n [a] s z)
8
9  decl plus : nat -> nat -> nat
10 defn plus = \n. \k. n [nat] succ k
11
12 decl times : nat -> nat -> nat
13 defn times = \n. \k. n [nat] (plus k) zero
14
15 norm _0 = zero
16 norm _1 = succ _0
17 norm _2 = succ _1
18 norm _3 = succ _2
19
20 norm _6 = times _2 _3
```

Listing 1: Polymorphic natural numbers in LAMBDA

So far, this straightforwardly follows the structure of the motivating examples. To represent the predecessor function, we require pairs of natural numbers. But what are their types? Recall:

$$\mathit{pair} \triangleq \lambda x.\, \lambda y.\, \lambda k.\, k\, x\, y$$

from which we conjecture

$$\mathit{pair} : \mathtt{nat} \to \mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat} \to \tau) \to \tau$$

where τ is arbitrary. So we realize that this function is *polymorphic* and we abstract over the result type of the continuation. We call the type of pairs of natural numbers *nat2*. In the type of the *pair* function it is then convenient to place the type abstraction *after* the two natural numbers have been received.

$$\begin{array}{lcl} \mathtt{nat2} & \triangleq & \forall (t.\, (\mathtt{nat} \to \mathtt{nat} \to t) \to t) \\ \\ \mathit{pair} & : & \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat2} \\ \mathit{pair} & \triangleq & \lambda (x : \mathtt{nat})\, \lambda (y : \mathtt{nat})\, \Lambda (t)\, \lambda k.\, k\, x\, y \end{array}$$

Now we can define *pred2*, with the specification that *pred2* $\overline{n}$ = *pair* $\overline{n}$ $\overline{n \dot{-} 1}$. We leave open the two places we have to provide a type.

$$\begin{array}{lcl} \mathit{pred2} & : & \mathtt{nat} \to \mathtt{nat2} \\ \mathit{pred2} & \triangleq & \lambda(n : \mathtt{nat}).\, n\, [\ \square\ ]\, (\lambda p.\, p\, [\ \square\ ]\, (\lambda(x : \mathtt{nat})\, \lambda y.\, \mathit{pair}\ (\mathit{succ}\ x)\ x))\ (\mathit{pair}\ \mathit{zero}\ \mathit{zero}) \end{array}$$

In the first box, we need to fill in the result type of the iteration (which is the type argument to $\overline{n}$), and this is *nat2*. In the second box we need to fill in the result type for the decomposition into a pair, and that is also *nat2*. Then, for the final definition of *pred* we only extract the second component of the pair, so the continuation only returns a natural number rather than a pair.

$$\begin{array}{lcl} \mathit{pred} & : & \mathtt{nat} \to \mathtt{nat} \\ \mathit{pred} & \triangleq & \lambda n.\, \mathit{pred2}(n)\ [\mathtt{nat}]\ (\lambda x.\, \lambda y.\, y) \end{array}$$

Below is a summary of this code in LAMBDA.

```
1  type nat2 = !a. (nat -> nat -> a) -> a
2
3  decl pair : nat -> nat -> nat2
4  defn pair = \x. \y. /\a. \k. k x y
5
6  decl pred2 : nat -> nat2
7  defn pred2 = \n. n [nat2] (\p. p [nat2] (\x. \y. pair (succ x) x))
8                            (pair zero zero)
9
10 decl pred : nat -> nat
11 defn pred = \n. pred2 n [nat] (\x. \y. y)
12
13 norm _6_5 = pred2 _6
14 norm _5 = pred _6
```

Listing 2: Predecessor on natural numbers in LAMBDA

Exercises

Exercise 1 Fill in the blanks in the following judgments so that it holds, or indicate there is no way to do so. You do not need to justify your answer or supply a typing derivation, and the types do not need to be "most general" in any sense. As always, feel free to use LAMBDA to check your answers.

(i) $\square \vdash \forall t.\, t \to \beta\ \mathit{type}$

(ii) $\boxed{} \vdash \Lambda(t)\, x\, [t \to t]\, y\, [\beta] : \forall t.\, \beta$

(iii) $\cdot \vdash \lambda(x :?)\, x\, [\boxed{}]\, x\, x : \boxed{}$

(iv) $t\ \mathit{type} \vdash \boxed{} : \forall \beta.\, t \to \beta$

(v) $x : \forall t.\, (\forall \beta.\, \beta \to \beta) \to t, \gamma\ \mathit{type} \vdash \boxed{} : (\gamma \to \gamma) \to \gamma$

Exercise 2 Prove that if $\Gamma \vdash e : \tau$ under the presupposition that $\Gamma\ \mathit{ctx}$ then $\Gamma \vdash \tau\ \mathit{type}$.

Exercise 3 We write F for a (mathematical) function from types to types (which is not expressible in the polymorphic λ-calculus but requires System F^ω). A more general family of types (one for each F) for self-application is given by

$$\begin{array}{lcl} u_F & = & \forall t.\, t \to F(t) \\ \omega_F & : & u_F \to F(u_F) \\ \omega_F & = & \lambda(x :?)\, x\, [u_F]\, x \end{array}$$

We recover the type from this lecture with $F = \Lambda(t)\, t$. You may want to verify the general typing derivation in preparation for the following questions, but you do not need to show it.

(i) Consider $F = \Lambda(t)\, t \to t$. In this case $u_F = \mathit{bool}$. Calculate the type and characterize the behavior of ω_F as a function on Booleans.

(ii) Consider $F = \Lambda(t)\, (t \to t) \to t$. Calculate u_F, the type of ω_F, and characterize the behavior of ω_F. Can you relate u_F and ω_F to the types and functions we have considered in the course so far?

References

[Gir71] Jean-Yves Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000. Notes for lecture course given at the International Summer School in Computer Programming at Copenhagen, Denmark, August 1967.