Functional Programming Lecture Notes - Part 2

Thomas Dinsdale-Young

8th September, 2017

3 Simply-typed Lambda Calculus

The lambda calculus we have seen so far has no typing discipline. A function could be applied to a boolean, a number, a pair, another function or indeed any other term, without regard for the intention of the function. Types provide a way of specifying properties of functions and enforcing correct usage.

The types of the simply-typed lambda calculus are defined as:

$$\sigma, \rho ::= \tau \mid \sigma \to \rho$$

where τ ranges over an infinite set of type variables. By convention, $\alpha \to \beta \to \gamma$ is interpreted as $\alpha \to (\beta \to \gamma)$. If we were to add additional types to the lambda calculus, such as booleans or numbers, we would add type constants for these. For now, however, we only have type variables and function types. The reading of the type $\sigma \to \rho$ is the type of functions from σ to ρ.

Terms of the simply-typed lambda calculus are as before, except that we attach a type to each lambda abstraction that indicates the source type of the function:

$$M, N ::= x \mid MN \mid \lambda x : \sigma.\, M$$

We want to define a judgement that associates a type with a lambda term. However, to do so, we need to know the types of the free variables in a term. A *typing context* Γ is a list of pairs $x : \sigma$ of variables and types. We assume that each variable occurs at most once in a typing context. (For now, the list might as well be a set — the order is not significant, and to simplify some considerations, we will sometimes treat it like a set. Later, however, we shall see more elaborate type systems where the order does become significant.)

The typing judgement $\Gamma \vdash M : \sigma$ assigns type σ to term M in typing context Γ. It is defined as follows:

$$\textsc{Start}\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \to \textsc{ elimination}\ \frac{\Gamma \vdash M : \sigma \to \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \rho} \qquad \to \textsc{ introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \rho}$$

Let's look at this judgement in action with a few examples. First, the identity function:

$$\to\!\text{I}\ \frac{\textsc{St}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x : \alpha.\, x : \alpha \to \alpha}$$

We can type the function differently if we give its argument a different type.

$$\rightarrow\text{I}\ \dfrac{\text{St}\ \dfrac{}{x : \alpha \to \beta \vdash x : \alpha \to \beta}}{\vdash \lambda x : \alpha \to \beta.\, x : (\alpha \to \beta) \to (\alpha \to \beta)}$$

$$\rightarrow\text{I}\ \dfrac{\text{St}\ \dfrac{}{x : (\alpha \to \beta) \to \gamma \vdash x : (\alpha \to \beta) \to \gamma}}{\vdash \lambda x : (\alpha \to \beta) \to \gamma.\, x : ((\alpha \to \beta) \to \gamma) \to (\alpha \to \beta) \to \gamma}$$

We can also type other familiar functions:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \beta \vdash x : \alpha}}{x : \alpha \vdash \lambda y : \beta.\, x : \beta \to \alpha}}{\vdash \lambda x : \alpha, y : \beta.\, x : \alpha \to \beta \to \alpha}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{St}\ \dfrac{}{\Gamma \vdash x : \alpha \to (\beta \to \gamma)} \quad \text{St}\ \dfrac{}{\Gamma \vdash z : \alpha}}{\Gamma \vdash xz : \beta \to \gamma} \quad \rightarrow\text{E}\ \dfrac{\text{St}\ \dfrac{}{\Gamma \vdash y : \alpha \to \beta} \quad \text{St}\ \dfrac{}{\Gamma \vdash z : \beta}}{\Gamma \vdash yz : \beta}}{x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha \vdash xz(yz) : \alpha \to \gamma}\ \rightarrow\text{E}}{x : \alpha \to \beta \to \gamma, y : \alpha \to \beta \vdash \lambda z : \alpha.\, xz(yz) : \alpha \to \gamma}\ \rightarrow\text{I}}{x : \alpha \to \beta \to \gamma \vdash \lambda y : \alpha \to \beta, z : \alpha.\, xz(yz) : (\alpha \to \beta) \to \alpha \to \gamma}\ \rightarrow\text{I}}{\vdash \lambda x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha.\, xz(yz) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma}\ \rightarrow\text{I}$$

where $\Gamma = [x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha]$.

Notice that the typing rules are entirely directed by the syntax of terms. That is to say, for any given term, exactly one of the typing rules can apply, depending on whether the term is a variable (START), application ($\to$ ELIMINATION) or abstraction ($\to$ INTRODUCTION). It is thus simple to decide whether $\Gamma \vdash M : \sigma$ holds — simply apply the typing rules until you either fail or build a correct derivation.

There are terms that cannot be typed. For example, consider the term xx. To type this term, the application rule requires that $x : \sigma \to \rho \in \Gamma$ and $x : \sigma \in \Gamma$ for some σ and ρ. We would thus require a type σ such that $\sigma = (\sigma \to \rho)$, which is just not possible.

A useful property of the typing judgement is that if a term is typeable, then it is also typeable in an extended context.

Lemma 1 (Weakening)**.** *If $\Gamma \vdash M : \sigma$ and Γ, Γ' is a well-defined typing context, then $\Gamma, \Gamma' \vdash M : \sigma$.*

Proof sketch. Simply replace Γ with Γ, Γ' in the derivation of $\Gamma \vdash M : \sigma$. The derivation will remain valid up to renaming of variables bound in M (which may clash with variables in Γ'). □

Note that since we consider terms up to α-equivalence, we can always assume that bound variables have different names from those in the context. For instance, $x : \alpha \vdash \lambda x : \beta.\, x : \beta \to \beta$ holds because $x : \alpha \vdash \lambda y : \beta.\, y : \beta \to \beta$ holds, and the terms are α-equivalent.

Conversely to the above, we can remove variables from the context that do not occur in the term without affecting the typing judgement.

Lemma 2. *If* $\Gamma, \Gamma' \vdash M : \sigma$ *and none of the variables in* Γ' *occur free in* M*, then* $\Gamma \vdash M : \sigma$*.*

Proof sketch. Simply replace Γ, Γ' with Γ in the initial derivation. The derivation will remain valid, since none of the axioms will refer to variables in Γ'. □

An important property is that we can substitute a term for a free variable of the same type in a judgement.

Lemma 3. *Suppose that* $\Gamma, x : \sigma \vdash M : \rho$ *and* $\Gamma \vdash N : \sigma$*. Then* $\Gamma \vdash M[N/x] : \rho$*.*

Proof sketch. The derivation of $\Gamma \vdash M[N/x] : \rho$ is obtained by replacing all occurrences of

$$\textsc{St}\ \frac{}{\Gamma, x : \sigma, \Gamma' \vdash x : \sigma}$$

in the derivation of $\Gamma, x : \sigma \vdash M : \sigma$ by derivations of $\Gamma, \Gamma' \vdash N : \sigma$ (obtained by weakening from $\Gamma \vdash N : \sigma$). □

It follows from this that β-reduction preserves typing.

Theorem 1. *If* $\Gamma \vdash M : \sigma$ *and* $M \to_\beta N$*, then* $\Gamma \vdash N : \sigma$*.*

This result is important to our intuition about what types represent: if a term has a given type then any result it computes to will also have that type. For instance, if we have a function from numbers to booleans, and we call it with a number, then we would expect the result of the evaluation to be a boolean. This property of a type system is known as *preservation.*

Another important result is that all typed terms are normalising; indeed, they are strongly normalising.

Theorem 2. *If* $\Gamma \vdash M : \sigma$ *then* M *is (strongly) normalising with respect to* β*-reduction.*

This means that terms which do not terminate cannot be typed in our system. In particular, we might like to type a fixed-point combinator as $(\sigma \to \sigma) \to \sigma$, yet we cannot, since applying it to $\lambda x : \sigma.\, \sigma$ will yield a non-terminating computation. It is possible to add a typing rule for recursion, which is common in general purpose functional programming languages, such as the following:

$$\frac{\Gamma, x : \rho \vdash M : \rho \qquad \Gamma, x : \rho \vdash N : \sigma}{\Gamma \vdash \mathsf{letrec}\ x : \rho = M\ \mathsf{in}\ N : \sigma}$$

(The evaluation rule for letrec reduces $\mathsf{letrec}\ x : \rho = M\ \mathsf{in}\ N$ to $N[(\mathsf{letrec}\ x : \rho = M\ \mathsf{in}\ M)/x]$.)

3.1 A Connection to Logic

The typing judgement $\Gamma \vdash M : \sigma$ may be very reminiscent of a proof judgement $\Gamma \vdash \phi$ in formal logic, meaning that statement ϕ is provable from the assumptions Γ. The core proof rule of most logical systems is *modus ponens*: if "ϕ implies ψ" holds and "ϕ" holds, then "ψ" holds. Formulated as a proof rule:

$$\frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

This looks familiar. In fact, it is the same as the $\to$ ELIMINATION rule from our type system, except without the terms.

One way of constructing a proof system is to take modus ponens together with a system of axioms (called a Hibert-style system).[1] One such system is the following:

$$\frac{}{\Gamma \vdash A \to A} \qquad \frac{}{\Gamma \vdash A \to (B \to A)}$$

$$\frac{}{\Gamma \vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))}$$

These axioms should also look familiar: they correspond to the types of the **I**, **K** and **S** combinators! These axioms generate the implicational fragment of intuitionistic logic (called the positive implicational calculus). Intuitionistic logic is the standard propositional logic without the law of the excluded middle (the axiom $A \vee \neg A$), and the implicational fragment consists of the formulae that are expressible only using implication.

Another approach is to have a rule for introducing implications, and a rule for using assumptions:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \qquad \frac{}{\Gamma, \phi \vdash \phi}$$

Again, these should look familiar as the $\to$ INTRODUCTION and START rules of the simply-typed lambda calculus, but without the terms.

There is a very close correspondence between logic and type theory, which even goes beyond what we have seen so far. This correspondence is called the Curry-Howard isomorphism. The slogan is that propositions are types, and a proof of a proposition is a program of that type.

For example, consider a proof of the proposition $x > 6 \to x > 2$. One way of thinking of such a proof is as a *function* that transforms a proof that $x > 6$ into a proof that $x > 2$.

In the proof rules for logic, we did not have a term as we did for the lambda calculus. However, we can reasonably think of typed lambda terms as being proofs, since from them we can deterministically reconstruct the derivation tree. (The proof rules are directed by the syntax of the term.)

In logic, we are typically only concerned with *whether* a proof exists, and do not care what the proof is per se. Consequently, the proof judgement does not include the proof term. By contrast, with programs we typically wish to be able to run the program to obtain a result. Consequently, we really do care what the program is.

Another contrast is that we may not care about whether a program terminates, and therefore be willing to admit the letrec rule to our type system. However, the letrec rule allows us to build a term of any type we like ($\vdash \mathsf{letrec}\ x : \sigma = x\ \mathsf{in}\ x : \sigma$). In a logic, however, we do not want every proposition to be provable, so we could not admit such a rule: all proofs must be terminating.

[1] For such systems, the context Γ is actually redundant.

3.2 Curry and Church-style Type Systems

The approach we have taken to the simply-typed lambda calculus has been to associate a type with every abstraction; this is the approach taken by Church. Another approach, taken by Curry, is not to associate types with abstractions. In this case, the typing rule for λ-abstraction becomes:

$$\frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x.\, M) : \sigma \to \rho} \to \text{ INTRODUCTION}$$

In such a system, the type of a term is not uniquely determined by the term itself. However, any typeable term does have a *principal type* — a type from which all other valid types can be obtained by substituting type variables. For instance, $\lambda x.\, x$ has principal type $\alpha \to \alpha$. It also has type $(\beta \to \gamma) \to (\beta \to \gamma)$, which is an instance of the principal type obtained by substituting $(\beta \to \gamma)$ for α. Finding principal types can be done by *type inference*, which we will not cover here.

3.3 Exercises

Exercise 10. Define the type of natural numbers $\mathbf{nat} = \sigma \to (\sigma \to \sigma) \to \sigma$. Check that the Church numerals can be typed at this type. Recalling the arithmetic operations from Exercise 8, which of these can be typed as operators on **nat**? Which cannot, and why?

Exercise 11. Suppose that we add pairs to the λ-calculus. That is, we add three new term formers to the language: $\langle M, N \rangle$ (the pair of M and N), $\pi_1 M$ (the first projection of M), and $\pi_2 M$ (the second projection of M). We extend β-reduction with rules for projecting pairs:

$$\frac{}{\pi_1 \langle M, N \rangle \to_\beta M} \qquad \frac{}{\pi_2 \langle M, N \rangle \to_\beta N}$$

Extend the types with the new type former for pairs: $\sigma \times \tau$.

Define three new typing rules, one for each new term former, such that the new β-reduction rules preserve types.

Exercise 12. Besides α- and β-conversion, there is a third common conversion rule for λ-calculus: η-conversion. For the simply-typed lambda calculus, η-reduction is given by:

$$\frac{x \notin \mathrm{FV}(M)}{\lambda x : \sigma.\, M x \to_\eta M}$$

An intuitive way of viewing this is as an expression of *functional extensionality:* the notion that two functions are equal if they map the same inputs to the same outputs (i.e. if $f(x) = g(x)$ for all x, then $f = g$). In particular, we have $(\lambda x : \sigma.\, M x) N \to_\beta M N$.

Show that η-reduction preserves types. That is, if $\Gamma \vdash \lambda x : \sigma.\, M x : \rho$ and $x \notin \mathrm{FV}(M)$ then $\Gamma \vdash M : \rho$. [Hint: ρ is not a base type. How does it decompose?]

4 System F

One problem with the simply-typed lambda calculus is that the Church encoding techniques we used to represent datastructures in the untyped lambda calculus are less flexible in the simply-typed setting. For instance, we could represent the booleans by the type $\mathbf{bool} = \sigma \to \sigma \to \sigma$: $\mathsf{true} = \lambda x : \sigma, y : \sigma.\, x$ and $\mathsf{false} = \lambda x : \sigma, y : \sigma.\, y$ both inhabit this type. We would like to define the conditional operator $\mathsf{if} : \mathbf{bool} \to \tau \to \tau \to \tau$ that returns the second argument if the first is true and the third if it is false. However, our attempt to define such an operator is doomed to failure:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash b : \tau \to (\tau \to \tau) \qquad \overline{\Gamma \vdash x : \tau}}{\Gamma \vdash bx : \tau \to \tau} \qquad \overline{\Gamma \vdash y : \tau}}{\Gamma \vdash (bx)y : \tau}}{\vdash \lambda b : \mathbf{bool}, x : \tau, y : \tau.\, bxy : \mathbf{bool} \to \tau \to \tau \to \tau}$$

where $\Gamma = [b : \sigma \to \sigma \to \sigma, x : \tau, y : \tau]$. The judgement $\Gamma \vdash b : \tau \to (\tau \to \tau)$ simply does not hold. We cannot call b with x, since x has type τ and b expects something of type σ.

Similar issues prevent us from defining the predecessor function on Church numerals (per Exercise 10) and from representing pairs with $\sigma \times \rho = (\sigma \to \rho \to \alpha) \to \alpha$. Of course, one solution is to simply extend the language with these (and any other) datatypes (as in Exercise 11).

Alternatively, we can observe that the problem with how we are representing these datatypes is that they are too specific: our true has type $\alpha \to \alpha \to \alpha$ for *some specific* α, when we would like it to have that type *for every choice of* α. What would permit us to do this is to *parametrise* true (and other terms) by the type α, effectively abstracting over the type. To do this, we introduce a new notation for type abstraction: $\Lambda\alpha.\, M$. This allows us to define a *polymorphic* version of true:

$$\Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x$$

Just as we use a λ-abstraction by applying it to a term, we use a Λ-abstraction by applying it to a type. The terms of this new calculus are thus:

$$M, N ::= x \mid MN \mid \lambda x : \sigma.\, M \mid M\sigma \mid \Lambda\alpha.\, M$$

We correspondingly extend the β-reduction rules with:

$$\overline{(\Lambda\alpha.\, M)\sigma \to_\beta M[\sigma/\alpha]}$$

We also need a type constructor to correspond to Λ-abstraction. If term M has type σ then we can think of term $\Lambda\alpha.\, M$ as having "type σ for all choices of α" (noting that α may well occur in the type σ). The type constructor is thus $\forall\alpha.\, \sigma$ (read as "for all α, σ"). The types are then:

$$\sigma, \rho ::= \alpha \mid \sigma \to \rho \mid \forall\alpha.\, \sigma$$

Note that $\forall\alpha.\, \sigma$ binds occurrences of the type variable α in the type σ; similarly, $\Lambda\alpha.\, M$ binds occurrences of the type variable α in the term M. This has

implications for α-equivalence: we consider terms to be identical if they differ only in the names of their bound type (e.g. α) and term (e.g. x) variables. It also has implications for substitution: we do not substitute bound occurrences of a variable, and we avoid variable capture under substitution.

Now we get to the typing judgement.

$$\textsc{Start}\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \to\ \textsc{elimination}\ \frac{\Gamma \vdash M : \sigma \to \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \rho} \qquad \to\ \textsc{introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \rho}$$

$$\forall\ \textsc{elimination}\ \frac{\Gamma \vdash M : \forall \alpha.\, \sigma}{\Gamma \vdash M\rho : \sigma[\rho/\alpha]} \qquad \forall\ \textsc{introduction}\ \frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \Lambda\alpha.\, M : \forall \alpha.\, \sigma}$$

Note that the $\forall$ INTRODUCTION rule has the condition $\alpha \notin \mathrm{FV}(\Gamma)$: that is, α does not occur in the free variables of Γ. This condition is essential, since otherwise we could construct such derivations as

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash \Lambda\alpha.\, x : \forall \alpha.\, \alpha}}{x : \alpha \vdash (\Lambda\alpha.\, x)\sigma : \sigma}$$

Considering that β-reduction should preserve typing, we would thus expect to have $x : \alpha \vdash x : \sigma$, which does not make sense. The problem is that we have captured the variable α, which *should* refer to the type of x in the context.

This calculus is called System F, or second-order lambda calculus. It was discovered independently by Jean-Yves Girard and John C. Reynolds in the 1970s. As with the simply-typed lambda calculus, well-typed terms are strongly normalising.

4.1 Encoding Datatypes

Recall that the Church encoding of datatypes in the untyped setting was to represent a value as a function that takes in interpretations of the constructors and produces the corresponding value in terms of them. In System F, we can represent such values as also being parametrised by the *type* to interpret the value in.

For example, with the natural numbers

$$n ::= \mathsf{0} \mid \mathsf{S}n$$

we can represent numbers as

$$\mathsf{0} = \Lambda\nu.\, \lambda o : \nu, s : \nu \to \nu.\, o \qquad \mathsf{1} = \Lambda\nu.\, \lambda o : \nu, s : \nu \to \nu.\, so$$

$$\mathsf{2} = \Lambda\nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(so) \qquad \mathsf{3} = \Lambda\nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(s(so))$$

$$\mathsf{4} = \Lambda\nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(s(s(so)))$$

each of which has type $\forall \nu.\, \nu \to (\nu \to \nu) \to \nu$.

Note that the type of natural numbers allows us to define functions by recursion on natural numbers. A definition by recursion defines the function f explicitly at $f(\mathsf{0})$, and defines $f(\mathsf{S}\ n)$ in terms of $f(n)$. That is, $f : \mathbf{nat} \to \sigma$ is given

by some $base : \sigma$ and $rec : \sigma \to \sigma$. We can thus define $f = \lambda n : \mathbf{nat}.\, n\, \sigma\, base\, rec$ which has type $\mathbf{nat} \to \sigma$ given $\mathbf{nat} = \forall \nu.\, \nu \to (\nu \to \nu) \to \nu$.

We can also represent datatypes that are defined in terms of other types. For example, the type of pairs of values of types σ and ρ: $\sigma \times \rho$. An implementation of such pairs has one constructor that takes a σ and a ρ to give a pair. We can therefore interpret $\sigma \times \rho$ as $\forall \alpha.\, (\sigma \to \rho \to \alpha) \to \alpha$. The pair of M and N (having types σ and τ respectively) is thus represented as $\Lambda \alpha.\, \lambda p : \sigma \to \tau \to \alpha.\, pMN$.

As another example, consider lists of values of type σ: $\mathbf{list}\ \sigma$. Lists have two constructors:

$$\mathsf{nil} : \mathbf{list}\ \sigma$$
$$\mathsf{cons} : \sigma \to \mathbf{list}\ \sigma \to \mathbf{list}\ \sigma$$

We can therefore interpret the type as

$$\mathbf{list}\ \sigma = \forall \alpha.\, \alpha \to (\sigma \to \alpha \to \alpha) \to \alpha$$

4.2 Logic

We have already seen a correspondence between propositional logic and the simply-typed lambda calculus. One way to extend this is to represent additional propositional connectives ($\wedge, \vee, \bot, \dots$) as type constructions. To do so, let us consider the axioms for each connective, and choose types that allow us to satisfy these axioms.

First, consider $\bot$. The one axiom[2] for $\bot$ is elimination:

$$\bot \to \phi$$

That is to say, everything follows from $\bot$. An appropriate choice for $\bot$ is the type $\forall \alpha.\, \alpha$, since we have

$$\to\mathrm{I}\ \dfrac{\forall\mathrm{E}\ \dfrac{\mathrm{St}\ \dfrac{}{f : (\forall \alpha.\, \alpha) \vdash f : (\forall \alpha.\, \alpha)}}{f : (\forall \alpha.\, \alpha) \vdash f\psi : \psi}}{\vdash \lambda f : (\forall \alpha.\, \alpha).\, f\psi : (\forall \alpha.\, \alpha) \to \psi}$$

Second, consider $\phi \vee \psi$. Disjunction has two introduction axioms:

$$\phi \to (\phi \vee \psi) \qquad\qquad \psi \to (\phi \vee \psi)$$

From either ϕ or ψ we can get $\phi \vee \psi$. Disjunction also has one elimination axiom:

$$(\phi \to \theta) \to (\psi \to \theta) \to (\phi \vee \psi) \to \theta$$

If we can obtain θ from ϕ and also from ψ, then we can obtain it from $\phi \vee \psi$. The elimination rule can give us an idea of how to interpret $\phi \vee \psi$: for any θ, we want to obtain θ given that $\phi \to \theta$ and $\psi \to \theta$. So we can choose $\phi \vee \psi = \forall \theta.\, (\phi \to \theta) \to (\psi \to \theta) \to \theta$. We then have:

$$\vdash \lambda f : (\phi \to \theta), g : (\psi \to \theta), h : (\phi \vee \psi).\, h\theta fg : (\phi \to \theta) \to (\psi \to \theta) \to (\phi \vee \psi) \to \theta$$
$$\vdash \lambda a : \phi.\, \Lambda \theta.\, \lambda f : (\phi \to \theta), g : (\psi \to \theta).\, fa : \phi \to (\phi \vee \psi)$$
$$\vdash \lambda g : \psi.\, \Lambda \theta.\, \lambda f : (\phi \to \theta), g : (\psi \to \theta).\, gb : \psi \to (\phi \vee \psi)$$

[2] Technically, this is an *axiom schema* — every instance obtained by substituting the free variables is an axiom.

4.3 Exercises

Exercise 13. If the side condition $\alpha \notin \mathrm{FV}(\Gamma)$ is dropped from the $\forall$ INTRODUCTION rule, show that there is a closed term (i.e. having no free variables) of type σ for arbitrary σ.

Exercise 14. For lists, define functions

$$\mathsf{nil} : \forall\beta.\, \mathbf{list}\ \beta$$
$$\mathsf{cons} : \forall\beta.\, \beta \to \mathbf{list}\ \beta \to \mathbf{list}\ \beta$$

Such that $\mathsf{nil}\ \sigma$ produces the empty list and $\mathsf{cons}\ \sigma\ x\ l$ constructs the list with head x and tail l. Check that these function satisfy their types. Check that

$$\mathsf{cons}\ \mathbf{nat}\ 1\ (\mathsf{cons}\ \mathbf{nat}\ 2\ (\mathsf{nil}\ \mathbf{nat})) =_\beta \Lambda\alpha.\, \lambda n : \alpha, c : (\mathbf{nat} \to \alpha \to \alpha).\, c1(c2n)$$

Define the function

$$\mathsf{concat} : \forall\beta.\, \mathbf{list}\ \beta \to \mathbf{list}\ \beta \to \mathbf{list}\ \beta$$

that concatenates two lists. Check that

$$\mathsf{concat}\ \mathbf{nat}\ (\Lambda\alpha.\, \lambda n : \alpha, c : (\mathbf{nat} \to \alpha \to \alpha).\, c1n)\ (\Lambda\alpha.\, \lambda n : \alpha, c : (\mathbf{nat} \to \alpha \to \alpha).\, c2n)$$
$$=_\beta \Lambda\alpha.\, \lambda n : \alpha, c : (\mathbf{nat} \to \alpha \to \alpha).\, c1(c2n)$$

Exercise 15. Define the function

$$\mathsf{reverse} : \forall\beta.\, \mathbf{list}\ \beta \to \mathbf{list}\ \beta$$

that reverses a list. To do so, it may be helpful to first define a function

$$\mathsf{reverseHelp} : \forall\beta.\, \mathbf{list}\ \beta \to \mathbf{list}\ \beta \to \mathbf{list}\ \beta$$

such that $\mathsf{reverseHelp}\ \beta\ a\ b = \mathsf{concat}\ \beta\ (\mathsf{reverse}\ \beta\ a)\ b$. Check that your function is well-typed, and that it computes the expected output on some appropriate input.

Exercise 16. Binary trees $\mathbf{tree}\ \sigma$ over a type σ are defined by two constructors:

$$\mathsf{leaf} : \sigma \to \mathbf{tree}\ \sigma$$
$$\mathsf{branch} : \mathbf{tree}\ \sigma \to \mathbf{tree}\ \sigma \to \mathbf{tree}\ \sigma$$

Define the type of trees as a Church encoding. Define a function $\mathsf{leaves} : \forall\beta.\, \mathbf{tree}\ \beta \to \mathbf{list}\ \beta$ that collapses a tree to a list of leaves (in left-to-right order). You may assume the concat operator on lists.

Exercise 17. The introduction axiom for conjunction is

$$\phi \to \psi \to \phi \wedge \psi$$

and the elimination axioms are

$$\phi \wedge \psi \to \phi$$
$$\phi \wedge \psi \to \psi$$

Define an interpretation of $\phi \wedge \psi$ as a type, and give terms that interpret the axioms. Is this familiar?