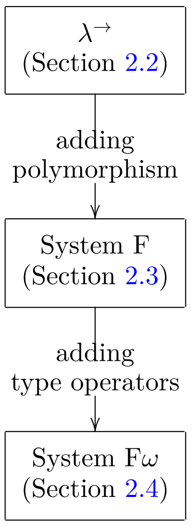
# Chapter 2

# Lambda calculus

The *lambda calculus* serves as the basis of most functional programming languages. More accurately, we might say that functional programming languages are based on the *lambda calculi* (plural), since there are many variants of lambda calculus. In this chapter we'll introduce three of these variants, starting with the simply typed lambda calculus (Section 2.2), moving on to System F, the polymorphic lambda calculus (Section 2.3), and concluding with System F$\omega$, a variant of System F with type operators (Section 2.4).



$\lambda^{\rightarrow}$ (Section 2.2)
adding polymorphism
System F (Section 2.3)
adding type operators
System F$\omega$ (Section 2.4)

Figure 2.1: Chapter plan

None of the calculi in this chapter are particularly suitable for programming. The simpler systems are insufficiently expressive — for example, they have no facilities for defining data types. In contrast, System F$\omega$, the last system that we'll look at, has excellent abstraction facilities but is rather too verbose for writing programs. In OCaml, the language we'll use for the rest of the course, we might define the function which composes two functions as follows:

```
fun f g x -> f (g x)
```

We can define the same function in System F$\omega$, but the simple logic is rather lost in a mass of annotations:

$$
\begin{array}{l}
\Lambda\alpha :: * . \\
\quad \Lambda\beta :: * . \\
\quad\quad \Lambda\gamma :: * . \\
\quad\quad\quad \lambda f : \alpha \to \beta . \\
\quad\quad\quad\quad \lambda g : \gamma \to \alpha . \\
\quad\quad\quad\quad\quad \lambda x : \gamma . \; f \; (g \; x)
\end{array}
$$

If these systems are unsuitable for programming, why should we study them at all? One reason is that many — perhaps most — of the features available in modern functional programming languages have straightforward translations into System F$\omega$ or other powerful variants of the lambda calculus[1]. The calculi in this chapter will give us a simple and uniform framework for understanding many language features and programming patterns in the rest of the course which might otherwise seem complex and arbitrary.

The foundational nature of System F and System F$\omega$ means that we'll see them in a variety of roles during the course, including:

- the elaboration language for type inference in Chapter 3.
- the proof system for reasoning with propositional logic in Chapter 4.
- the setting for dualities in Chapter 5
- the background for parametricity properties in Chapter 6
- the language underlying and motivating higher-order polymorphism in languages such as OCaml in Chapter 6
- the elaboration language for modules in Chapter 6
- the core calculus for GADTs in Chapter 7

## 2.1 Typing rules

We'll present the various languages in this chapter using *inference rules*, which take the following form:

$$\frac{\text{premise 1} \qquad \text{premise 1} \qquad \text{...} \qquad \text{premise N}}{\text{conclusion}}\ \text{rule name}$$

This is read as follows: from proofs of *premise 1*, *premise 2*, *...premise n* we may obtain a proof of *conclusion* using the rule *rule name*. Here is an example rule, representing one of the twenty-four Aristotelian syllogisms:

$$\frac{\text{all } M \text{ are } P \qquad \text{all } S \text{ are } M}{\text{all } S \text{ are } P}\ \text{modus barbara}$$

The upper-case letters $M$, $P$, and $S$ in this rule are *meta-variables*: we may replace them with valid terms to obtain an *instance* of the rule. For example, we might instantiate the rule to

[1] Some implementations of functional languages, such as the Glasgow Haskell Compiler, use a variant of System F$\omega$ as an internal language into which source programs are translated as an intermediate step during compilation to machine code. OCaml doesn't use this strategy, but understanding how we might translate OCaml programs to System F$\omega$ still gives us a useful conceptual model.

$$\frac{\text{all programs are buggy} \qquad \text{all functional programs are programs}}{\text{all functional programs are buggy}}\text{modus barbara}$$

The rules that we will see in this chapter have some additional structure: each statement, whether a premise or a conclusion, typically involves a *context*, a *term* and a *classification*. For example, here is the rule →-*elim*[2]:

$$\frac{\begin{array}{c}\Gamma \vdash M : A \to B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B}\to\text{-elim}$$

Both the premises and the conclusion take the form $\Gamma \vdash M : A$, which we can read "In the context $\Gamma$, $M$ has type $A$." It is important to note that each occurrence of $\Gamma$ refers to the same context (and similarly for $M$, $N$, $A$, and $B$): the →-*elim* rule says that if the term $M$ has type $A \to B$ in a context $\Gamma$, and the term $N$ has type $A$ in the same context $\Gamma$, then the term $M\ N$ has type $B$ in the same context $\Gamma$. As before, $\Gamma$, $M$, etc. are metavariables which we can instantiate with particular terms and contexts to obtain facts about particular programs.

## 2.2 Simply typed lambda calculus

We'll start by looking at a minimal language. The simply typed lambda calculus lies at the core of typed functional languages such as OCaml. Every typed lambda calculus program is (after a few straightforward syntactic changes) a valid program in OCaml, and every non-trivial OCaml program is built from the constructs of the typed lambda calculus along with some "extra stuff" — polymorphism, data types, modules, and so on — which we will cover in later chapters.

The name "simply typed lambda calculus" is rather unwieldy, so we'll use the traditional and shorter name $\lambda^{\to}$. The arrow → indicates the centrality of function types $A \to B$.

Let's start by looking at some simple $\lambda^{\to}$ programs.

- The simplest complete program is the **identity function** which simply returns its argument. Since $\lambda^{\to}$ is not polymorphic we need a separate identity function for each type. At a given type $A$ the identity function is written as follows:

  $\lambda\mathrm{x}:\mathrm{A}.\,\mathrm{x}$

  In OCaml we write

  ```
  fun x -> x
  ```

  or, if we'd like to be explicit about the type of the argument,

[2] We will often arrange premises vertically rather than horizontally.

```
fun (x:a) -> x
```

- The **compose function** corresponding to the mathematical composition of two functions is written

  $$\lambda f : B \to C.\lambda g : A \to B.\lambda x : A.f\ (g\ x)$$

  for types $A$, $B$ and $C$. In OCaml we write

  ```
  fun f g x -> f (g x)
  ```

Although simple, these examples illustrate all the elements of $\lambda^{\to}$: types, variables, abstractions, applications and parenthesization. We now turn to a more formal definition of the calculus. For uniformity we will present both the grammar of the language and the type rules[3] as inference rules.

The elements of the lambda calculi described here are divided into three "sorts":

- **terms**, such as the function application f p. We use the letters $L$, $M$ and $N$ (sometimes subscripted: $L_1$, $L_2$, etc.) as metavariables that range over terms, so that we can write statements about arbitrary terms: "For any terms $M$ and $N$ ...".
- **types**, such as the function type $\mathcal{B} \to \mathcal{B}$. The metavariables $A$ and $B$ range over expressions that form types. We write $M : A$ to say that the term $M$ has the type $A$.
- **kinds**, which you can think of as the types of type expressions. The metavariable $K$ ranges over kinds. We write $T :: K$ to say that the type expression $T$ has the kind $K$.

**Kinds in $\lambda^{\to}$** Rules that introduce kinds take the following form:

$$K \text{ is a kind}$$

Kinds play little part in $\lambda^{\to}$, so their structure is trivial. The later calculi will enrich the kind structure. For now there is a single kind, called $*$.

$$\frac{}{* \text{ is a kind}}\ *\text{-kind}$$

**Kinding rules in $\lambda^{\to}$** The set of types is defined inductively using rules of the form

$$\Gamma \vdash A :: K$$

which you can read as "type $A$ has kind $K$ in environment $\Gamma$. We will have more to say about environments shortly. In $\lambda^{\to}$ there are two kinding rules, which describe how to form types:

[3] Here and throughout this chapter we will focus almost exclusively on the grammar and typing rules of the language — the so-called static semantics — and treat the dynamic semantics (evaluation rules) as "obvious". There are lots of texts that cover the details of evaluation, if you're interested; Pierce's book is a good place to start.

$$\frac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B} \qquad\qquad \frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A \to B :: *}\,\text{kind-}\!\to$$

The kind-$\mathcal{B}$ rule introduces a base type $\mathcal{B}$ of kind $*$. Base types correspond to primitive types available in most programming languages, such as the types int, float, etc. in OCaml. They will not play a very significant part in the development, but without them we would have no way of constructing types in $\lambda^{\to}$.

The kind-$\to$ rule gives us a way of forming function types $A \to B$ from types $A$ and $B$. The arrow $\to$ associates rightwards: $A \to B \to C$ means $A \to (B \to C)$, not $(A \to B) \to C$. We'll use parentheses in the obvious way when we need something other than the default associativity, but we won't bother to include the (entirely straightforward) rules showing where parentheses can occur.

Using the rules kind-$\mathcal{B}$ and kind-$\to$ we can form a variety of function types. For example,

- $\mathcal{B}$, the base type.
- $\mathcal{B} \to \mathcal{B}$, the type of functions from $\mathcal{B}$ to $\mathcal{B}$.
- $\mathcal{B} \to (\mathcal{B} \to \mathcal{B})$, the type of functions from $\mathcal{B}$ to functions from $\mathcal{B}$ to $\mathcal{B}$. The expression $\mathcal{B} \to \mathcal{B} \to \mathcal{B}$ has the same meaning.
- $(\mathcal{B} \to \mathcal{B}) \to \mathcal{B}$, the type of functions from functions from $\mathcal{B}$ to $\mathcal{B}$ to $\mathcal{B}$.

Since the syntax of types is described using logical rules we can give formal derivations for particular types. For example, the type $(\mathcal{B} \to \mathcal{B}) \to \mathcal{B}$ can be derived as follows:

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B} \qquad \dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B}}{\Gamma \vdash \mathcal{B} \to \mathcal{B} :: *}\,\text{kind-}\!\to \qquad \dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B}}{\Gamma \vdash (\mathcal{B} \to \mathcal{B}) \to \mathcal{B} :: *}\,\text{kind-}\!\to$$

**Environment rules in $\lambda^{\to}$** Environments associate variables with their classifiers — i.e. term variables with types and type variables with kinds. Environments have no role to play in the rules kind-$\mathcal{B}$ and kind-$\to$, but will be needed for the additional kinding rules introduced in later sections, when we extend the type language with variables. In $\lambda^{\to}$ environments associate (term) variables with types.

Rules for forming environments have the form

$$\Gamma \text{ is an environment}$$

In $\lambda^{\to}$ there are two rules for forming environments:

$$\frac{}{\cdot \text{ is an environment}}\ \Gamma\text{-}\cdot \qquad \frac{\Gamma \text{ is an environment} \qquad \Gamma \vdash A :: *}{\Gamma, x{:}A \text{ is an environment}}\ \Gamma\text{-}:$$

The rule Γ-· introduces an empty environment. The rule Γ-: extends an existing environment Γ with a binding $x : A$ – that is, it associates the variable $x$ with the type $A$. (We use the letters $x$, $y$, $z$ for variables.) We will be a little informal in our treatment of environments, sometimes viewing them as sequences of bindings and sometimes as sets of bindings. We'll also make various simplifying assumptions; for example, we'll assume that each variable can only occur once in a given environment. With more care it's possible to formalise these assumptions, but the details are unnecessary for our purposes here.

As with types, we can use Γ-· and Γ-: to form a variety of environments. For example,

- The empty environment ·
- An environment with two variable bindings $\cdot, x : \mathcal{B}, f : (\mathcal{B} \to \mathcal{B})$

**Typing rules in $\lambda^{\to}$** The rule Γ-: shows how to add variables to an environment. We'll also need a way to look up variables in an environment. The following rule is the first of the three $\lambda^{\to}$ typing rules, which have the form $\Gamma \vdash M : A$ (read "the term $M$ has the type $A$ in environment Γ") and describe how to form terms:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{tvar}$$

The tvar rule shows how to type open terms (i.e. terms with free variables). If the environment Γ contains the binding $x{:}A$ then the term $x$ has the type $A$ in Γ.

The remaining two typing rules for $\lambda^{\to}$ show how to introduce and eliminate terms of function type — that is, how to define and apply functions.

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}\ \to\text{-intro} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \to B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B}\ \to\text{-elim}$$

The introduction rule →-intro shows how to form a term $\lambda x{:}A.M$ of type $A \to B$. You can read the rule as follows: "the term $\lambda x{:}A.M$ has type $A \to B$ in Γ if its body $M$ has type $B$ in Γ extended with $x{:}A$". Since we are *introducing* the → operator, the rule has → below the line but not above it.

The elimination rule →-elim shows how to apply terms of function type. The environment Γ is the same throughout the rule, reflecting the fact that no variables are bound by the terms involved. Since we are *eliminating* the → operator, the rule has → above the line but not below it.

The →-intro and →-elim form the first *introduction-elimination pair*[4]. We'll see many more such pairs as we introduce further type and kind constructors.

We illustrate the typing rules by giving derivations for the identity and compose functions (Figures 2.2 and 2.3)[5]

### 2.2.1 Adding products

Interesting programs typically involve more than functions. Useful programming languages typically provide ways of aggregating data together into larger structures. For example, OCaml offers various forms of variants, tuples and records, besides more elaborate constructs such as modules and objects. The $\lambda^{\rightarrow}$ calculus doesn't support any of these: there are no built-in types beyond functions, and its abstraction facilities are two weak to define interesting new constructs. In order to make the language a little more realistic we therefore introduce a built-in type of binary pairs (also called "products").

As before, we'll start by giving some programs that we can write using products:

- An `apply` function:

  $$\lambda p{:}(A{\rightarrow}B)\times A.\,\mathbf{fst}\ p\ (\mathbf{snd}\ p)$$

  In OCaml we write

  ```
  fun (f,p) -> f p
  ```

- A `dup` function:

  $$\lambda x{:}A.\,\langle x, x\rangle$$

  In OCaml we write

  ```
  fun x -> (x, x).
  ```

- The (bi)map function for pairs:

  $$\lambda f{:}A{\rightarrow}C.\,\lambda g.B{\rightarrow}C.\,\lambda p.A\times B.\,\langle f\ \mathbf{fst}\ p, g\ \mathbf{snd}\ p\rangle$$

  In OCaml we write

  ```
  fun f g (x,y) -> (f x, g y).
  ```

- The function which swaps the elements of a pair:

[4]It is possible to consider the environment and variable-lookup rules as an introduction and elimination pair for environments, but we won't take this point any further here.

[5]While the derivations may appear complex at first glance, they are constructed mechanically from straightforward applications of the three typing rules for $\lambda^{\rightarrow}$. If typing derivations featured extensively in the course we might adopt various conventions to make them simpler to write down, since much of the apparent complexity is just notational overhead.

$$\frac{\cdot, x{:}A \vdash x : A}{\cdot \vdash \lambda x{:}A.x : A \to A}\ \to\text{-intro}$$

Figure 2.2: The derivation for the identity function.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cdot, f{:}B \to C, g{:}A \to B, x{:}A \vdash f : B \to C \qquad \cfrac{\begin{array}{c}\cdot, f{:}B \to C, g{:}A \to B, x{:}A \vdash g : A \to B \\ \cdot, f{:}B \to C, g{:}A \to B, x{:}A \vdash x : A\end{array}}{\cdot, f{:}B \to C, g{:}A \to B, x{:}A \vdash g\ x : B}\ \to\text{-elim}}{\cdot, f{:}B \to C, g{:}A \to B, x{:}A \vdash f\ (g\ x) : C}\ \to\text{-elim}}{\cdot, f{:}B \to C, g{:}A \to B \vdash \lambda x{:}A.f\ (g\ x) : A \to C}\ \to\text{-intro}}{\cdot, f{:}B \to C \vdash \lambda g{:}A \to B.\lambda x{:}A.f\ (g\ x) : (A \to B) \to A \to C}\ \to\text{-intro}}{\cdot \vdash \lambda f{:}B \to C.\lambda g{:}A \to B.\lambda x{:}A.f\ (g\ x) : (B \to C) \to (A \to B) \to A \to C}\ \to\text{-intro}$$

Figure 2.3: The derivation for `compose`

```
λp:A×B.⟨snd p, fst p⟩
```

In OCaml we write

```
fun (x,y) -> (y,x)
```

.

**Kinding rules for ×** There is a new way of forming types $A \times B$, and so we need a new kinding rule.

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A \times B :: *}\text{kind-}\times$$

The kind-× rule is entirely straightforward: if $A$ and $B$ have kind $*$, then so does $A \times B$.

**Typing rules for ×** There are three new typing rules:

$$\frac{\begin{array}{c}\Gamma \vdash M : A \\ \Gamma \vdash N : B\end{array}}{\Gamma \vdash \langle M, N\rangle : A \times B}\times\text{-intro}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{fst } M : A}\times\text{-elim-1}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{snd } M : B}\times\text{-elim-2}$$

The ×-intro rule shows how to build pairs: a pair $\langle M, N\rangle$ of type $A \times B$ is built from terms $M$ and $N$ of types $A$ and $B$.

The ×-elim-1 and ×-elim-2 rules show how to deconstruct pairs. Given a pair $M$ of type $A \times B$, fst $M$ and snd $M$ are respectively the first and second elements of the pair. Unlike in OCaml, fst and snd are "keywords" rather than first-class functions. For particular types $A$ and $B$ we can define abstractions λp:A×B.**fst** p and λp:A×B.**snd** p, but we do not yet have the polymorphism required to give definitions corresponding to the polymorphic OCaml functions:

```
val fst : 'a * 'b -> 'a
let fst (a, _) = a
```

```
val snd : 'a * 'b -> 'b
let snd (_, b) = b
```

### 2.2.2 Adding sums

Product types correspond to a simple version of OCaml's records and tuples. We next extend $\lambda^{\rightarrow}$ with sum types, which correspond to a simple version of variants.

Here are some programs that we can write with $\lambda^{\rightarrow}$ extended with sums:

- The (bi-)map function over sums:

  ```
  λf:A→C.λg:B→C.λs:A+B.case s of x.f x | y.g y
  ```

In OCaml we write

```
fun f g s -> match s with Inl x -> f x | Inr y -> g y
```

- The function of type A+B→B+A which swaps the **inl** and **inr** constructors:

  ```
  λs:A+B. case s of x.inr [B] x | y. inl [A] y
  ```

  In OCaml we write

  ```
  function Inl x -> Inr x | Inr y -> Inl y
  ```

- The function of type A+A→A which projects from either side of a sum:

  ```
  λs:A+A. case s of x.x | y.y
  ```

  In OCaml we write

  ```
  function Inl x -> x | Inr y -> y
  ```

**Kinding rules for +** The kinding rule for sum types follows the familiar pattern:

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A + B :: *}\text{kind-+}$$

**Typing rules for +** There are three new typing rules:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl } [B]\ M : A + B}\text{+-intro-1}$$

$$\frac{\Gamma \vdash N : B}{\Gamma \vdash \text{inr } [A]\ N : A + B}\text{+-intro-2}$$

$$\frac{\Gamma \vdash L : A + B \quad \Gamma, x{:}A \vdash M : C \quad \Gamma, y{:}B \vdash N : C}{\Gamma \vdash \textbf{case}\ \text{L}\ \textbf{of}\ \text{x.M} \mid \text{y.N} : \text{C}}\text{+-elim}$$

The +-intro-1 and +-intro-2 rules show how to build values of sum type by *injecting* with **inl** or **inr**. In order to maintain the property that each term has a unique type we also require a type argument to **inl** and **inr**[6]

The +-elim rule shows how to deconstruct sums. We can deconstruct sum values $L$ of type $A$+$B$ if we can deconstruct both the left and the right summand. Given an OCaml variant definition

```
type plus = Inl of a | Inr of b
```

the **case** expression

```
case L of x.M | y.N
```

corresponds to the OCaml match statement

[6]These kinds of annotations are not needed in OCaml; can you see why?

**match** l **with** Inl x → m | Inr y → n

There is an appealing symmetry between the definitions of products and sums. Products have one introduction rule and two elimination rules; sums have two introduction rules and one elimination rule. We shall consider this point in more detail in chapters 4 and 5.

## 2.3 System F

The simply typed lambda calculus $\lambda^{\rightarrow}$ captures the essence of programming with functions as first-class values, an essential feature of functional programming languages. Our next calculus, **System F** (also known as the *polymorphic lambda calculus*) captures another fundamental feature of typed functional programming languages like OCaml and Haskell: parametric polymorphism.

We have already seen an example of the problems that arise in languages which lack support for parametric polymorphism. In Section 2.2.1 the fst and snd operations which project the elements of a pair were introduced as built-in operators with special typing rules. It would be preferable to be able to define fst and snd using the other features of the language, but it is clear that they cannot be so defined, since $\lambda^{\rightarrow}$ lacks even the facilities necessary to express their types. A similar situation often arises during the development of a programming language: the language is insufficiently expressive to support a feature that is useful or even essential for writing programs. The most pragmatic solution is often to add new built-in operations for common cases, as we have done with products[7]. In this chapter we take the alternative approach of systematically enriching the core language until it is sufficiently powerful to define new data types directly.

The difficulties caused by the lack of polymorphism go beyond pairs. We introduced $\lambda^{\rightarrow}$ by showing how to write the identity and compose functions and comparing the implementations with the corresponding OCaml code. In fact, the comparison is a little misleading: in OCaml it is possible to define a single identity function and a single compose function that work at all types, whereas in $\lambda^{\rightarrow}$ we must introduce a separate definition for each type. As the following examples show, System F allows us to define the functions in a way that works for all types.

- The polymorphic identity function of type $\forall\alpha{::}*.\alpha \to \alpha$:

  $\Lambda\alpha :: * . \lambda x : \alpha . x$

- The polymorphic compose function of type $\forall\alpha{::}*.\forall\beta{::}*.\forall\gamma{::}*.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$:

[7] Some examples from real languages: OCaml has a polymorphic equality operation which works for all first-order types, but which cannot be defined within the language; Haskell's deriving keyword supports automatically creating instances of a fixed number of built-in type classes; C99 provides a number of type-generic macros which work across numeric types, but does not offer facilities for the user to define such macros.

$$\Lambda\alpha::*.\Lambda\beta::*.\Lambda\gamma::*.\lambda \mathrm{f}:\beta\to\gamma.\lambda \mathrm{g}:\alpha\to\beta.\lambda \mathrm{x}:\alpha.\ \mathrm{f}\ \ (\mathrm{g}\ \ \mathrm{x})$$

- The polymorphic apply function of type $\forall\alpha{::}*.\forall\beta{::}*.(\alpha\to\beta)\times\alpha\to\beta$:

$$\Lambda\alpha::*.\Lambda\beta::*.\lambda \mathrm{p}:(\alpha\to\beta)\times\alpha.\ \mathbf{fst}\ \ \mathrm{p}\ \ (\mathbf{snd}\ \ \mathrm{p})$$

To put it another way, we can now use abstraction *within* the calculus ($\forall\alpha::*.A$) where we previously had to use abstraction *about* the calculus (For all types $A$ ...).

**Kinding rules for** $\forall$ As the examples show, System F extends $\lambda^{\to}$ with a type-level operator $\forall\alpha{::}*.-$ that binds a variable $\alpha$ within a particular scope. There are two new kinding rules:

$$\frac{\Gamma,\alpha{::}K\vdash A::*}{\Gamma\vdash\forall\alpha{::}K.A::*}\ \text{kind-}\forall \qquad\qquad \frac{\alpha{::}K\in\Gamma}{\Gamma\vdash\alpha::K}\ \text{tyvar}$$

The kind-$\forall$ rule builds *universal types* $\forall\alpha{::}K.A$. The type $\forall\alpha{::}K.A$ has kind $*$ under an environment $\Gamma$ if the type $A$ has kind $*$ under $\Gamma$ extended with an entry for $\alpha$.

The tyvar rule is a type-level analogue of the tvar rule: it allows type variables to appear within type expressions, making it possible to build open types (i.e. types with free variables). If the environment $\Gamma$ contains the binding $\alpha{::}K$ then the type $\alpha$ has the kind $K$ in $\Gamma$

These rules involve a new kind of variable into the language. *Type variables* are bound by $\forall$ and (as we shall see) by $\Lambda$, and can be used in place of concrete types in expressions. It is important to distinguish between type variables (for which we write $\alpha$, $\beta$, $\gamma$) and metavariables (written $A$, $B$, $C$). We use metavariables when we wish to talk about types without specifying any particular type, but type variables are part of System F itself, and can appear in concrete programs.

**Environment rules for** $\forall$ The tyvar rule requires that we extend the definition of environments to support type variable bindings (associations of type variables with their kinds):

$$\frac{\Gamma\text{ is an environment}\qquad K\text{ is a kind}}{\Gamma,\alpha{::}K\text{ is an environment}}\ \Gamma\text{-}{::}$$

**Typing rules for** $\forall$ Since we have a new type constructor $\forall$, we need a new pair of introduction and elimination rules:

$$\frac{\Gamma,\alpha{::}K\vdash M:A}{\Gamma\vdash\Lambda\alpha{::}K.M:\forall\alpha{::}K.A}\ \forall\text{-intro}\qquad \frac{\Gamma\vdash M:\forall\alpha{::}K.A\qquad\Gamma\vdash B::K}{\Gamma\vdash M\ [B]:A[\alpha:=B]}\ \forall\text{-elim}$$

The $\forall$-intro rule shows how to build values of type $\forall\alpha{::}K.A$ — that is, polymorphic values. The term $\Lambda\alpha{::}K.M$ has type $\forall\alpha{::}K.A$ in $\Gamma$ if the body $M$ has the type $A$ in $\Gamma$ extended with a binding for $\alpha$.

The $\forall$-elim rule shows how to use values of polymorphic type via a second form of application: applying a (suitably-typed) term to a *type*. If $M$ has a polymorphic type $\forall\alpha{::}*.A$ then we can apply it to the type $B$ (also written "*instantiate* it at type B") to obtain a term of type $A[\alpha := B]$ — that is, the type $A$ with all free occurrences of $\alpha$ replaced by the type $B$. (Once again, substitution needs to be carefully defined to avoid inadvertently capturing variables, and once again we omit the details.)

There is nothing in the OCaml language that quite corresponds to the explicit introduction and elimination of polymorphic terms. However, one way to view OCaml's implicit polymorphism is as a syntactic shorthand for (some) System F programs, where constructs corresponding to $\forall$-intro and $\forall$-elim are automatically inserted by the type inference algorithm. We will consider this view in more detail in Chapter 3 and describe OCaml's alternatives to System F-style polymorphism in Chapter 6.

### 2.3.1 Adding existentials

For readers of a logical bent the name of the new type operator $\forall$ is suggestive. Might there be a second family of type operators $\exists$? It turns out that there is indeed a useful notion of $\exists$ types: just as $\forall$ is used for terms which can be instantiated at *any* type, $\exists$ can be used to form types for which we have *some* implementation, but prefer to leave the details abstract. These *existential types* play several important roles in programming, and we will return to them on various occasions throughout the course. For example,

- There is a close connection between the $\forall$ and $\exists$ operators in logic and the type operators that we introduce here, which we will explore in Chapter 4.
- Just as in logic, there is also a close connection between the $\forall$ operator and the $\exists$ operator, which we will consider in more detail in Chapter 5.
- The types of modules can be viewed as a kind of existential type, as we shall see in chapter 6.
- OCaml's variant types support a form of existential quantification, as we shall see in Chapter 6.

As we shall see in Chapter 5, it is possible to encode existential types using universal types. For now we will find it more convenient to introduce them directly as an extension to System F.

**Kinding rules for $\exists$** Adding existentials involves one new kinding rule, which says that $\exists\alpha{::}K.A$ has kind $*$ if $A$ has kind $*$ in an extended environment:

$$\frac{\Gamma, \alpha{::}K \vdash A :: *}{\Gamma \vdash \exists\alpha{::}K.A :: *}\ \text{kind-}\exists$$

**Typing rules for ∃** The typing rules for ∃ follow the familiar introduction-elimination pattern:

$$\frac{\Gamma \vdash M : A[\alpha := B] \qquad \Gamma \vdash \exists\alpha{::}K.A :: *}{\Gamma \vdash \text{pack } B, M \text{ as } \exists\alpha{::}K.A : \exists\alpha{::}K.A} \;\exists\text{-intro}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \exists\alpha{::}K.A \\ \Gamma, \alpha{::}K, x{:}A \vdash M' : B\end{array}}{\Gamma \vdash \text{open } M \text{ as } \alpha, x \text{ in } M' : B} \;\exists\text{-elim}$$

The ∃-intro rule shows how to build values of existential type using a new construct, pack. A pack expression associates two types with a particular term $M$: if $M$ may be given the type $A[\alpha := B]$ in the environment $\Gamma$ for suitable $A$, $\alpha$ and $B$ then we may pack $M$ together with $B$ under the type $\exists\alpha{::}K.B$. It is perhaps easiest to consider the conclusion first: the expression pack $B, M$ as $\exists\alpha{::}K.A$ has the existential type $\exists\alpha{::}K.A$ if replacing every occurrence of $\alpha$ in $A$ with $B$ produces the type of $M$.

The ∃-elim rule shows how to use values of existential type using a new construct open. "Opening" a term $M$ with the existential type $\exists\alpha{::}K.A$ involves binding the existential variable $\alpha$ to the type $A$ and a term variable $x$ to the term $M$ within some other expression $M'$. It is worth paying careful attention to the contexts of the premises and the conclusion. Since $\alpha$ is only in scope for the typing of $M'$ it cannot occur free in the result type of the conclusion $B$. That is, the existential type is not allowed to "escape" from the body of the open.

**∃ examples** Existential types are more complex than the other constructs we have introduced so far, so we will consider several examples. Each of our examples encodes a data type using the constructs available in System F.

### 2.3.2 Encoding data types in System F

Our first example is a definition of the simplest datatype — that is, the "unit" type with a single constructor and no destructor. OCaml has a built-in unit type, but if it did not we might define its signature as follows:

```
type t
val u : t
```

The following System F expression builds a representation of the unit type using the type of the polymorphic identity function, which also has a single inhabitant:

```
pack (∀α::*.α → α,
      Λα.λa:α.a)
  as ∃υ::*.υ
```

In the examples that follow we will write 1 to denote the unit type and $\langle\rangle$ to denote its sole inhabitant.

Our next example defines a simple abstract type of booleans. OCaml has a built-in boolean type, but if it did not we might define a signature for bools as follows:

```
type t
val ff : t
val tt : t
val _if_ : t → 'a → 'a → 'a
```

That is: there are two ways of constructing boolean values, tt and ff, and a branching construct _if_ which takes a boolean and two alternatives, one of which it returns. We might represent boolean values using a variant type:

```
type boolean =
    False : boolean
  | True : boolean
```

Using boolean we can give an implementation of the signature:

```
module Bool :
sig
  type t
  val ff : t
  val tt : t
  val _if_ : t → 'a → 'a → 'a
end =
struct
  type t = boolean
  let ff = False
  let tt = True
  let _if_ cond _then_ _else_ =
    match cond with True → _then_ | False → _else_
end
```

If we ask OCaml to type-check the body of the module, omitting the signature, it produces the following output:

```
sig
  type t = boolean
  val ff : boolean
  val tt : boolean
  val _if_ : boolean -> 'a -> 'a -> 'a
end
```

The relationship between the inferred module type and the signature corresponds closely to the relationship between the supplied existential type and the type of the body in the rule ∃-intro. Substituting the actual representation type boolean for the abstract type t in the signature gives the module type of the body. We will explore this behaviour more fully in Chapter 6.

Just as we can use variants to define booleans in OCaml, we can use sums to define booleans in System F. Here is a definition analogous to the Bool module above, using the left injection for **false** and the right injection for **true**:

```
pack (1+1,
        ⟨inr [1] ⟨⟩,
        ⟨inl [1] ⟨⟩,
         λb:Bool.Λα::*.λr:α.λs:α.case b of x.s | y.r⟩⟩)
    as ∃β::*.
         β ×
         β ×
         β → ∀α::*.α → α.α
```

The unit and boolean examples encode data types with small fixed numbers of inhabitants. However System F is also sufficiently powerful to encode datatypes with infinitely many members, as we now proceed to show.

A type representing the natural numbers can be defined in OCaml as a variant with two constructors:

```
type nat =
     Zero : nat
   | Succ : nat -> nat
```

Using these constructors we can represent every non-negative integer: for example, the number three is represented as follows:

```
let three = Succ (Succ (Succ Zero))
```

We can encode the natural numbers in System F as follows:

```
pack (∀α::*.α →(α → α)→ α,
        ⟨Λα::*.λz:α.λs:α → α.z,
        ⟨λn:∀α::*.α →(α → α)→ α.
           Λα::*.λz:α.λs:α → α.s (n [α] z s),
        ⟨λn:∀α::*.α →(α → α)→ α.n⟩⟩⟩)
   as ∃ℕ :: ∗.
        ℕ ×
        (ℕ → ℕ) ×
        (ℕ → ∀α.α →(α → α)→ α)
```

As for booleans there are two constructors corresponding to the constructors of the data type and a branching operation (which we will call foldℕ) which makes it possible to define functions that discriminate between different numbers. The branching operation accepts a natural number of type ℕ, a type argument $\alpha$ specifying the type of the result, a value of type $\alpha$ to return in case the input is zero and a function of type $\alpha \to \alpha$ to use in case the input is the successor of some other number $n$. Using foldℕ we might define the function which tests whether a number is equal to zero by instantiating the result type with Bool and passing suitable arguments for the zero and successor cases:

```
λm:ℕ.λn:ℕ.foldℕ m [Bool] true (λb:Bool.false)
```

Similarly we might define the addition function using foldℕ by instantiating the result type with ℕ:

```
λm:ℕ.λn:ℕ.foldℕ m [ℕ] n succ
```

## 2.4 System F$\omega$

We motivated the introduction of System F with the observation that adding products to $\lambda^{\rightarrow}$ involves special typing rules for fst and snd, since $\lambda^{\rightarrow}$ does not support polymorphic operations. System F addresses this deficiency: we can express the types of fst and snd within the calculus itself, making it possible to abstract the operations. For example, here is a polymorphic function which behaves like fst:

```
Λα::*.Λβ::*.λp:α × β.fst p
```

However, System F shares with $\lambda^{\rightarrow}$ the problem that it is not possible to define a parameterised type of binary pairs within the calculus: we can build separate, unrelated definitions for Bool×Bool, ℕ×Bool, $\mathbb{N} \times \mathbb{N}$, and so on, but no single definition that suffices for all these types. The difficulty lies in the *kind* of the $\times$ operator. As the kind-$\times$ rule (Section 2.2.1) shows, $\times$ is applied to two type expressions of kind $*$ to build another such type expression. System F does not offer a way of introducing this type of parameterised type constructor, but the calculus that we now consider extends System F with exactly this facility.

The calculus System F$\omega$ adds a third type of λ-abstraction to the two forms that are available in System F. We already have λx:A.M, which abstracts terms to build terms, and Λ$\alpha$::K.M, which abstracts types to build terms. The new abstraction form λ$\alpha$::K.A abstracts types to build types.

Up until now the structure of kinds has been trivial, limited to a single kind *, to which all type expressions belonged. We now enrich the set of kinds with a new operator ⇒, allowing us to construct kinds which contain type operators and even higher-order type operators. The new type abstraction form λ$\alpha$::K.A allows us to populate these new kinds with type operators. We'll also add a corresponding type application form A B for applying type operators.

Let's start by looking at some examples of type expressions that we can build in this enriched language.

- The kind *⇒*⇒* expresses the type of binary type operators such as $\times$ and $+$. The following type expression abstracts such an operator and applies it twice to the unit type 1:

  ```
  λφ::*⇒*⇒*.φ 1 1
  ```

- The kind (*⇒*)⇒*⇒* expresses the type of type operators which are parameterised by a unary type operator and by a type. The following type expression, which applies the abstracted type operator twice to the argument, is an example:

$$\lambda\varphi :: * \Rightarrow * . \lambda\alpha :: * . \varphi \ \ (\varphi \ \ \alpha)$$

It is still the case that only type expressions of kind $*$ are inhabited by terms. We will continue to use the name “type” only for type expressions of kind $*$.

**Kinds in System F$\omega$** There is one new rule for introducing kinds:

$$\frac{K_1 \text{ is a kind} \qquad K_2 \text{ is a kind}}{K_1 \Rightarrow K_2 \text{ is a kind}} \Rightarrow\text{-kind}$$

It is worth noting that the addition of new kinds retroactively enriches the existing rules. For example, in the kind-$\forall$ rule the type variable $\alpha$ is no longer restricted to the kind $*$.

**Kinding rules for System F$\omega$** We have two new ways of forming type expressions, so we need two new kinding rules. The new rules form an introduction-elimination pair for the new kind constructor $\Rightarrow$, the first such pair at the type level.

$$\frac{\Gamma, \alpha{::}K_1 \vdash A :: K_2}{\Gamma \vdash \lambda\alpha{::}K_1.A :: K_1 \Rightarrow K_2} \Rightarrow\text{-intro} \qquad \frac{\begin{array}{c}\Gamma \vdash A :: K_1 \Rightarrow K_2 \\ \Gamma \vdash B :: K_1\end{array}}{\Gamma \vdash A\ B :: K_2} \Rightarrow\text{-elim}$$

The introduction rule $\Rightarrow$-intro shows how to form a type expression $\lambda\alpha{::}K_1.A$ of kind $K_1 \Rightarrow K_2$. Comparing it with the corresponding rule for terms, $\rightarrow$-intro, reveals that the structure of the two rules is the same.

The elimination rule $\Rightarrow$-elim shows how to apply type expressions to type expressions, and follows the pattern of the corresponding term-level application rule, $\rightarrow$-elim.

**Type equivalence** We have passed over one important aspect of type-level abstraction. The $\Rightarrow$-elim rule specifies that the domain kind of the type operator and the kind of the operand should be the same. But what do we mean by “the same”? In the earlier calculi a simple syntactic equality would do the trick: two types are the same if they are built from the same symbols in the same order (after removing any superfluous parentheses). Now that we have added type-level operations we need a more “semantic” notion of equality: two type expressions should be considered the same if they are the same once fully reduced — i.e., once all applications of $\lambda$-expressions have been eliminated. For simplicity we won’t go into any more detail about this aspect of System F$\omega$, but it is essential to a fully correct formal treatment.

### 2.4.1 Encoding data types in System F$\omega$

The new type-level programming facilities introduced in System F$\omega$ significantly increase the expressive power of the language, as we will see in the following examples.

**Encoding sums in System F$\omega$** We are finally able to encode the definitions of the sum and product abstractions directly within the calculus itself. Here is an encoding of sums using polymorphism. We expose a binary type operator and functions corresponding to the two constructors inl and inl, and to case.

$$
\begin{array}{l}
\textbf{pack}\ \lambda\alpha::*.\,\lambda\beta::*.\,\forall\gamma::*.\,(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma\,,\\
\quad\langle\Lambda\alpha::*.\,\Lambda\beta::*.\,\lambda\mathrm{v}:\alpha.\,\Lambda\gamma::*.\,\lambda\mathrm{l}:\alpha\to\gamma.\,\lambda\mathrm{r}:\beta\to\gamma.\,\mathrm{l}\ \ \mathrm{v}\\
\quad\ \langle\Lambda\alpha::*.\,\Lambda\beta::*.\,\lambda\mathrm{v}:\beta.\,\Lambda\gamma::*.\,\lambda\mathrm{l}:\alpha\to\gamma.\,\lambda\mathrm{r}:\beta\to\gamma.\,\mathrm{r}\ \ \mathrm{v}\\
\quad\ \ \Lambda\alpha::*.\,\Lambda\beta::*.\,\lambda\mathrm{c}:\forall\gamma::*.\,(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma.\,\mathrm{c}\rangle\rangle\\
\quad\textbf{as}\ \exists\varphi::*\Rightarrow*\Rightarrow*.\\
\qquad\ \forall\alpha::*.\,\forall\beta::*.\,\alpha\to\varphi\ \alpha\ \beta\\
\qquad\times\ \forall\alpha::*.\,\forall\beta::*.\,\beta\to\varphi\ \alpha\ \beta\\
\qquad\times\ \forall\alpha::*.\,\forall\beta::*.\,\varphi\ \alpha\ \beta\to\forall\gamma::*.\,(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma
\end{array}
$$

The encoding follows the same pattern as $\mathbb{N}$: a sum value is represented as a function with two parameters, one for each sum constructor. A left injection inl $M$ is represented as a function that passes $M$ to its first argument; a right injection inr $M$ as a function that passes $M$ to its second argument. Observe that the addition of higher kinds allows us to use create higher-kinded existential variables: $\varphi$ has kind $*\Rightarrow*\Rightarrow*$.

As we saw when we extended $\lambda^{\to}$ with polymorphism, the extra abstraction facilities enable us to express *in* the language what we could previously only express in statements *about* the language. We could previously say things like "For all binary type operators $\phi$"; now we can abstract over binary type operators within the calculus itself.

**Encoding lists in System F$\omega$** There is a simple connection between the Bool type that we could encode in System F and the sum type that we have encoded in System F$\omega$: instantiating the arguments of the sum type with 1 gives us Bool. Similarly, we could encode $\mathbb{N}$ in System F, but System F$\omega$ allows us to encode a list type, which we can think of as a kind of parameterised version of $\mathbb{N}$.

A definition of lists in OCaml has two constructors, Nil and Cons:

```
type 'a list =
    Nil : 'a list
  | Cons : 'a * 'a list -> 'a list
```

We therefore encode lists in System F$\omega$ using a function of two arguments, whose types reflect the types of the corresponding constructors[8]:

[8] We could easily define lists using an existential type as we have done for the other encod-

```
List = λα::*.∀φ ::**.φ α →(α → φ α → φ α)→ φ α
```

The constructor for the empty list is represented as a function which takes two arguments and returns the first:

```
nil = Λα::*.Λφ::*⇒*.λn:φ α.λc:α → φ α → φ α.n;
```

The function corresponding to Cons takes two additional arguments x and xs corresponding to the arguments of the Cons constructor:

```
cons = Λα::*.λx:α.λxs:List   .
          Λφ::*⇒*.λn:φ α.λc:α → φ α → φ α.
               c x (xs [φ] n c);
```

Finally, the destructor for lists corresponds to OCaml's List.fold_right function:

```
foldList = Λα::*.Λβ::*.λc:α → β → β.λn:β.λl:List α.l [λγ::*.β] n c
```

The analogue of the addition function that we defined using the encoding of ℕ is the binary append function for lists, which may be defined as follows:

```
append = Λα::*.
           λl:List α.λr:List α.
              foldList [α] [List α] (cons [α]) l r
```

We have seen how System F$\omega$ makes it possible to encode a number of common data types: unit, booleans, numbers, sums and lists. However, these encodings use relatively little of the expressive power of the calculus. We will finish with two slightly more exotic examples which illustrate some of the things that become possible with first class polymorphism and type-level abstraction.

**Encoding non-regular data types in System F$\omega$** Most data types used in OCaml are "regular": when defining of a type t, all occurrences of t within the definition are instantiated with the same parameters. For example, the tree constructor occurs four times on the right hand side of the following definition of a tree type, and at each occurrence it is applied to the parameter 'a:

```
type 'a tree =
  Empty : 'a tree
| Tree : 'a tree * 'a * 'a tree -> 'a tree
```

In contrast, in the following definition the argument of SuccP has the type ('a * 'a) perfect: the type constructor perfect is applied to the pair type 'a * 'a rather than to the parameter 'a:

```
type 'a perfect =
   ZeroP : 'a -> 'a perfect
 | SuccP : ('a * 'a) perfect -> 'a perfect
```

ings, but as the types grow larger the monolithic **pack** expressions become less readable, so we will switch at this point to presenting the components of the encoding individually

This kind of non-regular or "nested" type definition makes it possible to represent constraints on data that are difficult or impossible to capture using regular data type definitions. For example, whereas tree can represent trees with any number of elements, the perfect type can only be used to represent trees where the number of elements is a power of two.

The combination of type operators and polymorphism makes it possible to encode non-regular types in System F$\omega$. Here is a definition of a type corresponding to perfect:

```
Perfect = λα::*.∀φ::*⇒*.(∀α::*.α → φ α) →(∀α::*.φ (α × α)→ φ α)→ φ α
```

As in our other examples, there is an argument corresponding to each constructor of the type. In order to capture the non-regularity of the original type these arguments are themselves polymorphic functions.

The functions corresponding to ZeroP and SuccP follow the usual pattern, except that we must instantiate the polymorphic function arguments when applying them:

```
zeroP = Λα::*.λx:α.
          Λφ::*⇒*.λz:∀α::*.α → φ α.λs:φ (α × α)→ φ α.z [] x

succP = Λα::*.λp:Perfect (α × α).
          Λφ::*⇒*.λz:∀α::*.α → φ α.λs:(∀β::*.φ (β × β)→ φ β).
            s [α] (p [φ] z s)
```

We will have more to say about non-regular types in Chapter 7, since they are fundamental to GADTs.

**Encoding type equality in System F$\omega$** Our final example encodes a rather unusual data type which will also play a fundamental role in Chapter 7.

Perhaps you have encountered Leibniz's definition of equality, which states that objects should be considered equal if they behave identically in any context. We can express this notion of equality for types within System F$\omega$ as follows:

```
Eq = λα::*.λβ::*.∀φ::*⇒*.φ α → φ β
```

That is, for any types $\alpha$ and $\beta$ we can build a value Eq $\alpha$ $\beta$ if, for any unary type operator $\varphi$ we can convert from $\varphi\ \alpha$ to type $\varphi\ \beta$. (It might be supposed that we should also include the converse conversion $\varphi\ \alpha \to \varphi\ \beta$; we shall see in a moment why it's unnecessary to do so.)

Applying the Eq operator twice to any type $\alpha$ gives us a type Eq $\alpha$ $\alpha$, which is inhabited by a polymorphic identity function. We call the inhabitant refl, since it represents the reflexivity of equality:

```
refl = Λα::*.Λφ::*⇒*.λx:φ α.x
```

Similarly we can define values of the following types to represent the symmetry and transitivity properties:

```
symm  : ∀α::*.∀β::*.Eq α β → Eq β α
trans : ∀α::*.∀β::*.∀γ::*.Eq α β → Eq β γ → Eq α γ
```

Here are the definitions:

$$\text{symm} = \Lambda\alpha::*.\Lambda\beta::*.\lambda e:(\forall\varphi::*\Rightarrow*.\varphi\,\alpha \to \varphi\,\beta).e\ [\lambda\gamma::*.\text{Eq}\ \gamma\ \alpha]\ (\text{refl}\ [\alpha])$$
$$\text{trans} = \Lambda\alpha::*.\Lambda\beta::*.\Lambda\gamma::*.\lambda ab:\text{Eq}\ \alpha\ \beta.\lambda bc:\text{Eq}\ \beta\ \gamma.bc\ [\text{Eq}\ \alpha]\ ab$$

Finally, we can define a function lift whose type tells us that if two types $\alpha$ and $\beta$ are equal then $\varphi\ \alpha$ and $\varphi\ \beta$ are also equal, for any $\varphi$:

$$\text{lift} : \forall\alpha::*.\forall\beta::*.\forall\varphi::*\Rightarrow*.\text{Eq}\,\alpha\,\beta \to \text{Eq}\ (\varphi\,\alpha)\ (\varphi\,\beta)$$

Here is the definition of lift :

$$\text{lift} = \Lambda\alpha::*.\Lambda\beta::*.\Lambda\varphi::*\Rightarrow*.\lambda e:\text{Eq}\ \alpha\ \beta.e\ [\lambda\gamma::*.\text{Eq}\ (\varphi\,\alpha)\ (\varphi\,\gamma)]\ (\text{refl}\ [\varphi\,\alpha])$$

**Kind polymorphism** As the notation for type-level abstraction suggests, System F$\omega$ enriches System F with what amounts to a simply-typed lambda calculus at the type level. This observations suggests ways that we might further extend the abstraction facilities of the calculus — for example, we might add type-level polymorphism over kinds in the same way that we added term-level polymorphism over types. Polymorphism over kinds would allow us to generalize our definition of equality to arbitrary type operators.

## 2.5 A note on the lambda cube and dependent types

**Dependent types** In this chapter we've seen three different forms of abstraction:

- The first calculus, $\lambda^{\to}$ (Section 2.2), supports "term-to-term" abstraction. The $\to$-intro rule allows us to build *terms* $\lambda$x:A.M denoting functions that abstract over *terms*.
- The second calculus, System F (Section 2.3) adds support for "type-to-term" abstraction. The $\forall$-intro rule allows us to build *terms* $\Lambda\alpha$::K.M that abstract over *types*.
- The final calculus, System F$\omega$ (Section 2.4) adds support for "type-to-type" abstraction. The $\Rightarrow$-intro rule allows us to build *types* (or, more precisely, type-expressions) $\lambda\alpha$::K.A that abstract over *types* (strictly: type expressions).

At this point the question naturally arises: might there be a calculus which supports "term-to-type" abstraction? The (affirmative) answer comes in the form of a richly expressive set of "dependently typed" calculi. Dependent types form the basis of many tools used in computer assisted theorem proving (such as Coq[9] and Twelf[10]) and are increasingly making their way into practical programming languages such as Idris, Agda, and GHC Haskell. Dependent types

[9]https://coq.inria.fr/
[10]http://twelf.org/

are beyond the scope of this course, but we'll note in passing how the extra power they offer comes in handy when we look at Propositions as Types (Chapter 4) and Generalised Algebraic Data Types (Chapter 7).

**The lambda cube** We've presented the route from $\lambda^{\rightarrow}$ to System F to System F$\omega$ as a natural, almost inevitable progression. However, type-level abstraction and polymorphism are sufficiently independent that we could equally well have introduced them in the opposite order. In fact, the three ways of extending $\lambda^{\rightarrow}$ with additional forms of abstraction can be applied in any combination, leading to a collection of eight different calculi. This is sometimes visualised as a cube with $\lambda^{\rightarrow}$ at one corner and (a version of) the calculus of constructions $P\omega$, which supports all four forms of abstraction, in the farthest corner, as shown in Figure 2.4. The three systems ($\lambda^{\rightarrow}$, System F, and System F$\omega$) that we've considered in this chapter all lie on the left face.

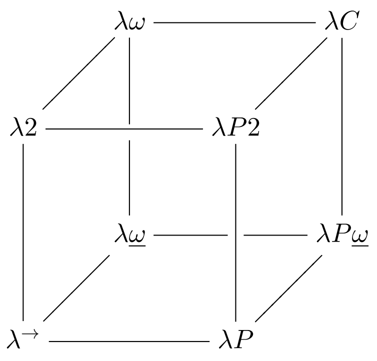


Figure 2.4: The "lambda cube"

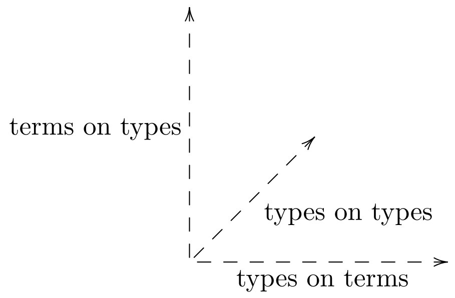


Figure 2.5: Dependencies in the lambda cube

**Key**

- $\lambda^{\rightarrow}$: simply typed lambda calculus (Section 2.2)
- $\lambda 2$: System F (Section 2.3)
- $\lambda\omega$: System F$\omega$ (Section 2.4)

## 2.6 Exercises

1. [★]: Give a typing derivation for swap
2. [★★]: We have seen how to implement booleans in System F using sums and the unit type. Give an equivalent implementation of booleans using polymorphism, using ℕ encoding as an example.
3. [★★★]: Implement a signed integer type, either using booleans, products and ℕ, or directly in System F. Give an addition function for your signed integers.
4. [★★]: Show how to encode the tree type in System F$\omega$.
5. [★★]: Write a function that computes the sum of a list of natural numbers in System F$\omega$.
6. [★]: Give an encoding of OCaml's option type in System F$\omega$:

```
type 'a option =
  None : 'a option
| Some : 'a -> 'a option
```

7. [★★★]: Use existentials, the list type, the product type, the ℕ encoding and your option type from question 6 to implement a stack interface corresponding to the following OCaml signature:

```
type 'a t
val empty : 'a t
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> 'a option * 'a t
val size : 'a t -> int
```

These notes aim to be self-contained, but fairly terse. There are many more comprehensive introductions to the typed lambda calculi available. The following two books are highly recommended:

- **Types and Programming Languages**
  Benjamin C. Pierce
  MIT Press (2002)
  `http://www.cis.upenn.edu/~bcpierce/tapl/`
  There are copies in the Computer Laboratory library and many of the college libraries.

- **Lambda Calculi with Types**
  Henk Barendregt
  in Handbook of Logic in Computer Science Volume II, Oxford University Press (1992)
  Available online: `http://ttic.uchicago.edu/~dreyer/course/papers/barendregt.pdf`