1.5 System Fω

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**:

$\Lambda\alpha::\star.\Lambda\beta::\star.\lambda\mathtt{p}:\alpha\times\beta.\,\mathbf{fst}\ \mathtt{p}$

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`, ℕ × ℕ, and so on, but no single definition that suffices for all these types. The difficulty lies in the *kind* of the × operator. As the kind-× rule (Section 1.2.1) shows, × 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 Fω adds a third type of λ-abstraction to the two forms that are available in System F. We already have $\lambda\mathtt{x}:\mathtt{A}.\mathtt{M}$, which abstracts terms to build terms, and $\Lambda\alpha::\mathtt{K}.\mathtt{M}$, which abstracts types to build terms. The new abstraction form $\lambda\alpha::\mathtt{K}.\mathtt{A}$ abstracts types to build types.

Up until now the structure of kinds has been trivial, limited to a single kind $\star$, 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 $\lambda\alpha::\mathtt{K}.\mathtt{A}$ 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 $\star\Rightarrow\star\Rightarrow\star$ expresses the type of binary type operators such as × and +. The following type expression abstracts such an operator and applies it to the types 1 and `Bool`:

  $\lambda\phi::\star\Rightarrow\star\Rightarrow\star.\,\phi\ \mathtt{1}\ \mathtt{Bool}$

- The kind $(\star\Rightarrow\star)\Rightarrow\star\Rightarrow\star$ 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:

  $\lambda\phi::\star\Rightarrow\star.\lambda\alpha::\star.\,\phi\ (\phi\ \alpha)$

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 Fω 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 α is no longer restricted to the kind $\star$.

Kinding rules for System Fω 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.

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

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 λ-expressions have been eliminated. For simplicity we won't go into any more detail about this aspect of System Fω, but it is essential to a fully correct formal treatment. (Pierce's book has the full details: see the *Type equivalence* rules on p451.)

1.5.1 Encoding data types in System Fω

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

Encoding sums in System Fω We are finally able to encode the definitions of the sum and product abstractions directly within the calculus itself. First, here is a definition of a sum type in OCaml:

```
type ('a, 'b) sum =
  Inl : 'a → ('a, 'b) sum
| Inr : 'b → ('a, 'b) sum
```

A sum type can represent values of either of two types, and so `sum` has two type parameters, `'a` and `'b`. There are two constructors, `Inl` and `Inr`. `Inl` (the left injection) takes a value of type `'a` and builds a `sum` value; `Inr` (the right injection) behaves similarly for `'b`.

The following function, `case`, examines a `sum` value and applies either `l` or `r` to the argument according to whether the value was constructed with `Inl` or `Inr`:

```
val case :
 ('a,'b) sum → ('a → 'c) → ('b → 'c) → 'c

let case s l r =
  match s with
    Inl x → l x
  | Inr y → r y
```

The return type `'c` is the same in either case.

In System Fω sums can be encoded using polymorphism. The `Sum` type constructor is a type-level function of two arguments, just as the OCaml `sum` accepts two type parameters:

$$\texttt{Sum} \;=\; \lambda\alpha.\lambda\beta.\forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$$

The encoding follows the same pattern as ℕ: a sum value is represented as a function with two parameters, one for each sum constructor.

Each of the `inl` and `inr` constructors is represented as a functions. 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.

$$\begin{aligned}
\textbf{inl} \;&=\; \Lambda\alpha.\Lambda\beta.\lambda\texttt{v}{:}\alpha.\Lambda\gamma. \\
&\qquad \lambda\texttt{l}{:}\alpha \to \gamma.\lambda\texttt{r}{:}\beta \to \gamma.\texttt{l v} \\
\textbf{inr} \;&=\; \Lambda\alpha.\Lambda\beta.\lambda\texttt{v}{:}\beta.\Lambda\gamma. \\
&\qquad \lambda\texttt{l}{:}\alpha \to \gamma.\lambda\texttt{r}{:}\beta \to \gamma.\texttt{r v}
\end{aligned}$$

Finally, since a `Sum` value is already represented as a function that behaves like `case`, the destructor (here called `foldSum`) is simply a version of the identity function:

$$\texttt{foldSum} = \Lambda\alpha.\Lambda\beta.\lambda\texttt{c}{:}\forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma.\texttt{c}$$

As with previous encodings the components `Sum`, `inl`, `inr`, `foldSum` can be packed up as an existential type:

pack `Sum`, $\langle$**inl**, $\langle$**inr**, `foldSum`$\rangle\rangle$
as $\exists\phi::\star\Rightarrow\star\Rightarrow\star.$
$\forall\alpha::\star.\forall\beta::\star.\alpha\to\phi\ \alpha\ \beta$
$\times\ \forall\alpha::\star.\forall\beta::\star.\beta\to\phi\ \alpha\ \beta$
$\times\ \forall\alpha::\star.\forall\beta::\star.\phi\ \alpha\ \beta\to\forall\gamma::\star.(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma$

Observe that the addition of higher kinds allows us to use create higher-kinded existential variables: ϕ has kind $\star\Rightarrow\star\Rightarrow\star$.

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 ϕ"; now we can abstract over binary type operators within the calculus itself.

Encoding lists in System Fω 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 Fω: instantiating the arguments of the sum type with `1` gives us `Bool`. Similarly, we could encode $\mathbb{N}$ in System F, but System Fω 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 Fω using a function of two arguments, whose types reflect the types of the corresponding constructors[8]:

`List` $=\ \lambda\alpha::\star.\forall\phi::\star\Rightarrow\star.\phi\ \alpha\to(\alpha\to\phi\ \alpha\to\phi\ \alpha)\to\phi\ \alpha$

The higher-kinded ϕ is a little more general than is strictly necessary here: a simpler encoding might replace $\phi\ \alpha$ with a single type variable β of kind $\star$ without significant loss. However, the approach of using a higher-kinded type constructor to represent a parameterised type (i.e. `list`) extends straightforwardly to the more exotic examples that follow, which cannot be expressed using only type expressions of kind $\star$.

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

`nil` $=\ \Lambda\alpha::\star.\Lambda\phi::\star\Rightarrow\star.\lambda$`n:`$\phi\ \alpha.\lambda$`c:`$\alpha\to\phi\ \alpha\to\phi\ \alpha.$`n;`

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

`cons` $=\ \Lambda\alpha::\star.\lambda$`x:`$\alpha.\lambda$`xs:List \alpha.`
$\Lambda\phi::\star\Rightarrow\star.\lambda$`n:`$\phi\ \alpha.\lambda$`c:`$\alpha\to\phi\ \alpha\to\phi\ \alpha.$

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

```
                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 $\mathbb{N}$ 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 Fω 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 Fω 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
```

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 Fω. Here is a definition of a type corresponding to `perfect`:

```
Perfect = λα::⋆.∀φ::⋆⇒⋆.(∀α::⋆.α → φ α) \to (∀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 lecture 8, since they are fundamental to GADTs.

Encoding type equality in System Fω Our final example encodes a rather unusual data type which will also play a fundamental role in lecture 8 (GADTs).

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 Fω as follows:

```
Eql = λα::⋆.λβ::⋆.∀ϕ::⋆⇒⋆.ϕ α → ϕ β
```

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

Passing any type α twice to the Eql operator gives us a type `Eql` α α, 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 to represent the symmetry and transitivity properties of equality. Here is the type of symmetry, which turns a value of type `Eql` α β into a value of type `Eql` β α for any α and β:

```
symm : ∀α::⋆.∀β::⋆.
         Eql α β → Eql β α
```

Defining `symm` involves a little ingenuity. The first steps are easy, following the type structure: we must abstract over types α and β, and accept a value of type `Eql $\alpha$ $\beta$` (here expanded for clarity):

```
symm = Λα::⋆.Λβ::⋆.λe:(∀ϕ::⋆⇒⋆.ϕ α → ϕ β) ....
```

Now we must construct a term of type `Eql` β α. The variable `e` is the only available term involving β and α, so we next look for a way to use `e`. Since `e` has type $\forall\phi::\star\Rightarrow\star.\ldots$ the first step in using `e` involves finding a suitable argument to take the place of ϕ. For reasons that will become clear very shortly we pick $\lambda\gamma::\star$`.Eql` γ α and pass it to `e`:

```
symm = Λα::⋆.Λβ::⋆.λe:(∀ϕ::⋆⇒⋆.ϕ α → ϕ β).
  e [λγ::⋆.Eql γ α] ...
```

Substituting our chosen argument in place of ϕ and simplifying the result reveals that `e` $[\lambda\gamma::\star$`.Eql` γ $\alpha]$ has type `Eql` α $\alpha \rightarrow$ `Eql` β α. To produce a value of type `Eql` β α it only remains to find a value of type `Eql` α α; the `refl` function constructed earlier suits the purpose perfectly, completing the definition:

```
symm = Λα::⋆.Λβ::⋆.λe:(∀ϕ::⋆⇒⋆.ϕ α → ϕ β).
  e [λγ::⋆.Eql γ α] (refl [α])
```

Defining `trans`, which represents the transitivity property of equality, is similar. The type of `trans` combines two `Eql` values: if α is equal to β and β is equal to γ then α is equal to γ:

```
trans : ∀α::⋆.∀β::⋆.∀γ::⋆.Eql α β → Eql β γ → Eql α γ
```

The definition of `trans` again involves picking a suitable instantiation for ϕ . This time, instantiating ϕ with `Eql` α in the second argument `bc` gives a function of type `Eql` α $\beta \rightarrow$`Eql` α γ, which can be applied to the first argument `ab` to complete the construction:

```
trans = Λα::⋆.Λβ::⋆.Λγ::⋆.λab:Eql α β.λbc:Eql β γ.bc [
    Eql α] ab
```

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

```
lift : ∀α::⋆.∀β::⋆.∀ϕ::⋆⇒⋆.Eq α β → Eq (ϕ α) (ϕ β)
```

Here is the definition of `lift`:

```
lift = Λα::⋆.
         Λβ::⋆.
           Λϕ::⋆⇒⋆.
             λe:Eql α β.
               e [λγ::⋆.Eql (ϕ α) (ϕ γ)] (refl [ϕ α])
```

Kind polymorphism As the notation for type-level abstraction suggests, System Fω enriches System F with what amounts to a simply-typed lambda calculus at the type level. This observation 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.

1.6 Exercises

1. [★★]: Show how to encode the `tree` type in System Fω.
2. [★★]: Write a function that computes the sum of a list of natural numbers in System Fω.
3. [★]: Give an encoding of OCaml's `option` type in System Fω:

```
type 'a option =
  None : 'a option
| Some : 'a → 'a option
```

4. [★★★]: Use existentials, the list type, the product type, the $\mathbb{N}$ encoding and your option type from question 3 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 books and resources 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

- **Advanced Topics in Types and Programming Languages**
  Benjamin C. Pierce (editor)
  MIT Press (2005)
  https://www.cis.upenn.edu/~bcpierce/attapl/
  Chapter 2 covers dependent types, including the Calculus of Constructions

- **The Part II Types lecture notes** Andrew M. Pitts
  https://www.cl.cam.ac.uk/teaching/1617/Types/types.notes.pdf