A Short Introduction to Systems F and F_ω

Pablo Nogueira

22nd February 2006

Contents

1 Introduction

These notes contain a short introduction to the syntactic, context-dependent, and operational aspects of System F and System F_ω. The former extends the Simply (and explicitly) Typed Lambda Calculus with (universal) parametric polymorphism. The latter extends it with type operators. The following references contain detailed presentations of the systems and of typed lambda calculi in general: [Car97, CW85, Pie02, Mit96].

2 Preliminaries

What is the Lambda Calculus? We can describe the Lambda Calculus [Chu41, Mit96, Bar84] as a tiny programming language of unnamed, first-class functions. It was originally conceived as a formal (*i.e.*, symbolic) language of untyped functions that was part of a proof system of functional equality. The logical part of the system was proved inconsistent but the language of functions was deem useful and gave rise to different families of languages that model different aspects of computation. Typed extensions with polymorphism, recursion, built-in primitives, plus naming and definitional facilities at value and type level make up the *core languages* of functional languages [Lan66, Pie02, Rea89]. Improvements to the core language's operational aspects form the basis of functional language implementations.

Object and meta language. We assume familiarity with the distinction between ***object language***, a particular formal language under study, and ***meta-language***, the notation used when talking about the object language.

Definitions and equality. It is common practice in mathematics to use equality as a definitional device. Since equality is also used as a relation on already defined entities, we distinguish equality from definitional equality and use the symbol $\stackrel{\text{def}}{=}$ for the latter. A definition induces an equality in the sense that if $X \stackrel{\text{def}}{=} Y$ then $X = Y$; the converse need not be true.

Format of typing rules. Type rules are written inductively in natural deduction style:

$$\frac{\text{antecedent}_1 \quad \ldots \quad \text{antecedent}_n}{\text{consequent}_1 \quad \ldots \quad \text{consequent}_n}$$

where $n \geq 0$. Rules can be read forwards (when all antecedents are the case *then* all the consequents are the case) or backwards (all the consequents are the case *if* all the antecedents are the case). Variables appearing in antecedents and consequents are assumed universally quantified unless indicated otherwise.

Grammatical conventions. We use EBNF notation to define syntax. Non-terminals are written in capitalised slanted. Any other symbol stands for a terminal with the exception of the postfix meta-operators `?`, $^+$, and *. Their meaning is as follows: `X?`, `X`*, and `X`$^+$ denote, respectively, zero or one `X`, zero or more `X`, and one or more `X`, where `X` can be a terminal or non-terminal.

Parentheses are also used as meta-notation for grouping, *e.g.*, `(X Y)`*.

The following EBNF example is a snippet of C++'s grammar [Str92]:

CondOp `::= if` *Cond Block* `(else` *Block*`)?`
Cond `::= (` *Expression* `)`
Block `::=` *Stmt* `|  {` *Stmt*$^+$ `}`

In the first production parentheses are meta-notation. In the second they are object-level symbols, for they are not followed by a postfix meta-operator.

Quantification. We follow the widely used and well-known "quantifier-dot" convention when denoting quantification. For example, in $\forall x.P$ the scope of bound variable x starts from the dot to the end of the expression P. Also,

$\forall x \in S.\ P$ abbreviates $\forall x.\ x \in S \Rightarrow P$

3 Pure Simply Typed Lambda Calculus

Type `::=` $*$ —— base type
`|` *Type* $\rightarrow$ *Type* —— function type
`|` `(`*Type*`)` —— grouping

Term `::=` *TermVar* —— term variable
`|` *Term Term* —— term application
`|` λ *TermVar* `:` *Type* `.` *Term* —— term abstraction
`|` `(`*Term*`)` —— grouping

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t_1 : \sigma \rightarrow \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash (t_1\ t_2) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash (\lambda x{:}\sigma\,.\,t) : \sigma \rightarrow \tau}$$

$$\frac{}{(\lambda x{:}\tau\,.\,t)\ t' \rhd t[t'/x]}\ \beta \qquad \frac{t_1 \rhd t_1'}{t_1\ t_2 \rhd t_1'\ t_2}\ \text{LI1} \qquad \frac{t_2 \rhd t_2'}{x\ t_2 \rhd x\ t_2'}\ \text{LI2}$$

$$\frac{}{t \rhd t}\ \text{REF} \qquad \frac{t_1 \rhd t_2 \qquad t_2 \rhd t_3}{t_1 \rhd t_3}\ \text{TRS}$$

Figure 1: The Pure Simply Typed Lambda Calculus.

Figure 1 shows the syntax, type rules, and operational semantics of the Pure Simply Typed Lambda Calculus (PSTLC). There are terms carrying type annotations and for that reason it is dubbed 'explicitly typed'—or *á la* Church, who first proposed it [HS86]. The following paragraphs elaborate.

Terms and types. The PSTLC has a language of types (non terminal *Type*) for expressing the types of functions inductively from a unique base or ground type $*$, and a language of terms

(non-terminal *Term*) which consists of variables, ***lambda abstractions*** (unnamed functions), and ***applications*** of terms to other terms.[1] Variables stand for formal parameters or yet-to-be-defined primitive values when not bound by any λ. In a lambda abstraction $\lambda x{:}\tau.t$, the λ symbol indicates that x is a bound variable—*i.e.*, a formal parameter—, τ is the type of x, and t abbreviates an expression where x may occur free. Table 2 contains the list of meta-variable conventions followed throughout these notes.

σ, τ, ...	range over types.
x, y, ...	range over term variables.
t, t', ...	range over terms.
Γ	ranges over type-assignments.
α, β, ...	range over type variables (Section 5).
κ, ν	range over kinds (Section 5).

Figure 2: Meta-variable conventions for Lambda Calculi.

Types and terms are separated with the only exception that types can appear as annotations in lambda abstractions. The type of a function is also called its ***type signature***. It describes the function's arity, order, and the type of its arguments. The ***arity*** is the number of arguments it takes. The ***order*** is determined from its type signature as follows:

$$\begin{aligned} \text{order}(*) &\stackrel{\text{def}}{=} 0 \\ \text{order}(\sigma \to \tau) &\stackrel{\text{def}}{=} \max\,(\ 1 + \text{order}(\sigma),\ \text{order}(\tau)\ ) \end{aligned}$$

Let τ be the type of a lambda abstraction and suppose $\text{order}(\tau) = n$. If $n = 1$ then the lambda abstraction may either return a manifest (non-function) value of type $*$ or another lambda abstraction of order 1 as result. If $n > 1$, then it is a higher-order abstraction that either takes or returns a lambda abstraction of order n.

It is typical to blur the conceptual distinction between manifest values and function values by considering the former as ***nullary functions*** and the latter as ***proper functions***.[2]

The ***fixity*** of a function is an independent concept. It determines the syntactical denotation of the application of the function to its arguments. In some functional languages, functions can be infix, prefix, postfix, mixfix, and have their precedence and associativity defined by programmers. In the PSTLC, lambda abstractions are prefix, application associates to the left—for example, $t_1\ t_2\ t_3$ is parsed as $(t_1\ t_2)\ t_3$—and consequently arrows in type signatures associate to the right—for example, $* \to * \to *$ is parsed as $* \to (* \to *)$.

Multiple-argument functions are represented as ***curried*** higher-order functions that take one argument but return another function as result. For example, the term:

$$\lambda x{:}*\,.\,\lambda y{:}* \to *\,.\,y\ x$$

is a higher-order function that takes a manifest value x and returns a function that takes a function

[1] That application is denoted by whitespace is not quite deducible from the grammar alone. In order for the two terms to be distinguished there must be some separator token between them which is assumed to be whitespace.

[2] In the PSTLC not every term is a function: there is a ground type. In the untyped lambda calculus, every term is a function.

y as argument and applies it to x.

Related terminology. An ***operator*** is a term whose value is a function (a precise definition of 'value' is given on page 5). An ***operand*** is a term that plays the role of a ***parameter*** or ***argument***. A ***formal*** parameter or argument appears in a definition whereas an ***actual*** parameter or argument appears in an application. A ***call site*** is another name for an ***application*** of an operator to an operand.

In common parlance, the word 'type' is not only used in reference to type-terms but also in reference to ***data types***, *i.e.*, a concrete realisation of the type in an implementation design or actual code.

Type rules. The type rules listed in Figure 1 can be employed to check the type of a term compositionally from the type of its subterms. The type of a term depends on the type of its free variables. This context-dependent information is captured by a type-assignment function $\Gamma : \mathit{TypeVar} \to \mathit{Type}$ which acts as a symbol table of sorts that stores the types of free variables in scope. The operation $\Gamma, x : \tau$ denotes the construction of a new type-assignment and has the following definition:

$$(\Gamma, x : \tau)(y) \stackrel{\text{def}}{=} \text{if } x = y \text{ then } \tau \text{ else } \Gamma(y)$$

The type rules are rather intuitive. Notice only that Γ is enlarged in the last rule because x may occur free in t.

Operational semantics. The call-by-name operational semantics is shown in the last box of Figure 1. A reduction relation $\triangleright$ is defined between terms. Briefly, Rule β captures the reduction of an application of a lambda abstraction to an argument. The free occurrences of the parameter variable are substituted (avoiding variable capture) by the argument in the lambda abstraction's body. This is what the operation $t[t'/x]$ means, which reads "t where t' is substituted for free x" [Bar84]. Rule LI1 specifies that an application $t_1\ t_2$ can be reduced to the term $t_1'\ t_2$ when t_1 can be reduced to t_1'. Rule LI2 specifies that reduction must proceed to the argument of an application when the term being applied is a free variable. Together, these rules specify a leftmost-outermost reduction order. Rules REF and TRS specify that $\triangleright$ is a reflexive and transitive relation.

A ***value*** is a program term of central importance. Operationally, the set of values V is a subset of the set of normal forms N, which is in turn a subset of the set of terms T, that is, $V \subseteq N \subseteq T$. These sets are to be fixed by definition. A term is in normal form if no reduction rule, other than reflexivity, is applicable to it. In the PSTLC, all normal forms are values and they are defined by the following grammar:

$$\mathit{NF} \ ::= \ \mathit{TermVar} \mid \lambda\ \mathit{TermVar}\ :\ \mathit{Type}\ .\ \mathit{Term}$$

That is: variables and lambda abstractions are normal forms, which means that function bodies are evaluated only after the function is applied to an argument. This is reflected in the operational

semantics by the deliberate omission of the following rule:

$$\frac{t \ \rhd \ t'}{\lambda x{:}\tau \,.\, t \ \rhd \ \lambda x{:}\tau \,.\, t'}$$

It can be the case in other languages that there are normal forms that are not values. Examples are ***stuck terms*** which denote run-time errors.

Example. The following derivation proves a reduction:

$$\dfrac{\dfrac{\dfrac{}{(\lambda x{:}* \to * \,.\, x)\ (\lambda x{:}* \,.\, x) \ \rhd \ (\lambda x{:}* \,.\, x)}\,\beta}{(\lambda x{:}* \to * \,.\, x)\ (\lambda x{:}* \,.\, x)\ y \ \rhd \ (\lambda x{:}* \,.\, x)\ y}\,\text{LI1} \quad \dfrac{}{(\lambda x{:}* \,.\, x)\ y \ \rhd \ y}\,\beta}{(\lambda x{:}* \to * \,.\, x)\ (\lambda x{:}* \,.\, x)\ y \ \rhd \ y}\,\text{TRS}$$

The following is an example reduction of a well-typed PSTLC term to its normal form. The subterm being reduced at each reduction step is shown underlined.

$$\begin{aligned}
&\underline{(\lambda y{:}* \to * \,.\, y\ z)\ ((\lambda y{:}* \to * \,.\, y)\ (\lambda x{:}* \,.\, x))} \\
&\quad \rhd \underline{((\lambda y{:}* \to * \,.\, y)\ (\lambda x{:}* \,.\, x))\ z} \\
&\quad \rhd \underline{(\lambda x{:}* \,.\, x)\ z} \\
&\quad \rhd z
\end{aligned}$$

4 Adding primitive types and values.

The PSTLC is impractical as a programming language. Given a term t, its free variables have no meanings. The PSTLC *extended* with various primitives has been given specific names. In particular, the language PCF (Programming Computable Functions) is a PSTLC extended with natural numbers, booleans, cartesian products, and fixed points [Sto77, Mit96].

In Figure 3 we *extend* the grammar of terms and types of Figure 1 to include some primitive types. The base type $*$ is now removed from the language of types. Of particular interest are cartesian product and disjoint sum types that endow the Extended STLC (referred to as STLC from now on) with algebraic types roughly similar to those supported by functional languages.

We only show a tiny sample of type and reduction rules for primitives, the latter called δ***-rules*** in the jargon, to illustrate how the extension goes. Primitive types are all manifest and therefore their order is 0.

```
Type ::= Nat                    -- naturals
       | Bool                   -- booleans
       | Type × Type            -- products
       | Type + Type            -- disjoint sums
       | 1                      -- unit type

Term ::= Num                        -- natural literals
       | true                       -- boolean literals
       | false
       | + | − | ...                -- arithmetic functions
       | not Term                   -- boolean functions
       | if Term then Term else Term
       | ...
       | (Term , Term)              -- pairs
       | fst Term
       | snd Term
       | Inl Term                   -- sums
       | Inr Term
       | case Term of Inl TermVar then Term ; Inr TermVar then Term
       | unit                       -- unit value
```

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{unit} : \mathbf{1}}$$

$$\frac{\Gamma \vdash t : \texttt{Bool} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2) : \tau}$$

$$\frac{}{(\texttt{if true then } t_1 \texttt{ else } t_2) \rhd t_1} \qquad \frac{}{(\texttt{if false then } t_1 \texttt{ else } t_2) \rhd t_2}$$

$$\frac{t \rhd t'}{(\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2) \rhd (\texttt{if } t' \texttt{ then } t_1 \texttt{ else } t_2)}$$

Figure 3: The (Extended) Simply Typed Lambda Calculus. The base type $*$ is removed from the language of types and new productions are added for terms and type-terms to those in Figure 1. Only a small sample of type and reduction rules are shown.

Example. The following is an example reduction of a well-typed STLC term:

$$\underline{(\lambda x{:}\texttt{Nat}\,.\,\texttt{if}\ x > 0\ \texttt{then}\ 1\ \texttt{else}\ x+1)\ ((\lambda y{:}\texttt{Nat}\,.\,y+y)\ 4)}$$
$$\triangleright\ \texttt{if}\ \underline{((\lambda y{:}\texttt{Nat}\,.\,y+y)\ 4)} > 0\ \texttt{then}\ 1\ \texttt{else}\ ((\lambda y{:}\texttt{Nat}\,.\,y+y)\ 4)+1$$
$$\triangleright\ \texttt{if}\ (\underline{4+4}) > 0\ \texttt{then}\ 1\ \texttt{else}\ ((\lambda y{:}\texttt{Nat}\,.\,y+y)\ 4)+1$$
$$\triangleright\ \texttt{if}\ \underline{8 > 0}\ \texttt{then}\ 1\ \texttt{else}\ ((\lambda y{:}\texttt{Nat}\,.\,y+y)\ 4)+1$$
$$\triangleright\ \underline{\texttt{if true then}\ 1\ \texttt{else}\ ((\lambda y{:}\texttt{Nat}\,.\,y+y)\ 4)+1}$$
$$\triangleright\ 1$$

5 Adding parametric polymorphism: System F

The STLC is not polymorphic. For example, the identity function for booleans and naturals is expressed by two syntactically different lambda abstractions:

$$(\lambda x{:}\texttt{Nat}\,.\,x) \ :\ \texttt{Nat} \to \texttt{Nat}$$
$$(\lambda x{:}\texttt{Bool}\,.\,x) \ :\ \texttt{Bool} \to \texttt{Bool}$$

However, they only differ in type annotations. ***System F*** [Gir72, Rey74] extends the STLC with *universal parametric polymorphism*. It adds new forms of abstraction and application where types appear as terms, not just annotations. The new syntax can be motivated using the above identity functions. A parametrically polymorphic identity is obtained by abstracting over types (***universal abstraction***), *e.g.*:

$$\Lambda\alpha{:}*\,.\,\lambda x{:}\alpha\,.\,x$$

This term has type:

$$\forall\alpha{:}*\,.\,\alpha \to \alpha$$

(We explain the role of $*$ in a moment.) A capitalised lambda 'Λ' is introduced to distinguish universal abstraction over types from term abstraction. Dually, there is ***universal application***, *e.g.*:

$$(\Lambda\alpha{:}*\,.\,\lambda x{:}\alpha\,.\,x)\ \texttt{Nat} \quad\triangleright\quad (\lambda x{:}\texttt{Nat}\,.\,x)$$

A universal application is evaluated by substituting the type-term argument for the free occurrences of the bound type-variable in the body of the universal abstraction. Here is another example; notice that type-variable β appears in a type-application:

$$(\Lambda\beta{:}*\,.\,(\Lambda\alpha{:}*\,.\,\lambda x{:}\alpha\,.\,x)\ \beta)\ \texttt{Nat} \quad\triangleright\quad (\Lambda\alpha{:}*\,.\,\lambda x{:}\alpha\,.\,x)\ \texttt{Nat}$$

Figure 4 shows the additions to the grammar, to the type rules, and to the operational semantics. Because of the introduction of type variables, rules for type-term well-formedness are provided (a few are shown in the first row of the second box).

We reintroduce $*$ at a new level and call it a base or ground *kind*. Kinds classify types and are explained in detail in Section 6; for the moment, type variables in universal abstractions always

$$
\begin{array}{l}
\mathit{Kind} \;\texttt{::=}\; * \\
\\
\mathit{Type} \;\texttt{::=}\; \mathit{TypeVar} \\
\qquad \texttt{|}\; \forall\ \mathit{TypeVar}\ \texttt{:}\ \mathit{Kind}\ \texttt{.}\ \mathit{Type} \to \mathit{Type} \\
\\
\mathit{Term} \;\texttt{::=}\; \Lambda\ \mathit{TypeVar}\ \texttt{:}\ \mathit{Kind}\ \texttt{.}\ \mathit{Term} \qquad -\!- \text{ universal abstraction} \\
\qquad \texttt{|}\; \mathit{Term}\ \mathit{Type} \qquad\qquad\qquad\qquad\quad -\!- \text{ universal application}
\end{array}
$$

$$
\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma \to \tau} \qquad \frac{\Gamma, \alpha : \kappa \vdash \sigma}{\Gamma \vdash \forall \alpha{:}\kappa\,.\,\sigma} \qquad \ldots
$$

$$
\frac{\Gamma, \alpha : \kappa \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha{:}\kappa\,.\,t) \;:\; (\forall \alpha{:}\kappa\,.\,\tau)} \qquad \frac{\Gamma \vdash t : (\forall \alpha{:}\kappa\,.\,\tau) \qquad \Gamma \vdash \sigma}{\Gamma \vdash t\ \sigma \;:\; \tau[\sigma/\alpha]}
$$

$$
\frac{}{(\Lambda \alpha{:}\kappa\,.\,t)\ \sigma \;\rhd\; t[\sigma/\alpha]}
$$

Figure 4: System F extensions.

have kind $*$, for they can only be substituted for base types which are all manifest, but we use in advance the kind meta-variable κ because the language of kinds is extended in Section 6. Type-assignments now also store the kinds of type variables.

The type rules for universal abstraction and application are shown in the second row of the second box. Notice that a type-level, capture-avoiding substitution operation is assumed which replaces type variables for types in terms. The last box in Figure 4 enlarges the reduction relation to account for universal applications. Universal abstractions are normal forms like regular term abstractions.

6 Adding type operators: System F_ω

System F_ω extends System F with ***type operators***, *i.e.*, functions at the type level. They are also called ***type constructors***, but we prefer to use 'constructor' at the value level when referring to the terms associated with type operators, called ***value constructors***. An example of type operator is `List` which when applied to a manifest type τ returns the type of lists of type τ. Its associated value constructors are:

```
Nil  :: ∀ a:∗. List a
Cons :: ∀ a:∗. a → List a → List a
```

which are names for primitive constants without δ-rules—*e.g.*, a term like `Cons` t `Nil` is in normal form, whatever the t. Manifest types such as `Nat` or `Bool` are 'values' at the type level. A fully applied (*i.e.*, closed) type operator also constitutes a manifest type, *e.g.*: `List Nat`. Occasionally,

we blur the distinction between manifest types and type operators by considering the former as ***nullary type operators*** and the latter as ***proper type operators***.

To model type-level functions, the PSTLC of Figure 1 is lifted to the type level as shown in the first box of Figure 5, so that terms such as α, $\lambda\alpha{:}\kappa\,.\,\tau$, and $\tau\ \sigma$ (that is, ***type variables***, ***type-level abstractions***, and ***type-level applications***) are defined as legal type-terms.

The ***kind*** of a type-term is somewhat inaccurately described as the 'type' of a type-term. *But kinds only describe the arity and order of type operators.* The kind of a nullary type operator (a manifest type) is $*$. The kind of a proper type operator is denoted as $\kappa \to \nu$, where κ is the kind of its argument and ν the kind of its result. The order of a type operator is determined from its ***kind signature*** as follows:

$$\begin{array}{lcl} \mathrm{order}(*) & \stackrel{\mathrm{def}}{=} & 0 \\ \mathrm{order}(\kappa \to \nu) & \stackrel{\mathrm{def}}{=} & \max\,(\ 1 + \mathrm{order}(\kappa),\ \mathrm{order}(\nu)\ ) \end{array}$$

Kinds do not have a status as the 'types' of types when there are orthogonal features in the type language, like type classes [Blo91, WB89, Jon92], which render them inaccurate as such. For instance, the following two Haskell definitions of the type operator `List` have the same kind, but the second is constrained on the range of type arguments:

```
data List a = Nil | Cons a (List a)
data Ord a ⇒ List a = Nil | Cons a (List a)
```

Type checkers *kind-check* applications of type operators to arguments to make sure the latter have the right expected kind. Kind-checking rules are shown in the second box of Figure 5. The first three lines establish the kinds of manifest types and also depict the kind-checking rules for primitive type operators such as $+$ and $\times$, etc. The last line contains the type rules of the PSTLC but lifted as kind rules (compare with Figure 1).

The third box in Figure 5 shows a *type-level reduction* relation $\blacktriangleright$ between type-terms that is reflexive, transitive, and compatible with all ways of constructing type-terms. The symbol `P` stands for a primitive type, *i.e.*, `Bool`, `Nat`, or $\mathbf{1}$. The reduction relation is static: type-level applications are reduced by the type-checker at *compile* time. (This relation is a simple example of type-level computation.)

The last box in Figure 5 defines a relation of ***structural type equivalence*** which specifies that two type-terms are equal when their structure is equal. The relation is reflexive, symmetric, transitive, and compatible with all ways of constructing types.

Normal forms of type-terms are type variables, primitive types, type-level abstractions, and type-terms of the form $\tau_1 \times \tau_2$, $\forall\alpha{:}\kappa\,.\,\tau_1$, $\tau_1 + \tau_2$, and $\tau_1 \to \tau_2$, when τ_1 and τ_2 are themselves in normal form.

Notice that there are three sorts of substitutions, all using the same notation. There are two at the term level (term variables replaced for terms in term abstractions and type variables for type-terms in universal abstractions) plus one at the type level (type variables for type-terms in

$$\begin{array}{l} \mathit{Kind} \texttt{ ::= } * \\ \qquad \texttt{|}\ \mathit{Kind} \rightarrow \mathit{Kind} \\ \\ \mathit{Type} \texttt{ ::= } \mathit{Type}\ \ \mathit{Type} \qquad\qquad\qquad\qquad -- \text{ type application} \\ \qquad \texttt{|}\ \lambda\ \mathit{TypeVar} \texttt{ : } \mathit{Kind} \texttt{ . } \mathit{Type} \quad -- \text{ type abstraction} \end{array}$$

$$\frac{}{\Gamma \vdash \texttt{Nat} : *} \qquad \frac{}{\Gamma \vdash \texttt{Bool} : *} \qquad \frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) \ : \ *} \qquad \frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash (\tau_1 \times \tau_2) \ : \ *}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash (\tau_1 + \tau_2) \ : \ *}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha{:}\kappa \,.\, \tau \ : \ *}$$

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \ : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : \kappa \rightarrow \nu \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash (\tau_1 \ \tau_2) \ : \nu} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \nu}{\Gamma \vdash (\lambda \alpha{:}\kappa \,.\, \tau) \ : \kappa \rightarrow \nu}$$

$$\frac{}{(\lambda \alpha{:}\kappa \,.\, \tau) \ \tau' \ \blacktriangleright \ \tau[\tau'/\alpha]} \qquad \frac{\tau_1 \ \blacktriangleright \ \tau_1'}{\tau_1 \ \tau_2 \ \blacktriangleright \ \tau_1' \ \tau_2} \qquad \frac{\tau_2 \ \blacktriangleright \ \tau_2'}{\mathsf{P} \ \tau_2 \ \blacktriangleright \ \mathsf{P} \ \tau_2'}$$

$$\frac{\tau_1 \ \blacktriangleright \ \tau_1' \qquad \tau_2 \ \blacktriangleright \ \tau_2'}{\tau_1 \ + \ \tau_2 \ \blacktriangleright \ \tau_1' \ + \ \tau_2'} \qquad \frac{\tau_1 \ \blacktriangleright \ \tau_1' \qquad \tau_2 \ \blacktriangleright \ \tau_2'}{\tau_1 \ \times \ \tau_2 \ \blacktriangleright \ \tau_1' \ \times \ \tau_2'} \qquad \frac{\tau_1 \ \blacktriangleright \ \tau_1' \qquad \tau_2 \ \blacktriangleright \ \tau_2'}{\tau_1 \ \rightarrow \ \tau_2 \ \blacktriangleright \ \tau_1' \ \rightarrow \ \tau_2'}$$

$$\frac{\tau \ \blacktriangleright \ \tau'}{\forall \alpha{:}\kappa \,.\, \tau \ \blacktriangleright \ \forall \alpha{:}\kappa \,.\, \tau'} \qquad \frac{}{\tau \ \blacktriangleright \ \tau} \qquad \frac{\tau_1 \ \blacktriangleright \ \tau_2 \qquad \tau_2 \ \blacktriangleright \ \tau_3}{\tau_1 \ \blacktriangleright \ \tau_3}$$

$$\frac{}{\tau \ \equiv \ \tau} \qquad \frac{\tau_1 \ \equiv \ \tau_2}{\tau_2 \ \equiv \ \tau_1} \qquad \frac{\tau_1 \ \equiv \ \tau_2 \qquad \tau_2 \ \equiv \ \tau_3}{\tau_1 \ \equiv \ \tau_3}$$

$$\frac{\tau_1 \ \equiv \ \sigma_1 \qquad \tau_2 \ \equiv \ \sigma_2}{\tau_1 \rightarrow \tau_2 \ \equiv \ \sigma_1 \rightarrow \sigma_2} \qquad \frac{\tau \ \equiv \ \sigma}{\forall x{:}\kappa \,.\, \tau \ \equiv \ \forall x{:}\kappa \,.\, \sigma} \qquad \frac{\tau \ \equiv \ \sigma}{\lambda \alpha{:}\kappa \,.\, \tau \ \equiv \ \lambda \alpha{:}\kappa \,.\, \sigma}$$

$$\frac{\tau_1 \ \equiv \ \sigma_1 \qquad \tau_2 \ \equiv \ \sigma_2}{\tau_1 \ \tau_2 \ \equiv \ \sigma_1 \ \sigma_2} \qquad \frac{}{(\lambda \alpha{:}\kappa \,.\, \tau) \ \tau' \ \equiv \ \tau[\tau'/\alpha]}$$

Figure 5: System F_ω adds type operators, a reduction relation ($\blacktriangleright$), and an equivalence relation ($\equiv$) on type-terms.

type-level abstractions).

7 Adding general recursion

All the languages described so far are strongly normalising, *i.e.*, terms and type-terms always reduce to normal form [Pie02, Mit96]. However, in order to use System F_ω for real programming we need to introduce some form of recursion. In this section we extend the language of terms and types to cater for *general recursive* functions and type operators. In functional languages, functions (term-level or type-level) are recursive when the function name is applied to another term within its own body. For instance, the recursive definition of the list type and the factorial function can be written in Haskell as follows:

```
data List a = Nil | Cons a (List a)
factorial n = if n==0 then 1 else n * factorial(n-1)
```

Which can be translated to the lambda notation of System F_ω as follows:

```
List = λa:∗. 1 + (a × (List a))
factorial = λn:Nat. if n==0 then 1 else n * factorial (n-1)
```

But term and type lambda abstractions are unnamed; the naming mechanism above is meta-notation. Recursion must be achieved indirectly. Let us abstract in both cases over the name of the function to remove the recursion:

```
List° = λf:∗ → ∗. λa:∗. 1 + (a × f a)
factorial° = λf:Nat→Nat. λn:Nat. if n==0 then 1 else n * f (n-1)
```

It is typical at this point to resort to meta-level arguments or denotational semantics to explain that the equations:

```
List = List° List
factorial = factorial° factorial
```

have a least solution in some semantic domain that gives meaning to System F_ω syntax. Such solution is the *least fixed point* of the equation. Fortunately, there is no need to resort to meta-level arguments. Operationally, recursive functions are reduced by unfolding their body at each recursive call. This unfolding can be carried out *at the object level* by two new primitive term and type constants `fix` and `Fix` respectively. They are called ***fixed-point operators*** because, semantically, they return the least fixed point of their argument.[3] Their type- and kind-signatures are respectively:

$$\texttt{fix} : \forall\tau{:}\kappa.\ (\tau \to \tau) \to \tau$$
$$\texttt{Fix} : \forall\kappa.\ (\kappa \to \kappa) \to \kappa$$

Their type-checking and reduction rules are shown in Figure 6. The intuition is that at the meta-level, the following equations must hold:

[3] Fixed-point operators are also denoted Y and μ in the literature.

$$\textit{Type} \;::=\; \texttt{Fix}\ \textit{Type}$$
$$\textit{Term} \;::=\; \texttt{fix}\ \textit{Term}$$

$$\frac{\Gamma \vdash \tau : \kappa \to \kappa}{\Gamma \vdash \texttt{Fix}\,\tau : \kappa} \qquad \frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \texttt{fix}\,t : \tau}$$

$$\frac{}{\texttt{Fix}\ (\lambda\alpha{:}\kappa\,.\,\tau) \blacktriangleright \tau[\alpha/\texttt{Fix}\ (\lambda\alpha{:}\kappa\,.\,\tau)]} \qquad \frac{}{\texttt{fix}\ (\lambda x{:}\alpha\,.\,t) \rhd t[x/\texttt{fix}\ (\lambda x{:}\alpha\,.\,t)]}$$

Figure 6: Extension of System F_ω with fixed-point operators.

```
(fix f°) = f° (fix f°)
(Fix F°) = F° (Fix F°)
```

which turned into reduction rules give:

```
(fix f°) ▷ f° (fix f°)
(Fix F°) ▶ F° (Fix F°)
```

but since `F°` and `f°` abbreviate respectively type and term lambda abstractions, we have:

```
(fix (λx:α.t)) ▷ (λx:α.t) (fix (λx:α.t))
               ▷ t[x/(fix (λx:α.t))]

(Fix (λα:κ.τ)) ▶ (λα:κ.τ) (Fix (λα:κ.τ))
               ▶ τ[α/(Fix (λα:κ.τ))]
```

Figure 6 collects these steps in a single reduction rule. Call-by-name guarantees that recursive calls are unfolded only when their value is required.

Going back to our examples, the terms:

```
Fix (λf:∗→∗. λa:∗. 1 + (a × f a))
fix (λf:Nat→Nat. λn:Nat. if n==0 then 1 else n ∗ f (n-1))
```

are both legal System F_ω syntax that represent the recursive list type operator and the factorial function. The following is an example reduction that demonstrates the unfolding (ellipsis abbreviate some subexpressions and reduction steps):

```
(fix (λf:Nat→Nat. λn:Nat. if n==0 then 1 else n ∗ f (n-1))) 2
▷ (λn:Nat. if n==0 then 1 else n ∗ (fix ...) (n-1)) 2
▷ if 2==0 then 1 else 2 ∗ (fix ...) (2-1)
▷ if false then 1 else 2 ∗ (fix ...) (2-1)
▷ 2 ∗ (fix ...) (2-1)
▷ 2 ∗ (λn:Nat. if n==0 then 1 else n ∗ (fix ...)) (2-1)
▷ 2 ∗ if (2-1)==0 then 1 else (2-1) ∗ (fix ...) ((2-1)-1)
```

```
...
▷ 2 * if false then 1 else (2-1) * (fix ...) ((2-1)-1)
▷ 2 * (2-1) * (fix ...) ((2-1)-1)
...
▷ 2 * (2-1) * 1
...
▷ 2
```

Et Voilà.

References

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam, revised edition, 1984.

[Blo91] Stephen Blott. *Type Classes*. PhD thesis, Glasgow University, Glasgow, Scotland, UK, 1991.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997. Revised 2004.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computer Surveys*, 17(4):471–522, December 1985.

[Gir72] Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. London Mathematical Society, Student Texts 1. Cambridge University Press, 1986. Reprint 1993.

[Jon92] Mark P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Oxford University, July 1992.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, March 1966. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Rea89] Chris Reade. *Elements of Functional Programming*. International Series in Computer Science. Addison-Wesley, 1989.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, Massachusetts, 1977.

[Str92] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2nd edition, 1992.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.