Vrije Universiteit Brussel
Faculteit Wetenschappen

An Introduction to Polymorphic Lambda Calculus with Subtyping.

Kim Mens
(e-mail: kimmens@is1.vub.ac.be)

Techreport vub-tinf-tr-94-01

October 16, 1994

Department of Computer Science
TINF(WE)
VUB
Pleinlaan 2
B-1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anonymous FTP: progftp.vub.ac.be
World Wide Web: progwww.vub.ac.be

An Introduction to Polymorphic Lambda Calculus with Subtyping.

Abstract

In this paper, an elaborate overview is presented of several extensions of standard *lambda calculus*. We start out with a description of untyped lambda calculus. Then we add explicit *types* and show how *polymorphism* can be introduced. Next we give a description of a *higher order* polymorphic typed lambda calculus. In this system (called F_ω) types, type constructors, constructors of type constructors and so on can all be considered as first class values. The system can still be augmented with a notion of *subtyping*, yielding system $F_\omega^{\leq}$. The importance of $F_\omega^{\leq}$ lies in its ability to model a variety of fundamental concepts of object oriented programming. To facilitate this description of object oriented features some further extensions are sometimes made. More specifically, records are often introduced to model objects, existential quantification to model encapsulation, and recursive types to model late binding of self.

1. Table of contents

Abstract
1. Table of contents
2. An overview of lambda calculus
3. Type free lambda calculus
    - 3.1. An introduction to type free lambda calculus
    - 3.2. Syntax
    - 3.3. Inference and reduction rules
        - 3.3.1. Reduction rules
        - 3.3.2. Equational theory
    - 3.4. Theoretical properties
4. Typed lambda calculus
    - 4.1. Extending type free lambda calculus with explicit types
        - 4.1.1. Types
        - 4.1.2. Contexts
    - 4.2. Syntax
    - 4.3. Inference and reduction rules
        - 4.3.1. Typing
        - 4.3.2. Reduction rules
        - 4.3.3. Equational theory
    - 4.4. Theoretical properties
5. Second-order typed lambda calculus
    - 5.1. Extending typed lambda calculus with polymorphism
    - 5.2. Syntax
    - 5.3. Inference and reduction rules
        - 5.3.1. Typing
        - 5.3.2. Reduction rules
        - 5.3.3. Equational theory
    - 5.4. Theoretical properties
6. The systems F_n of polymorphic typed lambda calculus

6.1. Motivation
6.2. Construction of the systems F_n
7. System F_ω of polymorphic typed lambda calculus
7.1. Syntax
7.2. Inference and reduction rules
7.2.1. Typing
7.2.2. Kinding
7.2.3. Reduction rules
7.2.4. Equational theory
7.3. Theoretical properties
8. System $F_\omega^{\leq}$ of polymorphic typed lambda calculus
8.1. Extending system F_ω with subtyping
8.2. Syntax
8.3. Inference and reduction rules
8.3.1. Typing
8.3.2. Kinding
8.3.3. Subtyping
8.3.4. Reduction rules
8.3.5. Equational theory
8.4. Theoretical properties
9. Further extensions to system $F_\omega^{\leq}$
9.1. Records
9.1.1. Example
9.1.2. Syntax and rules
9.2. Existential quantification
9.2.1. Example
9.2.2. Syntax and rules
9.3. Recursive types
9.3.1. Example
9.3.2. Syntax and rules
9.3.3. Fixed-point operator
10. Conclusion
Acknowledgements
References

2. An overview of lambda calculus

Lambda calculus (λ-calculus) is a formal system, originally intended as a tool to study the notion of functions in mathematics, but now mainly used to study the concepts of algorithms and computability. There is a growing interest in various kinds of lambda calculus models among computer scientists. The appeal of these abstract theories is due to their elementary, yet quite universal nature. In lambda calculus several important programming concepts are present (or can be expressed faithfully in them) in their most pure form and without the restrictions that are sometimes imposed by commercial programming languages. For example lambda calculus is used as a foundation for functional programming languages (Miranda, Haskell, Scheme, Lisp), just like Turing machines model computations in imperative languages (Pascal, C). Typed variants of lambda calculus are among other things used for computer-aided design and verification of formal reasonings.

Church invented **type free lambda calculus** in the thirties. In this calculus, everything represents a function. Numbers (cf. Church numbers), booleans and data structures can be represented by appropriate functions. Yet there is only one type: the type of functions from values to

values, where all the values are themselves functions of the same type. Since everything has the *same* type, this calculus is called a "type free" (or untyped) calculus.

There are also typed versions of lambda calculus. Type assignment is interesting for the following reasons. First of all, the type of a term gives a *partial specification* of what the function denoted by that term is supposed to do. Usually this type as specification is given before the term as program is constructed. The verification of whether this term, once it is constructed, is indeed of the required type, provides a *partial correctness proof* for the program. Finally, types play a role in *efficiency*. If it is known that a subterm of a program has a certain type, then this subterm may be executed more efficiently by making use of this type information.

The use of type expressions to describe the functional behaviour of untyped lambda terms was first introduced by Curry. In his theory, terms are the same as in type free lambda calculus, but types are implicitly assigned to them. In other words, type information is automatically extracted from the expressions of the language. Consequently, programmers do not need to explicitly give the type of a certain expression, because the type can be implicitly deduced using a type inference algorithm, embodying a system of inference rules. An example of this approach is the so-called **Curry system with intersection types** [CaCo90]. These intersection types make it possible to state that a variable has a finite number of types at the same time. The information expressed by an intersection type is necessarily stronger than any member type of the intersection. The use of intersection types provides a direct basis for subtyping and polymorphism, by using the ordering of types induced by the various intersection types.

Church introduced a **typed lambda calculus** in an entirely different way, by explicitly specifying a type at each introduction of a bound variable. Thus in Church's system [Chur40] the terms are annotated versions of the type free terms, and each term has a unique type that is derivable from the way it is annotated. In this text, we will focus on such explicitly typed versions of lambda calculus.

A further extension is the introduction of *polymorphism* in our typed lambda calculus. Polymorphic functions are functions whose operands can have more than one type. An important distinction has been made between two major kinds of polymorphism: *parametric* polymorphism and *ad-hoc* polymorphism (see [CaWe85]). Parametric polymorphism is obtained when a function works uniformly on a range of types that exhibit some common structure. The functions that exhibit parametric polymorphism are sometimes called *generic functions*. Ad-hoc polymorphism is obtained when a function works, or appears to work, on a finite set of potentially unrelated different types. In terms of implementation, a generic function will execute the same code for arguments of any admissible type, whereas an ad-hoc polymorphic function may execute different codes for each type of argument.

In 1971, Girard ([Gira72], [Huet90] chapter 6) introduced the idea of parametric polymorphism into typed lambda calculus, by adding a scheme of abstraction with respect to types (terms dependent on types). The same system of *polymorphic typed lambda calculus* was independently reinvented by Reynolds ([Reyn74], [Huet90] chapter 5) in 1974. Both formulations were essentially the same, although both authors were led to the construction of the language by entirely different motivations. Because this system is often referred to as ***second-order* typed lambda calculus**, and to conform with the notations used in the rest of this paper, we will call[1] this system $\mathbf{F_2}$.

If we allow formation of new types by allowing quantification over connectives of higher kinds, then we get a series of **systems $\mathbf{F_n}$** of more and more powerful languages culminating in F_ω. System F_1 is Church's ordinary typed lambda calculus, and F_2 corresponds to Girard's and Reynolds' second-order typed lambda calculus. F_3 is obtained from F_2 by allowing type constructors that transform existing types into new types. By adding quantification over constructors of successively higher kinds, we obtain the languages F_4, F_5, F_6,... The union of all these languages is called $\mathbf{F_\omega}$.

This system F_ω can still be augmented with a notion of *bounded quantification*. With bounded quantification, the type parameter of a second order expression is not allowed to range over the universe of all types, but only over a restricted subset of types. This is a natural extension of type system F_ω, allowing us to deal with subtyping. A simple version of such a system is **system $\mathbf{F_\omega^{\leq}}$**, described in the appendices of [HoPi92], [HoPi94] and [PiTu92].

---

[1] Girard [Gira72], [GiLT90] simply called it system F.

$F_\omega^{\leq}$ can also be constructed in an alternative way. Starting from system F_2, Cardelli and Wegner [CaWe85] proposed a calculus $\mathbf{F}^{\leq}$ of second-order bounded quantification. The subtype relation used in $F_\omega^{\leq}$ can be viewed as a straightforward higher order extension (due to Cardelli and Mitchell) of the relation $\leq$ used in $F^{\leq}$.

The importance of $F_\omega^{\leq}$ lies in its ability to model a variety of fundamental concepts of object oriented programming. (For example *universal quantification* can be used to model generic functions and bounded quantification to model subtypes and type inheritance.) To facilitate the description of object oriented concepts in $F_\omega^{\leq}$ some further extensions are sometimes made to the system.

One extension that has been widely used is to augment the lambda calculus with ***records***. The major contributions in that direction are due to Luca Cardelli ([Card88], [CaMi89]). A similar approach was followed by Michael Wand in [Wand89]. Modelling objects by means of records introduces a dimension for inheritance and subtyping, since records can be ordered according to their sets of fields.

In [PiTu92] and [Pier93] ***existential quantification*** and *packaging* (information hiding) are used to model abstract data types or encapsulation.

Recursive types are sometimes used to model the behaviour of self and to introduce a notion of late binding. Instead of directly including recursive types in system $F_\omega^{\leq}$, an alternative approach is to use **F-bounded quantification** [CCHM89]. This is an improvement over bounded quantification that seems useful whenever recursive type definitions and subtyping are used. However, in this text only the former approach will be discussed.

3. Type free lambda calculus

3.1. An introduction to type free lambda calculus

Lambda calculus is a completely formal defined system, consisting of expressions representing functions and reduction rules that prescribe how to evaluate these functions. The simplest expressions are constants and variables. There are only two expression forming operators to build more complex expressions from simple ones: *application* and *abstraction*. An *application* is used to apply an expression (called the *operator*) to another expression (called the *operand*). In type free lambda calculus, every expression may be applied to another, without any restrictions. An *abstraction* is a mechanism that allows us to make a unary function of a given expression. Functions with more than one argument can be formed by repeated abstraction. (This process is called *currying*.)

Let us give some examples of valid expressions. The identity function is a function that may be applied to an arbitrary expression `x` and always yields this expressions itself. It can be defined as follows:

```
λx.x
```

The doubling function yields the composition of an arbitrary function `y` with itself and may be specified as:

```
λy.λx.y y x
```

3.2. Syntax

The set of *terms* or *lambda expressions* is defined by the following abstract grammar:

```
<term> ::=   <variable>
       |     <constant>
       |     λ<variable>.<term>          abstraction
       |     <term> <term>               application
       |     (<term>)
```

• *Variables* x,y,z,... are defined as elements of an infinite collection of variables. Variables can be used as formal as well as actual parameters in the definition or application of a function.

• *Constants* a,b,c,... are assumed to be elements of a countable collection of constants.
• An *abstraction* is an expression of the form λx.E and denotes a unary function, which maps its argument x on the expression E. Here, λ is a binding operator binding the variable x in λx.E. The body E of a lambda abstraction extends as far to the right as possible: to the end of the whole expression, or up to un unmatched right parenthesis. An occurrence of a variable x in a lambda expression is said to be *bound* if it appears in the body of a subexpression of the form λx.E; otherwise it is *free*. A *closed* term is one with no free variables.
• An *application* is an expression of the form F E and denotes the application of the first lambda expression to the second. When the lambda expression F to be applied is itself a compound expression, for example F = λx.x, then we have to put the expression between parentheses, otherwise λx.x E would be read as: "the function which maps x on the application x E" instead of "the identity function λx.x applied to the term E". That is why the use of parentheses must be allowed in the definition of terms. Of course we will agree to drop these parentheses whenever there is no confusion possible. Furthermore, we will adopt the usual convention that application associates to the left.

For example,	(((λx.λy.x)(5))(4))	can be simplified to	(λx.λy.x) 5 4
and	λy.λx.y y x	is an abbreviation of	λy.λx.y(y(x))

3.3. Inference and reduction rules

3.3.1. Reduction rules

The first reduction scheme is α-reduction which expresses the fact that two lambda expressions are considered essentially the same if they differ only in the names of their bound variables.

λx.A = λy.{y/x}A if y not free in A (α)

The condition "*y not free*" in A is introduced to avoid the capture of free variables y in A. The notation {z/x}A where A is a term and x,z are variables denotes the **renaming** of every occurence of x to z in A, and is defined by induction on the construction of A as follows:

1)	{z/x}x = z	
2)	{z/x}y = y	if x≠y
3)	{z/x}λx.A = λz.{z/x}A	
4)	{z/x}λy.A = λy.{z/x}A	if x≠y
5)	{z/x}F A = {z/x}F {z/x}A	
6)	{z/x}(A) = ({z/x}A)	

The β-rule expresses the fact that when a unary operator with argument x and body A is applied on an operand B, this application can be simplified to the lambda expression resulting from the substitution of the actual parameter B for the formal parameter x in the body A.

(λx.A)B = [B/x]A (β)

The notation [B/x]A where A and B are terms and x is a variable denotes the **substitution** of B for all free occurrences of x in A, and is defined by induction on the construction of A as follows:

1)	[B/x]x ≅ B	
2)	[B/x]y ≅ y	if x≠y
3)	[B/x]λx.A ≅ λx.A	
4)	[B/x]λy.A ≅ λy.[B/x]A	if x≠y and either x is bound in A or y is bound in B
5)	[B/x]λy.A ≅ λz.[B/x]{z/y}A	if x≠z, y≠z, x≠y and z neither free nor bound in A(B) and x free in A and y free in B

```
6)      [B/x]F A ≅ [B/x]F [B/x]A
7)      [B/x](A) ≅ ( [B/x]A )
```

Remarks:

• The symbol ≅ denotes α-equality. Intuitively two expressions are α-equal, if they can be transformed into one another by means of the α-rule.[2] Thus the result of a substitution as defined above is not a unique lambda expression, since it is defined up to an α-equality.

• In case 5 of the definition we had to introduce a new bound variable z in order to avoid the capture of the free occurrence(s) of `y` in `B`. Let us give an example of such a situation.

```
          (λx.λy.x)y
→         [y/x]λy.x                    (β)
=         λz.[y/x]{z/y}x               (def. substitution)
=         λz.[y/x]x                    (def. renaming)
=         λz.y                         (def. substitution)
```

This example illustrates that part 5 of the definition of substitution indeed solves the problem of capture of free variables. If we simply would have reduced `(λx.λy.x)y` by changing every occurence of `x` in `λy.x` with `y`, then the resulting expression would have been `λy.y` (the identity function) instead of the function `λz.y`.

Although some authors prefer to omit it, for completeness we will include the extensionality axiom (the η-rule).[3] Models satisfying (η) seem more natural, since (η) (in combination with the other axioms and rules) implies that two functions are equal whenever they give equal results for all arguments.

```
λx.F x = F          if x not free in F        (η)
```

3.3.2. Equational theory

The following inference rules constitute an equational theory for lambda expressions.[4]

```
A = B                                A = B; B = C
-----                (symmetry)      -------------      (transitivity)
B = A                                A = C

F = G; A = B                         A = B
-------------        (congruence)    -----------        (ξ)
F A = G B                            λx.A = λx.B
```

Remarks:

• An implicit parameter of the inference system is the equality relation = on terms. We will assume that this relation is defined by the given reduction rules `(α)`, `(β)` and `(η)`.

• It is not really necessary to include a reflexivity axiom

```
A = A
```

because it follows from `(β)` by the symmetry and transitivity rules above.[5]

---

[2] Formally, two lambda expressions A and B are α-equal if A can be transformed in B using the α-rule, or if B is obtained from A by replacing a subexpression S of A with a lambda expression T such that S can be transformed in T by using the α-rule, or if there is some lambda expression C such that A ≅ C and C ≅ B. (see [Reve88])

[3] λ-calculus extended with η-reduction as an axiom is sometimes referred to as λη-calculus.

[4] A lambda theory is a set of equations which contains all instances of rules (α) and (β), and is closed under the inference rules (symmetry), (transitivity), (congruence) and (ξ). Furthermore, the theory is extensional if it also contains all instances of rule (η).

[5] Proof of the reflexivity rule:

```
1. (λx.A)x      = [x/x]A                (β)
                = A                     (definition of substitution)
2.      A       = (λx.A)x               (1) ← (symmetry)
3.      A       = A                     (2),(1) ← (transitivity)
```

3.4. Theoretical properties

The computational power of type free lambda calculus is the same as that of Turing machines: all recursive functions can be expressed in it.

A very good introduction to type free lambda calculus and some of its properties can be found in [Reve88]. Two expressions are β-*equivalent* if they can be transformed into one another by applying the β-rule a number of times on these expressions or on one of their subexpressions. However, the question whether two lambda expressions are β-equivalent (up to α-quivalence[6]) is algorithmically undecidable. One of the main reasons for this is the fact that in untyped lambda calculus, it is possible to construct terms for which β-reduction never terminates[7]. Fortunately, if a lambda expression has at least one terminating β-reduction, then one can always find the *normal form* of the expression by following a *normal order reduction* strategy. (An expression is in normal form if it cannot be reduced any further using β-reduction.) Moreover, if a lambda expression has a normal form then every terminating β-reduction will result in the same normal form (up to α-congruence). In other words, the order in which the different subexpressions of a given expression are β-reduced is irrelevant as long as the reduction terminates. This is an immediate consequence of the so-called *Church-Rosser theorem*[8]. This theorem also implies that the equality problem of lambda expressions having normal forms is decidable.

4. Typed lambda calculus

4.1. Extending type free lambda calculus with explicit types

4.1.1. Types

The type free lambda calculus of previous section represents a general framework also for its typed versions. We already mentioned (page 3) that we would only look at explicitly typed versions of lambda calculus[9]. Such a typed lambda calculus is like type free lambda calculus, except that every variable must be explicitly typed when introduced as a bound variable. For example, the identity function λx.x for integers in typed lambda calculus will have the form:

```
λx:Integer.x
```

The doubling function for booleans has the following form:

```
λy:Boolean→Boolean.λx:Boolean.y y x
```

Every lambda expression must have a type. The assignment of types to lambda expressions is straightforward. First, we assume that we have a fixed set of ground types (called *type-constants*) from which all types are built (e.g. `Boolean`, `Integer`, `Real`, ...). Arbitrary type expressions Φ, Ψ, Θ, …are built from the ground types using the connective →. A type of the form Φ→Ψ (where Φ and Ψ are type expressions) represents the type of a function that maps elements of type Φ on elements of type Ψ. The → symbol for constructing function types is our only type constructor, but that is quite sufficient for the time being, since we have only two expression forming operations: abstraction and application.

Next we have to define a set of rules to assign a type to each lambda expression. We also have to rewrite the reduction rules `(α)`, `(β)` and `(γ)` of typed lambda calculus. They will all remain essentially the same, except for an extra requirement of type consistency. E.g. the β-rule

```
(λx.A)B = [B/x]A
```

of ordinary lambda calculus remains the same, except that `x` and `B` must be of the same type. Because of this extra requirement, the syntax of typed lambda expressions will be more complex than that of the type free notation. Furthermore, we will have to introduce a notion of *contexts* to assign a type to free variables occurring in lambda expressions.

---

6 i.e. up to a renaming of bound variables

7 `Ω = (λx.x x)(λx.x x)` is an example of such a non-terminating term. (see `[Bare84]`)

8 The Church-Rosser theorem states that if `E` β-reduces to `M` and `E` β-reduces to `N`, then there is some `Z` such that `M` β-reduces to `Z` and `N` β-reduces to `Z`.

9 Such as the one described by Church in [Chur40].

4.1.2. Contexts

Because the type of a typed lambda expression depends on the context in which it occurs, we must know the types of all free variables in this expression before a type can be assigned to it. Hence we define a ***context*** (or *environment*, or *type-assignment*) Γ as a finite set

$\Gamma = \{x_1:\Psi_1, \ldots, x_k:\Psi_k\}$

of associations of types to variables, with no variable x_i appearing twice in Γ ([Huet90], Chapter 10). Notice that Γ can also be seen as a function (from some finite set of variables to the set of types), e.g. $\Gamma(x_i)$ denotes the type Ψ_i associated with the variable x_i in the context Γ. Another convenient notation is

$\Gamma, x:\Phi$

for the context

$\Gamma \cup \{x:\Phi\}$

where in writing this we assume that x does not yet occur in Γ. In other words, $\Gamma, x:\Phi$ is a new context resulting from the extension of Γ with a new typing $x \in \Phi$.

4.2. Syntax

The set of *terms* for typed lambda calculus is almost the same as for type free lambda calculus. The only difference is that each time a variable is bound using a λ, its type must be explicitly specified. So an *abstraction* is now an expression of the form $\lambda x:\Phi.E$ and represents a unary function, taking an argument x of type Φ and mapping it on the expression E. Hence the terms of typed lambda calculus are defined by:

```
<term> ::=<variable>
     |     <constant>
     |     λ<variable>:<type>.<term>                    abstraction
     |     <term> <term>                                application
     |     (<term>)
```

Constants can be introduced naturally in typed lambda calculus by assigning a type to each of them. For example, the operator `and` is assigned the type `Boolean→Boolean→Boolean` and is applicable only to lambda expressions of type `Boolean` and its application to other lambda expressions is in error[10].

The set of types can also be characterised by an abstract grammar:

```
<type> ::=   <type-constant>
     |       <type>→<type>                              function type
     |       (<type>)
```

• Type *constants* α, β, γ,… are assumed to be elements of a non-empty[11] countable collection of type constants.

• A *function type* Φ→Ψ denotes the type of a function that expects an argument of type Φ, and maps this argument on a result of type Ψ. Following standard practice, the → symbol associates to the right, for example

`Boolean→Boolean→Boolean`

is parsed as

`Boolean→(Boolean→Boolean)`

which is the type of a function taking a Boolean as argument and resulting in a function on Booleans. Parentheses are used when it is necessary to override this convention, for example when one wants to write the type of a function expecting a Boolean function as argument and yielding a Boolean value:

`(Boolean→Boolean)→Boolean`

---

[10] When we say that application to other lambda expressions is in error, we mean that these applications will not be well-typed, and therefore the corresponding rule will not be fired.

[11] The set of type-constants should be non-empty, because if there are no type-constants, then there is no base case for the inductive definition of the set of types, and therefore the set of types would be empty. Furthermore, since terms must be typed, the set of terms would be empty as well.

The set of contexts is given by:

```
<context> ::=  ∅                                  empty context
     |         <context>,<variable>:<type>        variable binding
```

• A *variable binding* Γ,x:Φ extends the context Γ with a binding of the variable x to the type Φ.
• The *empty context* ∅ is the context that contains no bindings.

4.3. Inference and reduction rules

4.3.1. Typing

The relationship between ordinary expressions and type expressions is expressed by formulas called *typings* (or *type judgments*). Let Γ be a context, E a term and Φ a type expression. Then Γ ⊢ E ∈ Φ
is a typing[12] that asserts that E takes on type Φ when its free variables are assigned types by Γ and is read as "E has type Φ with respect to Γ". If the context Γ is empty we simply write
⊢ E ∈ Φ
The valid typings are defined by the inference rules below.

The only axioms about the typing relation are:

```
⊢ c ∈ Φ        iff c is a constant of type Φ   (constant)

Γ ⊢ x ∈ Γ(x)   for each variable x in Γ        (variable)
```

The type derivation rule (→ elimination) can be used to derive the type of an application:

```
Γ ⊢ F ∈ Φ→Ψ ; Γ ⊢ A ∈ Φ
————————————————————————                       (→ elimination)
Γ ⊢ F A ∈ Ψ
```

For example,
if Γ ⊢ F ∈ Int→Boolean and Γ ⊢ A ∈ Int
then Γ ⊢ F A ∈ Boolean

The dual rule (→ introduction) derives the type of an abstraction:

```
Γ,x:Φ ⊢ E ∈ Ψ
——————————————                                 (→ introduction)
Γ ⊢ λx:Φ.E ∈ Φ→Ψ
```

For example, these type inference rules can be used to deduce:

```
⊢ λx:Boolean→Integer.λy:Real→Boolean.λz:Real.x y z
       ∈ (Boolean→Integer)→(Real→Boolean)→Real→Integer
```

Proof:

1. {x:Boolean→Integer,y:Real→Boolean,z:Real} ⊢ z ∈ Real
   by (variable)

---

[12] Note that the symbols " : " and "∈ " used in this paper have intuitively similar meanings, since both declare something to have a particular type. The difference between them is that ":" is part of the syntactic language - it is used *within* terms or contexts to declare the types of variables - whereas "∈ " is a notation of the metalanguage used to make statements *about* terms.

2. {x:Boolean→Integer,y:Real→Boolean,z:Real} ⊢ y ∈ Real→Boolean
   by (variable)
3. {x:Boolean→Integer,y:Real→Boolean,z:Real} ⊢ y z ∈ Boolean
   by (→ elimination) on (2) and (1)
4. {x:Boolean→Integer,y:Real→Boolean,z:Real} ⊢ x ∈ Boolean→Integer
   by (variable)
5. {x:Boolean→Integer,y:Real→Boolean,z:Real} ⊢ x y z ∈ Integer
   by (→ elimination) on (4) and (3)
6. {x:Boolean→Integer,y:Real→Boolean} ⊢ λz:Real.x y z ∈ Real→Integer
   by (→ introduction) on (5)
7. {x:Boolean→Integer} ⊢ λy:Real→Boolean.λz:Real.x y z
   ∈ (Real→Boolean)→Real→Integer
   by (→ introduction) on (6)
8. ∅ ⊢ λx:Boolean→Integer.λy:Real→Boolean.λz:Real.x y z
   ∈ (Boolean→Integer)→(Real→Boolean)→Real→Integer
   by (→ introduction) on (7)

4.3.2. Reduction rules

Since we write terms with type assignments, it is natural to include assignments in equations as well. By equation, we will mean an expression

$\Gamma \vdash A=B \in \Phi$

where $\Gamma \vdash A \in \Phi$ and $\Gamma \vdash B \in \Phi$. Intuitively, an equation

$\{x_1:\Psi_1,\ldots,x_k:\Psi_k\} \vdash A=B \in \Phi$

means, "if the variables $x_1,\ldots,x_k$ have types $\Psi_1,\ldots,\Psi_k$ respectively, then terms A and B denote the same element of type Φ".

The following axioms for terms constitute the reduction rules:

$$\frac{\Gamma,x:\Phi \vdash A \in \Psi \ ;\ y \text{ not free in } A \ ;\ y \text{ not in } \Gamma}{\Gamma \vdash \lambda x:\Phi.A = \lambda y:\Phi.\{y/x\}A \in \Phi\to\Psi} \qquad (\alpha)$$

As already mentioned before, the β-rule will need a consistency condition which checks whether the actual and formal parameter are of the same type.

$$\frac{\Gamma \vdash B \in \Phi \ ;\ \Gamma,x:\Phi \vdash A \in \Psi}{\Gamma \vdash (\lambda x:\Phi.A)B = [B/x]A \in \Psi} \qquad (\beta)$$

In typed lambda calculus, the extensionality axiom (η) implies that two elements of functional type Φ→Ψ are equal whenever they give equal results for all elements of type Φ.

$$\frac{\Gamma \vdash F \in \Phi\to\Psi \ ;\ x \text{ not free in } F}{\Gamma \vdash \lambda x:\Phi.F\ x = F \in \Phi\to\Psi} \qquad (\eta)$$

4.3.3. Equational theory

The equational theory remains essentially the same. The only difference is a small difference in notation, because now we have to take care of the contexts.

$$\frac{\Gamma \vdash A = B \in \Phi}{\Gamma \vdash B = A \in \Phi} \qquad \text{(symmetry)}$$

$$\frac{\Gamma \vdash A = B \in \Phi;\ \Gamma \vdash B = C \in \Phi}{\Gamma \vdash A = C \in \Phi} \qquad \text{(transitivity)}$$

$$\frac{\Gamma \vdash F = G \in \Phi\to\Psi;\ \Gamma \vdash A = B \in \Phi}{\Gamma \vdash F\ A = G\ B \in \Psi} \qquad \text{(congruence)}$$

$$\frac{\Gamma, x{:}\Phi \vdash A = B \in \Psi}{\Gamma \vdash \lambda x{:}\Phi.A = \lambda x{:}\Phi.B \in \Phi\to\Psi} \qquad (\xi)$$

As before, it is not necessary to introduce a separate reflexivity axiom.

4.4. Theoretical properties

Typed lambda calculus shares a number of interesting theoretical properties with the other systems that will be discussed in this paper. One of the most important is the fact that only terminating computations can be expressed. I.e. the given reduction rules (β) and (η) are *strongly normalizing*, which means that there is no infinite reduction sequence on any term. This stands in sharp contrast to untyped lambda calculus, where non-normalizable terms are easy to construct. A proof of the strong normalisation of typed lambda calculus can be found in [GiLT90]. An immediate consequence of this property is the decidability of β-equality (up to a renaming of bound variables).

The *Church-Rosser theorem* of type-free lambda calculus, which states that it does not matter which reducable subexpressions in a term are reduced first, also holds for typed lambda calculus. Together with the normalisation theorem, the Church-Rosser property for typed lambda calculus guarantees that every well-typed term reduces in a finite number of steps to a unique normal form.

5. Second-order typed lambda calculus

5.1. Extending typed lambda calculus with polymorphism

On page 3 we already pointed out that both Girard [Gira72] and Reynolds [Reyn74] developed a formulation of second-order typed lambda calculus, which was essentially the same. In fact, there was only a difference in notations.[13] This system F_2 of polymorphic lambda calculus allows the definition of *polymorphic* (or *generic*) functions that can accept arguments of a variety of types. These polymorphic functions are formed by explicit lambda abstraction over types. As illustrated in [GiLT90] this operation is extremely powerful and in particular all the usual data-types (integers, lists, trees, etc.) are definable with it.

A typical example of such a generic function is the definition of a polymorphic identity function. In the ordinary typed lambda calculus one can write

$\lambda x{:}\texttt{Boolean}.x$

to denote the identity function for the type `Boolean`. Unfortunately, to express the same function on integers, essentially the same function has to be rewritten, but with the type `Boolean` replaced by `Integer`:

$\lambda x{:}\texttt{Integer}.x$

In second-order polymorphic lambda calculus it is possible to write a *polymorphic* identity function which can generate all these identity functions of different types. To write such a polymorphic identity function, we start by replacing the type `Boolean` (in the identity function for Booleans) by a type variable ψ

$\lambda x{:}\psi.x$

[13] E.g. instead of using the symbol $\forall$ to denote the type of polymorphic functions, Reynolds used the symbol Δ. Girard also considered a type constructor $\exists$ which is "dual" to $\forall$ and related to existential quantification in logical formulas. Later on (page 29), we will show how existential quantification can be introduced.

Then, by explicitly abstracting on this type variable, we get the polymorphic identity function[14]:

Λψ.λx:ψ.x

that can be applied (or *instantiated*) to any type to obtain the identity function for that type, e.g.

(Λψ.λx:ψ.x)[Integer]

gives (using some kind of β-rule that allows type expressions to be substituted for occurrences of type variables in ordinary terms):

λx:Integer.x

Let us look at another example: the "doubling" function for the type `Integer` which accepts a function from integers to integers and yields the composition of this function with itself can be written in the simple typed lambda calculus as

λy:Integer→Integer.λx:Integer.y y x

Using a type variable ψ, we get

λy:ψ→ψ.λx:ψ.y y x

By abstracting on the type variable, we obtain the polymorphic doubling function

Λψ.λy:ψ→ψ.λx:ψ.y y x

that can be applied to any type resulting in the doubling function for that type.

To accommodate such a scheme of abstraction with respect to types, it is necessary to expand the variety of type expressions to provide types for the polymorphic functions. What is the type of a polymorphic function? It is something like an → type, since it is a function. But then again, it is not an ordinary function, since it takes a type as argument and returns a term. Therefore we want a different notation from →. With Girard, we will use the symbol ∀ to denote the type of polymorphic functions. But since the type of the result returned by such a polymorphic function can vary based upon the argument given to it, we also need an explicit way of indicating this dependence. For example, one writes

∀ψ.Φ

to denote the type of a polymorphic function that yields a result of type Φ (dependent on ψ) when applied to a type ψ. If a term `E` has type Φ then Λψ.`E` has type ∀ψ.Φ, and if a polymorphic function `F` has type ∀ψ.Φ then `F[`Ψ`]` has the type obtained from Φ by substituting Ψ for ψ in Φ. For example the type of the polymorphic identity function is[15]

∀ψ.ψ→ψ

which is the type of (polymorphic) functions associating to each type ψ a term of type ψ→ψ.
The polymorphic doubling function Λψ.λy:ψ→ψ.λx:ψ.y y x has type ∀ψ.(ψ→ψ)→(ψ→ψ).

Note that the system we are introducing is *explicitly* rather than *implicitly* polymorphic [PiDM89]. In languages with explicit polymorphism, a polymorphic function must be applied explicitly to a type argument to give a monomorphic instance, which can then be applied to term arguments. On the other hand, implicitly polymorphic languages (such as ML) generally omit types from the concrete syntax. Implicitly polymorphic functions may be applied directly to terms of different types; the task of determining the intended monomorphic instance is left to the interpreter or compiler.

5.2. Syntax

We extend the set of *terms* for typed lambda calculus by introducing a scheme of abstraction with respect to types. I.e. we add two new constructs, namely *type abstraction* and *type application* to the syntax of the terms.

```
<term> ::=...
     |     Λ<type-variable>.<term>                    type abstraction
     |     <term>[<type>]                             type application
```

---

[14] Notice the use of a capital lambda Λ to denote the abstraction over types, to remind ourselves that we are dealing with a function from *types* to terms.

[15] Intuitively, the domain of the polymorphic identity function is the collection of all types, and its range is the union of **all** types of the form ψ→ψ. That is why Girard used the notation ∀ψ.ψ→ψ.

• The syntax of constants remains unchanged, but now each constant must have a fixed, closed[16] type.
• A *type abstraction* is an expression of the form Λψ.E, where ψ is a type-variable. Such an expression denotes a polymorphic function, parametrised by the type variable ψ. Like λ, Λ is a binding operator binding the type variable ψ in Λψ.E.
• A *type application* F[Ψ] denotes the application of a type abstraction F to a type Ψ. To remind ourselves that this is a different sort of application than before, type arguments must be enclosed by brackets.

The syntax of types must be augmented with the notions of *type-variables*, and type expressions denoting the type of polymorphic functions.

```
<type> ::=   ...
     |       <type-variable>
     |       ∀<type-variable>.<type>            polymorphic function type
```

• A *type-variable* ψ is defined as an element of an infinite collection of type variables. We will adopt the convention to denote type-variables by small Greek letters ψ,φ,ω,…
• Type expressions of the form ∀ψ.Φ are introduced to denote the type of type-abstractions of the form Λψ.E, where E has type Φ. Note that ∀ is a binding operator (called *universal quantification*). The variable ψ is bound in ∀ψ.Φ.

The set of contexts does not change.

5.3. Inference and reduction rules

5.3.1. Typing

The typing rules of simple typed lambda calculus are extended with some typing rules for polymorphic function types:

$$\frac{\Gamma \vdash E \in \forall\psi.\Phi}{\Gamma \vdash E[\Theta] \in [\Theta/\psi]\Phi} \qquad (\forall \text{ elimination})^{17}$$

$$\frac{\Gamma \vdash E \in \Phi}{\Gamma \vdash \Lambda\psi.E \in \forall\psi.\Phi} \qquad (\forall \text{ introduction})$$

For example, the type of Λψ.λx:ψ.x is ∀ψ.ψ→ψ, and the type of

```
(Λψ.λx:ψ.x)[Integer]
```

is the type obtained from

```
∀ψ.ψ→ψ
```

by substituting `Integer` for ψ in ψ→ψ, yielding

```
Integer→Integer
```

which indeed is the type of the identity function for integers.

5.3.2. Reduction rules

The axioms and inference rules for equations between second-order lambda terms are similar to the axioms and rules of ordinary typed lambda calculus. The main difference is that we tend to have two versions of each axiom or rule, one for ordinary function abstraction or application

---

[16] I.e. not containing any free type variables.
[17] [Θ/ψ]Φ is the result of replacing all occurences of the type variable ψ in Φ with Θ, renaming bound variables in Φ to avoid capture of free variables in Θ. It is defined analogously to the definition of substitution on page 5.

(see page 10), and another for type abstraction or application. For example, (type α) assures that type expressions differing only in the names of bound type variables are identified.

$$\frac{\Gamma \vdash E \in \Phi \;;\; \varphi \text{ not free in } E}{\Gamma \vdash \Lambda\psi.E = \Lambda\varphi.\{\varphi/\psi\}E \in \forall\psi.\Phi} \qquad \text{(type } \alpha\text{)}$$

And (type β) and (type η) are defined as follows:

$$\frac{\Gamma \vdash E \in \Theta}{\Gamma \vdash (\Lambda\psi.E)[\Phi] = [\Phi/\psi]E \in [\Phi/\psi]\Theta} \qquad \text{(type } \beta\text{)}$$

$$\frac{\Gamma \vdash E \in \forall\psi.\Phi \;;\; \psi \text{ not free in } E}{\Gamma \vdash \Lambda\psi.E[\psi] = E \in \forall\psi.\Phi} \qquad \text{(type } \eta\text{)}$$

5.3.3. Equational theory

To the equational theory of the typed lambda calculus, two rules must be added. In the first place an equivalent of the (ξ) rule is needed for type abstraction.

$$\frac{\Gamma \vdash A = B \in \Phi}{\Gamma \vdash \Lambda\omega.A = \Lambda\omega.B \in \forall\omega.\Phi} \qquad \text{(type } \xi\text{)}$$

Secondly, we need a congruence rule for application of type expressions.

$$\frac{\Gamma \vdash A = B \in \forall\omega.\Phi}{\Gamma \vdash A[\Psi] = B[\Psi] \in [\Psi/\omega]\Phi} \qquad \text{(type congruence)}$$

5.4. Theoretical properties

Although some of the proofs (see [GiLT90]) become significantly more difficult, second-order polymorphic typed lambda calculus has essentially the same theoretical properties as ordinary typed lambda calculus. In particular, all terms of second-order polymorphic lambda calculus are strongly normalisable (i.e. they reach a normal form after a finite number of steps) and the normal form is unique (this uniqueness comes from an extension to the Church-Rosser property).

Because system F_2 is strongly normalising, recursion is not available, since recursion can be used to express nonterminating computations. However, the more restricted mechanism of primitive recursion may be formulated in the calculus. The class of functions definable in F_2 is even much larger than the primitive recursive functions. In particular, [PiDM89] shows how Ackermann's function which echibits surprisingly explosive growth and is not primitive recursive can be encoded in F_2. In fact, Girard [Gira72] and [GiLT90] show that every function that is representable in system F_2 is provable total under second-order Peano arithmetic, and vice versa that every function which is provable total is representable in F_2. Ackermann's function is an example of such a function.

[PiDM89] also illustrates how every set of inductively defined types can be translated into a set of types in the second-order polymorphic typed lambda calculus.

6. The systems F_n of polymorphic typed lambda calculus

6.1. Motivation

In typed lambda calculus, types were only used to describe the functionality of terms. Therefore types are not first class, because they can only occur within terms. System F_2 of

polymorphic typed lambda calculus allows the construction of polymorphic functions that take a type as argument and return a term. However, types are still not first class, since it is not possible to write functions from types to types. Therefore, we want to find an extension of system F_2 in which it is possible to write type constructors that take a type as argument and return a new type. This is system F_3. We can even go one step further, and construct a system F_4 in which a more general kind of type constructors is allowed: type constructors that take a type function (of the kind used in system F_3) as argument and return a type expression of system F_3. Analogously, in system F_5 type constructors are allowed that take a type expression of F_4 as input and return a type of F_4, and so on. In this way, we obtain a series of systems F_n of increasing power. The union of all these languages is called F_ω. In F_ω all kinds of type constructors are allowed.

6.2. Construction of the systems F_n

Now let us take a closer look at how the systems F_n are constructed.

• System F_1 corresponds to Church's ordinary typed lambda calculus.

• F_2 is Girard's and Reynolds' system of second-order typed lambda calculus.

• The main new feature of F_3 is its ability to express functions from types to types. We will denote the abstraction operator[18] on types by the symbol Λ, and application of a type Φ on a type Ψ is written $\Phi\ \Psi$. As with ordinary terms, we take the convention that this application is left-associative. Let us illustrate all this by means of an example.

In system F_2 it was possible to construct a polymorphic identity function

$$\mathtt{I} = \Lambda\psi.\lambda x{:}\psi.x$$

Instantiating this polymorphic identity `I` to a type α, yields

$$\mathtt{I}[\alpha] = \lambda x{:}\alpha.x$$

The polymorphic function `I` has a polymorphic type

$$\forall\psi.\psi\rightarrow\psi$$

Now, in system F_3, it is possible to express the type of `I` more specifically using the mapping

$$\Lambda\psi.\psi\rightarrow\psi$$

from types to types. Given a (type-) argument α, we can compute the type of `I[`α`]` by applying this (type-)function $\Lambda\psi.\psi\rightarrow\psi$ to α.

In general, if `E` is a polymorphic function of type $\forall\psi.\Phi$ then the type of the application `E[`α`]` is

$$(\Lambda\psi.\Phi)\,\alpha$$

The essential difference between $\forall\psi.\Phi$ and $\Lambda\psi.\Phi$ is that $\Lambda\psi.\Phi$ is a function from types to types, whereas $\forall\psi.\Phi$ is a type of a polymorphic function - that is, the type of a function that takes a type argument and returns a term. Type expressions of the form $\Lambda\psi.\Phi$ do not correspond to any terms at all: before they may be the type of a term, they must be applied to enough arguments to produce a "real" type.

To keep the syntax of constructor expressions of the form $\Lambda\psi.\Phi$ straight, we have to introduce a notion of *kinds*. Just as we needed types to make sure that terms involving abstraction and application were well-formed, we now need some notion of the "types" of type-expressions. We call the types of types *kinds*. There is a constant kind with name $\star$, which is the kind of types of terms. I.e. each well-typed term `E` of system F_2 has a type Φ and the kind of Φ is $\star$. Furthermore, if $\mathcal{K}$ is a kind, then so is $\star\Rightarrow\mathcal{K}$. For example a unary type function that takes a type and returns a new type has kind $\star\Rightarrow\star$, a binary type function has kind $\star\Rightarrow\star\Rightarrow\star$, and so on.

Note that the examples presented above are not entirely correct because they do not associate a kind to each binding of a type-variable in a term or type-expression. The exact syntax of F_3 is given by the following inductive definitions (see [PiDM89]):

---

[18] For type functions we use the same symbol Λ as for polymorphic functions. However, although both expect a type as argument, there can be no confusion between them, because a polymorphic function returns a term, whereas a type function returns a type. Furthermore, a polymorphic function is a term, while a type function is a type expression.

```
<kind> ::=   *
       |     *⇒<kind>

<type> ::=   <type-constant>
       |     <type-variable>
       |     <type>→<type>                          function type
       |     (<type>)
       |     ∀<type-variable>:<kind>.<type>         polymorphic function type
       |     Λ<type-variable>:<kind>.<type>         constructor
       |     <type> <type>                          constructor application

<term> ::=   <variable>
       |     <constant>
       |     λ<variable>:<type>.<term>                abstraction
       |     <term> <term>                            application
       |     (<term>)
       |     Λ<type-variable>:<kind>.<term>           type abstraction
       |     <term>[<type>]                           type application
```

Thus type constructors in F_3 have the form

$\Lambda\omega{:}\mathcal{K}.\Phi$

or more generally

$\Lambda\omega_1{:}\mathcal{K}_1.\ldots\Lambda\omega_n{:}\mathcal{K}_n.\Phi$

which build a new type Φ out of a number n of given types ω_i.

On page 8, we introduced *contexts* because it was necessary to assign a type to all free (ordinary) variables in terms. Now we have the same problem with type-variables and type expressions. Indeed, consider for example the type expression

$\Lambda\omega{:}\mathcal{K}.\varphi\ \omega$

Before we can assign a kind to this type expression, we must know the kind of the free type-variable φ. To put it in another way, the kind of this type expression will depend on the context in which it occurs. Hence the definition of contexts has to be enriched with associations of kinds to type-variables (in addition to associations of types to variables), and in analogy with typings, we have to introduce a notion of *kindings*. We will not present all details here, but will come back to these issues when introducing system F_ω.

• The only difference between F_3 and F_4 is in the kinds that may appear in expressions. Whereas F_3 uses only kinds such as $*$ and $*\Rightarrow*$, F_4 also allows the kind $(*\Rightarrow*)\Rightarrow*$, the kind of type constructors that take type functions as arguments.

• By adding quantification over constructors of successively higher kinds, we obtain the languages F_5 , F_6 ,...

• The union of all these languages is F_ω. This system will be explained in the next section

7. System F_ω of polymorphic typed lambda calculus

7.1. Syntax

System F_ω differs from F_3, F_4, etc. only in that the set of legal kinds is larger. We already mentioned that kinds can be used to describe the functionality of subexpressions of type expressions. Essentially, kinds are the "types" of things that can appear in type expressions. Subexpressions of type expressions may denote ordinary types (the types as defined in the syntax of system F_2), functions from types to types, functions from type functions to types, and so on.

We will use

$*$ to denote the kind consisting of all (ordinary) types,

$*\Rightarrow*$ to denote the kind of functions from ordinary types to ordinary types,

`(*⇒*)⇒*` to denote the kind of functions from type functions to types,
`*⇒(*⇒*)` to denote the kind of functions from types to type functions,
and so on.

For example `Λψ:*.ψ→ψ` is a type constructor of kind `*⇒*`, because it translates an ordinary type ψ (of kind `*`) in an ordinary type ψ→ψ. Notice how we use ⇒ instead of → to denote the kind of functions on type expressions, to reduce the confusion between types and kinds.

Formally, the set of kind expressions is given by the abstract grammar:

```
<kind> ::=   *
       |     <kind>⇒<kind>
       |     (<kind>)
```

Intuitively, the kind $\mathcal{K}⇒\mathcal{L}$ is the kind of functions from type expressions of kind $\mathcal{K}$ to type expressions of kind $\mathcal{L}$. Sometimes it is necessary to use parentheses, for example to make a distinction between

`*⇒(*⇒*)`

and

`(*⇒*)⇒*`

The former takes a type as argument and yields a function from types to types, while the latter expects a function from types to types as argument and returns a type. We will adopt the convention to drop unnecessary parentheses, and to use round letters $\mathcal{K},\mathcal{L},\mathcal{M},...$ for arbitrary kind expressions. As for →, we assume that the symbol ⇒ assoicates to the right.

The set of terms and types is exactly the same as for system F_3. When discussing F_3 we also remarked that it was necessary to extend the definition of contexts with associations of kinds to type variables:

```
<context> ::=  ...
          |    <context>,<type-variable>:<kind>    type variable binding
```

We will use the notation `Γ(ω)` to denote the kind associated to the type variable ω in the context Γ, and `Γ,ω:`$\mathcal{K}$ to denote a new context resulting from the extension of Γ with a new association ω∈$\mathcal{K}$.

7.2. Inference and reduction rules

7.2.1. Typing

The typing rules for F_ω are the same as those of F_2 (see pages 9 and 13) except that the rules `(∀ elimination)` and `(∀ introduction)` of page 13 need some changes to conform to the current syntax.

$$\frac{\Gamma \vdash E \in \forall\psi{:}\mathcal{K}.\Phi \;;\; \Gamma \vdash \Theta \in \mathcal{K}}{\Gamma \vdash E[\Theta] \in [\Theta/\psi]\Phi} \qquad (\forall \text{ elimination})$$

$$\frac{\Gamma,\psi{:}\mathcal{K} \vdash E \in \Phi \;;\; \psi \text{ not free in } \Gamma}{\Gamma \vdash \Lambda\psi{:}\mathcal{K}.E \in \forall\psi.\Phi} \qquad (\forall \text{ introduction})$$

We take "ψ not free in Γ" to mean that ψ is not free in any type expression assigned by Γ.

7.2.2. Kinding

On page 9 we defined typings of the form

`Γ ⊢ E ∈ Φ`

asserting that E has type Φ with respect to Γ. Because we have already adapted the definition of contexts to allow kind assignment to type variables, we can use the same notation for kindings. Intuitively, a *kinding* is a formula of the form

$$\Gamma \vdash \Phi \in \mathcal{K}$$

that asserts that Φ takes on kind $\mathcal{K}$ when its free type variables are assigned kinds by Γ. Formally, we just add the following rules to the inference rules for typings.

Axioms about the kinding relation are:

$\vdash \alpha \in \mathcal{K}$ iff α type-constant of kind $\mathcal{K}$ (type-constant)

$\Gamma \vdash \omega \in \Gamma(\omega)$ for each type-variable ω in Γ (type-variable)

The kind derivation rules are completely equivalent to the type derivation rules (→ elimination) and (→ introduction). (⇒ elimination) states that if a constructor Φ has kind $\mathcal{K}\Rightarrow\mathcal{L}$ and a type-expression Ψ has kind $\mathcal{K}$ then the constructor application Φ Ψ has kind $\mathcal{L}$.

$$\frac{\Gamma \vdash \Phi \in \mathcal{K}\Rightarrow\mathcal{L} \;;\; \Gamma \vdash \Psi \in \mathcal{K}}{\Gamma \vdash \Phi\ \Psi \in \mathcal{L}} \qquad (\Rightarrow \text{ elimination})$$

The dual kind derivation rule (⇒ introduction) shows how the kind of a constructor can be derived:

$$\frac{\Gamma, \omega{:}\mathcal{K} \vdash \Phi \in \mathcal{L}}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi \in \mathcal{K}\Rightarrow\mathcal{L}} \qquad (\Rightarrow \text{ introduction})$$

The following rule assures that all types of simple typed lambda calculus (system F_1) are of kind ∗. Indeed, only constructors are of type $\mathcal{K}\Rightarrow\mathcal{L}$ (where $\mathcal{K}$ and $\mathcal{L}$ are kinds), and in ordinary typed lambda calculus no constructors are allowed.

$$\frac{\Gamma \vdash \Phi \in *,\ \Gamma \vdash \Psi \in *}{\Gamma \vdash \Phi\rightarrow\Psi \in *} \qquad (\rightarrow)$$

An additional rule (∀) can be given to assure that all types occurring in F_2 are of kind ∗.

$$\frac{\Gamma, \omega{:}\mathcal{K} \vdash \Psi \in *}{\Gamma \vdash \forall\omega{:}\mathcal{K}.\Psi \in *} \qquad (\forall)$$

In other words, the combination of rules (∀) and (→) tells us that by using only the type constructors ∀ and →, no new kinds of types can be built. This is indeed the situation of system F_2, where we only had those two constructors, and where we had only one kind of types. (All types were assumed of the same kind, since the idea of kinds was only introduced later, in system F_3.)

7.2.3. Reduction rules

The axioms and inference rules are essentially the familiar rules of second-order polymorphic typed lambda calculus. However, in the current system some reduction rules for constructors are needed.

$$\frac{\Gamma \vdash \Psi \in \mathcal{L} \;;\; \varphi \text{ not free in } \Psi \;;\; \varphi \text{ not in } \Gamma}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Psi = \Lambda\varphi{:}\mathcal{K}.\{\varphi/\omega\}\Psi \in \mathcal{K}\Rightarrow\mathcal{L}} \quad \text{(constructor } \alpha\text{)}$$

$$\frac{\Gamma \vdash \Phi \in \mathcal{K} \;;\; \Gamma,\omega{:}\mathcal{K} \vdash \Psi \in \mathcal{L}}{\Gamma \vdash (\Lambda\omega{:}\mathcal{K}.\Psi)\Phi = [\Phi/\omega]\Psi \in \mathcal{L}} \quad \text{(constructor } \beta\text{)}$$

$$\frac{\Gamma \vdash \Phi \in \mathcal{L}\Rightarrow\mathcal{M} \;;\; \omega \text{ not free in } \Phi}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi\ \omega = \Phi \in \mathcal{L}\Rightarrow\mathcal{M}} \quad \text{(constructor } \eta\text{)}$$

7.2.4. Equational theory

The inference rules (symmetry), (transitivity), (congruence), (ξ) and (type ξ) are the same as for system F_2, but to conform to the current syntax, we have to rewrite the rule (type congruence).

$$\frac{\Gamma \vdash A=B \in \forall\omega{:}\mathcal{K}.\Phi \;;\; \Gamma \vdash \Psi_1=\Psi_2 \in \mathcal{K}}{\Gamma \vdash A[\Psi_1] = B[\Psi_2] \in [\Psi_1/\omega]\Phi} \quad \text{(type congruence)}$$

The only additional rules we have to introduce are a congruence rule and a ξ-rule for constructors:

$$\frac{\Gamma,\omega{:}\mathcal{K} \vdash \Phi = \Psi \in \mathcal{L}}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi = \Lambda\omega{:}\mathcal{K}.\Psi \in \mathcal{K}\Rightarrow\mathcal{L}} \quad \text{(constructor } \xi\text{)}$$

$$\frac{\Gamma \vdash \Phi_1 = \Phi_2 \in \mathcal{K}\Rightarrow\mathcal{L} \;;\; \Gamma \vdash \Psi_1 = \Psi_2 \in \mathcal{K}}{\Gamma \vdash \Phi_1\ \Psi_1 = \Phi_2\ \Psi_2 \in \mathcal{L}} \quad \text{(constructor congruence)}$$

7.3. Theoretical properties

In the first place, [PiDM89] shows how the languages F_1, F_2, etc. can be defined as restrictions of F_ω. To do this, the *order* of a kind is defined. Kind $*$ has order 1, and the order of kind $\mathcal{K}\Rightarrow\mathcal{L}$ is defined as 1+max(k,l) where k is the order of kind $\mathcal{K}$ and l is the order of kind $\mathcal{L}$. Then the nth-order polymorphic lambda calculus F_n consists only of those terms of F_ω for which types can be derived using the given rules without mentioning any kinds greater than or equal to n.

[PiDM89] also states the following theorems for F_ω:

- If $\Gamma \vdash \Phi \in \mathcal{K}$ then Φ has a normal form, which is unique up to α-equivalence.
- If $\Gamma \vdash E \in \Phi$ then E has a normal form, which is unique up to α-equivalence.
- $\Gamma \vdash E \in \Phi$ is decidable.

8. System $F_\omega^{\leq}$ of polymorphic typed lambda calculus

8.1. Extending system F_ω with subtyping

$F_\omega^{\leq}$ is an extension of system F_ω that is sometimes used as a basis for formal models of object oriented systems. For example [HoPi92], [HoPi94], [PiTu92], [Pier93], [Mitc92] and others

use $F_\omega^\leq$ to describe various aspects of object orientedness. The essential difference between F_ω and $F_\omega^\leq$ is that in $F_\omega^\leq$ the notion of *subtyping* is added. Intuitively a type is said to be a subtype of another if an expression of that type can be used in any place where a variable of the other type is required. We will use the notation $\Phi \leq \Psi$ to denote "type Φ is a subtype of type Ψ".

To obtain a complete description of system $F_\omega^\leq$ we will not only show the changes that have to be made to the syntax and rules of F_ω, but we will also repeat all the other rules and syntactic constructs. In order to clearly see the innovations made to system F_ω, all changes or extensions have been printed in **`bold`**.

Before continuing, we present the simple example of integer subrange types. Let `n...m` denote the subtype of the type `Integer` associated with the subrange from n to m, where n and m are known integers. The following subtyping relations hold for integer subrange types:

`n...m` ≤ `n'...m'` iff n'≤n and m≤m'

where the ≤ on the left denotes the subtyping relation and those on the right denote "less than or equal to". Subrange types may occur as type specifications in lambda expressions:

```
f = λx:2...5.x+1
```

This expression has type `2...5→3...6`.
Furthermore `f(3)` is a legal expression because the `Integer` constant `3` has the type `3...3` and also has the type of any supertype, including the type `2...5` of `x` above. Similarly the following should be legal:

```
g = λy:3...4.f(y)
```

because the type of `y` is a subtype of the domain of `f`.

8.2. Syntax

The set of terms is defined by the following abstract grammar:

```
<term> ::=   <constant>
       |     <variable>
       |     λ<variable>:<type>.<term>              abstraction
       |     <term> <term>                          application
       |     (<term>)
       |     Λ<type-variable>≤<type>.<term>         type abstraction
       |     <term>[<type>]                         type application
```

The only essential difference to system F_ω is that type abstraction is of the form

```
Λ<type-variable>≤<type>.<term>
```

instead of

```
Λ<type-variable>.<term>
```

Hence quantified type variables are restricted by specifying an upper bound for the type variable. To put it in other words, the type parameter of a second-order expression is not allowed to range over the universe of all types, but only over a restricted subset of types. This is what we call *bounded quantification*.

Types are defined by:

```
<type> ::=   <type-constant>
       |     <type-variable>
       |     <type>→<type>                          function type
       |     (<type>)
       |     ∀<type-variable>≤<type>.<type>         polymorphic function type
       |     Λ<type-variable>:<kind>.<type>         constructor
       |     <type> <type>                          constructor application
       |     TOP(<kind>)                            top-type
```

In the syntax of types, bounded quantification is also introduced by decorating occurrences of type variables in quantifiers with subtyping assumptions (i.e. an upper bound is specified). However, to keep the kind structure as simple as possible, constructors retain the form

$\Lambda\omega{:}\mathcal{K}.\Phi$

rather than changing to

$\Lambda\omega{\leq}\Psi.\Phi$

Furthermore a new sort of types is introduced, namely the *top types*. Intuitively a top type of a certain kind is the type such that every other type of that kind is a subtype of the top type. In other words TOP($\mathcal{K}$) is the maximal element of kind $\mathcal{K}$, according to $\leq$. (Indeed, there will be a rule expressing the fact that for every type Φ of kind $\mathcal{K}$, Φ is a subtype of `TOP(`$\mathcal{K}$`)`.)

The set of contexts is defined as follows:

`<context>::=` ∅		*empty context*
|	`<context>,<variable>:<type>`	*variable binding*
|	**`<context>,<type-variable>≤<type>`**	***type variable binding with bound***

In contexts, assumptions about type variables have the form $\Gamma,\omega{\leq}\Psi$ instead of $\Gamma,\omega{:}\mathcal{K}$.
Intuitively $\Gamma,\omega{\leq}\Psi$ means that ω is a type variable that can be used as a type variable of any type Φ, as long as $\Phi{\leq}\Psi$. As before, $\Gamma,x{:}\Phi$ denotes the extension of a context Γ with a new variable binding $x{\in}\Phi$. In the current notation, $\Gamma(x)$ still denotes the type associated to the ordinary variable `x` in the context Γ, but $\Gamma(\omega)$ denotes the *upper bound* associated to the type variable ω in the context Γ. It is important to note that now $\Gamma(\omega)$ is a *type* whereas in system F_ω, $\Gamma(\omega)$ was the *kind* associated with ω in context Γ.

As before, the set of kinds is:

```
<kind> ::= *
      |    <kind>⇒<kind>
      |    (<kind>)
```

We repeat the notion of typings and kindings:
$\Gamma \vdash E \in \Phi$ is a *typing* that asserts that `E` has type Φ with respect to Γ.
$\Gamma \vdash \Phi \in \mathcal{K}$ is a *kinding* that asserts that Φ takes on kind $\mathcal{K}$ with respect to Γ.
In system $F^{\leq}_{\omega}$, we also need *subtypings* of the form $\Gamma \vdash \Phi{\leq}\Psi$ which is read as "Φ is a subtype of Ψ with respect of Γ". The intuitive meaning of $\Phi{\leq}\Psi$ is that any expression of type Φ is allowed in every place where an expression of type Ψ is required.

8.3. Inference and reduction rules

8.3.1. Typing

Little has to be changed to the typing rules:

$\vdash c \in \Phi$ iff c constant of type Φ (constant)

$\Gamma \vdash x \in \Gamma(x)$ for every variable x in Γ (variable)

$$\frac{\Gamma \vdash F \in \Phi\rightarrow\Psi \;;\; \Gamma \vdash A \in \Phi}{\Gamma \vdash F\ A \in \Psi}$$ ($\rightarrow$ elimination)

$$\frac{\Gamma, x{:}\Phi \vdash E \in \Psi}{\Gamma \vdash \lambda x{:}\Phi.E \in \Phi \to \Psi} \qquad (\to \text{ introduction})$$

Rules (∀ elimination) and (∀ introduction) for polymorphic function types will be slightly different, because their syntax has changed a little.

$$\frac{\Gamma \vdash \mathbf{E} \in \forall \psi \leq \Phi.\Psi \;;\; \Gamma \vdash \Theta \in \mathcal{K} \;;\; \Gamma \vdash \Theta \leq \Phi}{\Gamma \vdash \mathbf{E}[\Theta] \in [\Theta/\psi]\Psi} \qquad \textbf{(}\forall \textbf{ elimination)}$$

$$\frac{\Gamma, \psi \leq \Phi \vdash \mathbf{E} \in \Psi \;;\; \psi \text{ not free in } \Gamma}{\Gamma \vdash \Lambda \psi \leq \Phi.\mathbf{E} \in \forall \psi \leq \Phi.\Psi} \qquad \textbf{(}\forall \textbf{ introduction)}$$

The only real innovation in the typing rules is a subsumption rule formally expressing the intuitive idea of subtyping: if E is of type Φ, and Φ is a subtype of Ψ, then E can also be seen as an expression of type Ψ. In other words, E can be used in every place where an expression of type Ψ is required.

$$\frac{\Gamma \vdash \mathbf{E} \in \Phi \;;\; \Gamma \vdash \Phi \leq \Psi \;;\; \Gamma \vdash \Psi \in \mathcal{K}}{\Gamma \vdash \mathbf{E} \in \Psi} \qquad \textbf{(subsumption)}$$

8.3.2. Kinding

The following kinding schemes are the same as before:

$$\vdash \alpha \in \mathcal{K} \quad \text{iff } \alpha \text{ type-constant of kind } \mathcal{K} \qquad \text{(type-constant)}$$

$$\frac{\Gamma \vdash \Phi \in \mathcal{K} \Rightarrow \mathcal{L} \;;\; \Gamma \vdash \Psi \in \mathcal{K}}{\Gamma \vdash \Phi\ \Psi \in \mathcal{L}} \qquad (\Rightarrow \text{ elimination})$$

$$\frac{\Gamma \vdash \Phi \in * \;;\; \Gamma \vdash \Psi \in *}{\Gamma \vdash \Phi \to \Psi \in *} \qquad (\to)$$

Again, because the syntax of polymorphic function types has not changed very much. Only rule (∀) is slightly different.

$$\frac{\Gamma, \omega \leq \Phi \vdash \Psi \in *}{\Gamma \vdash \forall \omega \leq \Phi.\Psi \in *} \qquad \textbf{(}\forall\textbf{)}$$

The typing rule for type variables however, is entirely different in $F_\omega^{\leq}$. In F_ω the rule was

$$\Gamma \vdash \omega \in \Gamma(\omega) \text{ for each type-variable } \omega \text{ in } \Gamma$$

Indeed $\Gamma(\omega)$ was the kind associated to the variable ω in context Γ. This rule does not make any sense in $F_\omega^{\leq}$, since type variables are not bound to kinds, but to types denoting the upper bound of the type variable. That is why the kinding rule for type variables becomes:

$$\frac{\Gamma \vdash \Gamma(\omega) \in \mathcal{K}}{\Gamma \vdash \omega \in \mathcal{K}} \qquad \textbf{(type-variable)}$$

In other words: if the upper bound associated to the type variable ω in context Γ is of kind $\mathcal{K}$, then ω itself must be of kind $\mathcal{K}$.

(⇒ introduction) is a way of saying that ω≤TOP($\mathcal{K}$) is essentially the same as ω:$\mathcal{K}$. This corresponds to our intuition.

$$\frac{\Gamma, \omega \leq \mathbf{TOP}(\mathcal{K}) \vdash \Phi \in \mathcal{L}}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi \in \mathcal{K} \Rightarrow \mathcal{L}} \quad \textbf{(⇒ introduction)}$$

Of course, we also need a kinding rule for top types.

$$\Gamma \vdash \mathbf{TOP}(\mathcal{K}) \in \mathcal{K} \quad \textbf{(TOP)}$$

(TOP) tells us that the kind of a top type of kind $\mathcal{K}$ is indeed $\mathcal{K}$.

8.3.3. Subtyping

In this paragraph, we will construct a rule-set to formally define how the subtyping relation is supposed to work. We will begin with a number of rules that correspond immediately with our intuition.

$$\frac{\Gamma \vdash \Phi \leq \Psi_1 \;;\; \Gamma \vdash \Psi_1 = \Psi_2 \in \mathcal{K}}{\Gamma \vdash \Phi \leq \Psi_2} \quad \textbf{(subtype conversion)}$$

$$\Gamma \vdash \Phi \leq \Phi \quad \textbf{(subtype reflexivity)}$$

$$\frac{\Gamma \vdash \Phi \leq \Psi \;;\; \Gamma \vdash \Psi \in \mathcal{K} \;;\; \Gamma \vdash \Psi \leq \Theta}{\Gamma \vdash \Phi \leq \Theta} \quad \textbf{(subtype transitivity)}$$

The next rule is a "pointwise subtyping" rule[19] for operators.
Intuitively, $\Lambda\omega{:}\mathcal{K}.\Phi$ is a subtype of $\Lambda\omega{:}\mathcal{K}.\Psi$ if and only if $[\Theta/\omega]\Phi$ is a subtype of $[\Theta/\omega]\Psi$ for every Θ of kind $\mathcal{K}$.

$$\frac{\Gamma, \omega \leq \mathbf{TOP}(\mathcal{K}) \vdash \Phi \leq \Psi}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi \leq \Lambda\omega{:}\mathcal{K}.\Psi} \quad \textbf{(subtype ξ)}$$

Because subtyping of operators is pointwise, we may promote the operator in a type application to any larger operator "in place":

$$\frac{\Gamma \vdash \Phi \leq \Psi}{\Gamma \vdash \Phi\ \Theta \leq \Psi\ \Theta} \quad \textbf{(subtype congruence)}$$

We already said Γ(ω) would be used to denote the upper bound associated to the type variable ω in the context Γ. This is formalised by following axiom:

$$\Gamma \vdash \omega \leq \Gamma(\omega) \quad \textbf{(subtype type-variable)}$$

---

[19] There also exist more sophisticated possibilities for operator subtyping than pointwise. For example [Card90] suggests a more powerful treatment of operator subtyping, including both monotonic and antimonotonic subtyping in addition to pointwise subtyping.

We also announced (page 21) that we would need a rule to express the fact that every type of kind $\mathcal{K}$ is a subtype of TOP($\mathcal{K}$).

$$\frac{\Gamma \vdash \Phi \in \mathcal{K}}{\Gamma \vdash \Phi \leq \mathbf{TOP}(\mathcal{K})} \qquad \textbf{(subtype TOP)}$$

The following rules (subtype $\rightarrow$) and (subtype $\forall$) describe the influence of the constructors $\rightarrow$ and $\forall$ on the subtype relation.

$$\frac{\Gamma, \omega \leq \Psi \vdash \Phi_1 \leq \Phi_2}{\Gamma \vdash \forall \omega \leq \Psi . \Phi_1 \leq \forall \omega \leq \Psi . \Phi_2} \qquad \textbf{(subtype } \forall\textbf{)}$$

$$\frac{\Gamma \vdash \Psi_1 \leq \Phi_1 \ ; \ \Gamma \vdash \Phi_2 \leq \Psi_2}{\Gamma \vdash \Phi_1 \rightarrow \Phi_2 \leq \Psi_1 \rightarrow \Psi_2} \qquad \textbf{(subtype } \rightarrow\textbf{)}$$

When you take a closer look at (subtype $\rightarrow$) you will see that it seems to be counter-intuitive. However, if you would follow the "intuitive" approach, you would get contradictions because of the *contravariance problem*. Let us illustrate this on the basis of an example. Consider a function of type $3 \ldots 7 \rightarrow 7 \ldots 9$. This can also be considered a function of type $4 \ldots 6 \rightarrow 6 \ldots 10$, since it maps integers between 3 and 7 (and hence between 4 and 6) to integers between 7 and 9 (and hence between 6 and 10). Note that the domain shrinks while the codomain expands.

8.3.4. Reduction rules

The reduction rules for ordinary expressions, types and constructors are essentially the same as before.

$$\frac{\Gamma, x{:}\Phi \vdash A \in \Psi \ ; \ y \text{ not free in } A \ ; \ y \text{ not in } \Gamma}{\Gamma \vdash \lambda x{:}\Phi . A = \lambda y{:}\Phi . \{y/x\}A \in \Phi \rightarrow \Psi} \qquad (\alpha)$$

$$\frac{\Gamma \vdash B \in \Phi \ ; \ \Gamma, x{:}\Phi \vdash A \in \Psi}{\Gamma \vdash (\lambda x{:}\Phi . A)B = [B/x]A \in \Psi} \qquad (\beta)$$

$$\frac{\Gamma \vdash F \in \Phi \rightarrow \Psi \ ; \ x \text{ not free in } F}{\Gamma \vdash \lambda x{:}\Phi . F\ x = F \in \Phi \rightarrow \Psi} \qquad (\eta)$$

$$\frac{\Gamma, \psi \leq \Psi \vdash \mathbf{E} \in \Phi \ ; \ \varphi \textbf{ not free in } \mathbf{E} \ ; \ \varphi \textbf{ not in } \Gamma}{\Gamma \vdash \Lambda \psi \leq \Psi . \mathbf{E} = \Lambda \varphi \leq \Psi . \{\varphi/\psi\}\mathbf{E} \in \forall \psi . \Phi} \qquad \textbf{(type } \alpha\textbf{)}$$

$$\frac{\Gamma, \psi \leq \Psi \vdash \mathbf{E} \in \Theta \ ; \ \Gamma \vdash \Phi \leq \Psi}{\Gamma \vdash (\Lambda \psi \leq \Psi . \mathbf{E})[\Phi] = [\Phi/\psi]\mathbf{E} \in [\Phi/\psi]\Theta} \qquad \textbf{(type } \beta\textbf{)}$$

$$\frac{\Gamma \vdash \mathbf{E} \in \forall \psi \leq \Psi . \Phi \ ; \ \psi \textbf{ not free in } \mathbf{E}}{\Gamma \vdash \Lambda \psi \leq \Psi . \mathbf{E}[\psi] = \mathbf{E} \in \forall \psi \leq \Psi . \Phi} \qquad \textbf{(type } \eta\textbf{)}$$

$$\frac{\Gamma,\omega{:}\mathcal{K} \vdash \Psi \in \mathcal{L} \;;\; \varphi \text{ not free in } \Psi \;;\; \varphi \text{ not in } \Gamma}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Psi = \Lambda\varphi{:}\mathcal{K}.\{\varphi/\omega\}\Psi \in \mathcal{K}\Rightarrow\mathcal{L}} \quad \textbf{(constructor } \alpha\textbf{)}$$

$$\frac{\Gamma \vdash \Phi \in \mathcal{K} \;;\; \Gamma,\omega{:}\mathcal{K} \vdash \Psi \in \mathcal{L}}{\Gamma \vdash (\Lambda\omega{:}\mathcal{K}.\Psi)\Phi = [\Phi/\omega]\Psi \in \mathcal{L}} \quad \text{(constructor } \beta\text{)}$$

$$\frac{\Gamma \vdash \Phi \in \mathcal{L}\Rightarrow\mathcal{M} \;;\; \omega \text{ not free in } \Phi}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi\ \omega = \Phi \in \mathcal{L}\Rightarrow\mathcal{M}} \quad \text{(constructor } \eta\text{)}$$

$$\frac{\Gamma \vdash \Phi \in \mathcal{K}}{\Gamma \vdash \mathbf{TOP}(\mathcal{K}\Rightarrow\mathcal{L})\Phi = \mathbf{TOP}(\mathcal{L})}$$

8.3.5. Equational theory

Although developing a full-fledged equational theory for $F^{\leq}_{\omega}$ remains a matter for future research, [HoPi94] tentatively proposes the following equational theory. To reduce clutter, they drop the evident well-kindedness premises; these can be filled in by analogy with the typing rules of page 21.

To start with, we repeat the familiar rules of equational theory:

$$\frac{\Gamma \vdash A = B \in \Phi}{\Gamma \vdash B = A \in \Phi} \quad \text{(symmetry)}$$

$$\frac{\Gamma \vdash A = B \in \Phi \;;\; \Gamma \vdash B = C \in \Phi}{\Gamma \vdash A = C \in \Phi} \quad \text{(transitivity)}$$

$$\frac{\Gamma \vdash F = G \in \Phi\to\Psi \;;\; \Gamma \vdash A = B \in \Phi}{\Gamma \vdash F\ A = G\ B \in \Psi} \quad \text{(congruence)}$$

$$\frac{\Gamma \vdash \Phi_1 = \Phi_2 \in \mathcal{K}\Rightarrow\mathcal{L} \;;\; \Gamma \vdash \Psi_1 = \Psi_2 \in \mathcal{K}}{\Gamma \vdash \Phi_1(\Psi_1) = \Phi_2(\Psi_2) \in \mathcal{L}} \quad \text{(constructor congruence)}$$

In the current system, the rules (ξ), (type ξ) and (constructor ξ) are:

$$\frac{\Gamma \vdash \Phi_2 \leq \Phi_1 \;;\; \Gamma \vdash \Psi_1 \leq \Psi_2 \;;\; \Gamma,\mathbf{x}{:}\Phi_1 \vdash \mathbf{A}=\mathbf{B} \in \Psi_1}{\Gamma \vdash \lambda\mathbf{x}{:}\Phi_1.\mathbf{A} = \lambda\mathbf{x}{:}\Phi_2.\mathbf{B} \in \Phi_2\to\Psi_2} \quad \textbf{(}\xi\textbf{)}$$

$$\frac{\Gamma,\psi\leq\Phi \vdash \Psi_1 \leq \Psi_2 \;;\; \Gamma,\psi\leq\Phi \vdash \mathbf{A} = \mathbf{B} \in \Psi_1}{\Gamma \vdash \Lambda\psi\leq\Phi.\mathbf{A} = \Lambda\psi\leq\Phi.\mathbf{B} \in \forall\psi\leq\Phi.\Psi_2} \quad \textbf{(type } \xi\textbf{)}$$

$$\frac{\Gamma, \omega{:}\mathcal{K} \vdash \Phi = \Psi \in \mathcal{L}}{\Gamma \vdash \Lambda\omega{:}\mathcal{K}.\Phi = \Lambda\omega{:}\mathcal{K}.\Psi \in \mathcal{K}\Rightarrow\mathcal{L}} \quad \textbf{(constructor } \xi\textbf{)}$$

It might appear that the rule (type ξ) should be generalised, by analogy with (ξ), to allow the comparison of type abstractions with different upper bounds. The rule (type ξ) would then become something like

$$\frac{\Gamma \vdash \Phi_2 \le \Phi_1 \;;\; \Gamma, \psi \le \Phi_1 \vdash \Psi_1 \le \Psi_2 \;;\; \Gamma, \psi \le \Phi_1 \vdash A = B \in \Psi_1}{\Gamma \vdash \Lambda\psi \le \Phi_1.A = \Lambda\psi \le \Phi_2.B \in \forall\psi \le \Phi_2.\Psi_2} \quad \text{(type } \xi\text{)}$$

But then it would also be necessary to similarly generalise the subtyping rule (subtype $\forall$), leading to a richer system, but one with a much more difficult metatheory. See [StPi94] for a related discussion.

In the presence of a subtyping relation, the (type congruence) rule scheme becomes:

$$\frac{\Gamma \vdash \mathbf{A=B} \in \forall\psi \le \Psi.\Phi \;;\; \Gamma \vdash \Theta_1 \le \Psi \;;\; \Gamma \vdash \Theta_2 \le \Psi \;;\; \Gamma \vdash [\Theta_1/\psi]\Phi \le \Sigma \;;\; \Gamma \vdash [\Theta_2/\psi]\Phi \le \Sigma}{\Gamma \vdash \mathbf{A}[\Theta_1] = \mathbf{B}[\Theta_2] \in \Sigma} \quad \textbf{(type congruence)}$$

We also need an equivalent of the subsumption rule for equalities.

$$\frac{\Gamma \vdash \mathbf{A=B} \in \Phi \;;\; \Gamma \vdash \Phi \le \Psi}{\Gamma \vdash \mathbf{A=B} \in \Psi} \quad \textbf{(subsumption equality)}$$

And finally, to handle with top types in equations, we need the rule:

$$\frac{\Gamma \vdash \mathbf{A} \in \mathbf{TOP(*)} \;;\; \Gamma \vdash \mathbf{B} \in \mathbf{TOP(*)}}{\Gamma \vdash \mathbf{A=B} \in \mathbf{TOP(*)}} \quad \textbf{(TOP equality)}$$

8.4. Theoretical properties

The metatheory of pure $F^{\le}_{\omega}$ has been studied by Steffen and Pierce [StPi94]. In particular, they prove a property of strong normalisation of well-kinded types

if $\Gamma \vdash \Phi \in \mathcal{K}$ then Φ has a unique normal form under (type β) reduction

and provide a sound and complete algorithm for determining whether $\Gamma \vdash \Phi \le \Psi$, where Φ and Ψ are well-kinded.

9. Further extensions to system $F^{\le}_{\omega}$

In this section we will give a number of possible extensions that can still be made to system $F^{\le}_{\omega}$, and show how these extensions can be used to model important concepts of object oriented programming. More specifically, we will show how to introduce records (and how they can be used to model objects), existential quantification (and its use to model encapsulation) and recursive types (and how to build recursive data structures with it).

9.1. Records

9.1.1. Example

The description of system $F_\omega^\leq$ we gave in the previous section is not entirely complete, because we still have not introduced the notion of records.[20] A record R with fields $n_1, n_2, \ldots, n_k$ bound to values $v_1, v_2, \ldots, v_k$ respectively will be written

$R = \{n_1=v_1; n_2=v_2; \ldots; n_k=v_k\}$

and field selection[21] will be denoted by means of the symbol #, e.g.

$R\#n_2 = v_2$

Records are rather important in this system, because it is mainly used to formulate mathematical models for object oriented systems, and a lot of these models view objects as records consisting of state and methods. For example, a movable, one-dimensional point object could be encoded as:

```
p =    { state =    {x=5};
         methods =  {setX = λstate:{x:Integer}.λi:Integer.{x=i};
                     getX = λstate:{x:Integer}.state#x} }
```

where p is a record of type

```
Point =      { state : {x:Integer} ;
               methods :
                    {setX : {x:Integer}→(Integer→{x:Integer});
                     getX : {x:Integer}→Integer} }
```

To apply the method setX of p on the internal state of p and a new integer 4 we write:

```
(p#methods#setX(p#state))(4)
```

which results in a record {x=4} of type {x:Integer}.

9.1.2. Syntax and rules

Records can be very easily introduced in the syntax:
First of all, a record type is needed, as well as a way to construct records and to select a certain field of a record. So the set of types and terms must be changed as follows:

```
<type> ::=...
     |     {<name>:<type>;...;<name>:<type>}          record type

<terms> ::=  ...
     |       {<name>=<type>;...;<name>=<type>}        record construction
     |       <term>#<name>                           record selection
```

where we assume that a <name> is a sequence of characters, and that all names occurring in a record construction or in a record type are disjoint.

Now we will extend the rule-system to allow records.
The kinding rule for records is:

$$\frac{\vdash \Gamma \text{ context} \;;\; \text{for each } i : \Gamma \vdash \Phi_i \in *}{\Gamma \vdash \{l_1:\Phi_1; \ldots; l_n:\Phi_n\} \in *} \qquad \text{(record)}$$

---

[20] Almost all authors ([HoPi92], [HoPi94], [PiTu92], [Pier93], [Mitc92], [CaWe85], ...) include the notion of records in their description of polymorphic lambda calculus.

[21] Normally the dot notation <term>.<name> is used for selecting a field in a record, but we already used a dot notation for lambda expressions: λ<variable>.name. So in order to avoid any possible confusion, we will use a (non-standard) notation <term>#<name> for record selection.

The first typing rule for records is a straightforward introduction rule:

$$\frac{\vdash \Gamma \text{ context } ; \text{ for each } i : \Gamma \vdash E_i \in \Phi_i}{\Gamma \vdash \{l_1=E_1;\ldots;l_n=E_n\} \in \{l_1:\Phi_1;\ldots;l_n:\Phi_n\}} \quad \text{(record introduction)}$$

And instead of defining (record elimination) in the intuitive way:

$$\frac{\Gamma \vdash E \in \{l_1:\Phi_1;\ldots;l_n:\Phi_n\}}{\Gamma \vdash E\#l_i \in \Phi_i} \quad \text{(record elimination)}$$

it can be defined more simply as

$$\frac{\Gamma \vdash E \in \{l:\Phi\}}{\Gamma \vdash E\#l \in \Phi} \quad \text{(record elimination)}$$

It is very easy to verify that the first formulation can be derived from this new formulation by means of the following subtyping relation for records:

$$\frac{\{l_1,\ldots,l_n\} \subseteq \{k_1,\ldots,k_m\} \quad \text{for each } k_i=l_j : \Gamma \vdash \Phi_i \leq \Psi_i}{\Gamma \vdash \{k_1:\Phi_1;\ldots;k_m:\Phi_m\} \leq \{l_1:\Psi_1;\ldots;l_n:\Psi_n\}} \quad \text{(subtype record)}$$

Hence, for a record type to be a subtype of another record type it is necessary that the subtype record has at least the same fields as the other record type. Indeed: it must be allowed to use the subtype in every place where the original type is expected and thus it needs at least as many information as its supertype. Furthermore each of the types of the fields of the subtype need to be subtypes of the types of the corresponding fields (if they exist) in the original record type.

Here are the equational rules for records:

$$\frac{\text{for each } i : \Gamma \vdash A_i=B_i \in \Phi_i}{\Gamma \vdash \{l_1=A_1;\ldots;l_n=A_n\} = \{l_1=B_1;\ldots;l_n=B_n\} \in \{l_1:\Phi_1;\ldots;l_n:\Phi_n\}} \quad \text{(eq record)}$$

$$\frac{\Gamma \vdash \{l_1=E_1;\ldots;l_n=E_n\} \in \{l_1:\Phi_1;\ldots;l_n:\Phi_n\}}{\Gamma \vdash \{l_1=E_1;\ldots;l_n=E_n\}\#l_i = E_i \in \Phi_i} \quad \text{(eq projection)}$$

And finally, (eq surjection) is a kind of η-rule for records:

$$\frac{\Gamma \vdash R \in \{l_1:\Phi_1;\ldots;l_n:\Phi_n\}}{\Gamma \vdash R = \{l_1=R\#l_1;\ldots;l_n=R\#l_n\} \in \{l_1:\Phi_1;\ldots;l_n:\Phi_n\}} \quad \text{(eq surjection)}$$

9.2. Existential quantification

9.2.1. Example

Existential quantification enlarges the expressiveness of our language by allowing abstract data types with hidden representation. The use of *existential types* to model encapsulation is illustrated in [PiTu92] and [Pier93]. They encode the type of one-dimensional point objects as:

```
Point = ∃ Rep≤TOP(*).{ state : Rep;
                       methods : {setX : Rep→(Integer→Rep);
                                  getX : Rep→Integer} }
```

Both the state of the object and the fact that the methods operate on it are visible in this encoding, but the existential type protects the state from external access. Indeed the existential type tells us that the state is of *some* type `Rep`, but it is not specified which type this is. So it is not possible to directly manipulate the state, because its type is unknown. In other words, the only thing we know about objects of type `Point` is that they have the following structure:

```
{ state : Rep;
  methods : {setX : Rep→(Integer→Rep);
             getX : Rep→Integer} }
```

but the representation type `Rep` is unknown. So some of the structure of the objects is hidden, but enough structure is visible to allow manipulations of the objects through operations the objects themselves provide (in this case: record operations).
New point objects must be created using *packing* (or *existential introduction*), for example:

```
p =   pack  {state = {x=5};
             methods =
                   {setX = λs:{x:Integer}.λi:Integer.{x=i};
                    getX = λs:{x:Integer}.s#x}}
      as    Point
      hiding {x:Integer}
```

This makes a new object of an existential type `Point` where the representation type `{x:Integer}` is hidden.
To invoke a method, for example `(Point'setX (p))(4)` we can proceed as follows. First, we *open* (or *unpack*) `p`, binding a type variable `Rep` to the actual (hidden) representation type of `p` and binding a variable `r` to the record containing its state and methods. Then we apply the `setX` function from `r#methods` to `r#state` and the new coordinate `i`, producing a new value of type `Rep`, which is repackaged as a `Point` object with the same methods and hidden representation type as the original. In other words:

```
Point'setX = λp:Point.open p as [Rep,r] in
                    λi:Integer.
                        pack { state = (r#methods#setX(r#state))(i);
                               methods = r#methods}}
                        as Point
                        hiding Rep
```

`(Point'setX (p))(4)` returns an object of type `Point` that is the same as `p` except that its internal state `{x=5}` is changed to `{x=4}`.

9.2.2. Syntax and rules

To introduce existential quantification, we add the following syntax:

```
<type> ::=   ...
       |     ∃<type-variable>≤<type>.<type>          existentially
                                                     quantified type

<term> ::=   ...
       |     pack <term>
             as <type>
             hiding <type>                           packing
       |     open <term>
             as [<type-variable>,<variable>]
             in <term>                               unpacking
```

Packing is used to create new objects of an existentially quantified type. Unpacking is necessary when access is needed to the different components of an object of an existentially quantified type.

Of course, we still have to specify the kinding, typing, subtyping and equational rules for existentially quantified types. Because existential quantification ∃ is "dual" to universal quantification ∀, these rules will be "dual" to the rules of ∀. For example, the kinding rule for ∃ is:

$$\frac{\Gamma, \omega \leq \Phi \vdash \Psi \in *}{\Gamma \vdash \exists \omega \leq \Phi . \Psi \in *} \quad (\exists)$$

The subtyping rule (subtype ∃) becomes:

$$\frac{\Gamma, \omega \leq \Theta \vdash \Phi_1 \leq \Phi_2 \ ; \ \Gamma \vdash \exists \omega \leq \Theta . \Phi_1 \in *}{\Gamma \vdash \exists \omega \leq \Theta . \Phi_1 \leq \exists \omega \leq \Theta . \Phi_2} \quad \text{(subtype } \exists\text{)}$$

Finally, we give the dual rules of (∀ elimination) and (∀ introduction).

$$\frac{\Gamma \vdash A \in \exists \omega \leq \Phi . \Psi \ ; \ \Gamma, \omega \leq \Phi, x : \Psi \vdash B \in \Theta}{\Gamma \vdash \textbf{open } A \textbf{ as } [\omega, x] \textbf{ in } B \in \Theta} \quad (\exists \text{ elimination})$$

This rule shows how the type of an unpacked object can be derived.

$$\frac{\Gamma \vdash \Phi \cong \exists \omega \leq \Psi_1 . \Psi_2 \ ; \ \Gamma \vdash \Theta \leq \Psi_1 \ ; \ \Gamma \vdash E \in [\Theta/\omega]\Psi_2}{\Gamma \vdash \textbf{pack } E \textbf{ as } \Phi \textbf{ hiding } \Theta \in \Phi} \quad (\exists \text{ introduction})^{22}$$

The reduction rules are:

$$\frac{\Gamma \vdash A \in \exists \omega \leq \Phi . \Psi \ ; \ \Gamma, \omega \leq \Phi, x : \Psi \vdash B \in \Phi}{\Gamma \vdash \textbf{open } A \textbf{ as } [\omega, x] \textbf{ in } B = \textbf{open } A \textbf{ as } [\varphi, y] \textbf{ in } \{\varphi/\omega\}\{y/x\}B \in \Phi} \quad (\exists\ \alpha)$$

$$\frac{\Gamma \vdash \Phi \cong \exists \omega \leq \Psi_1 . \Psi_2 \ ; \ \Gamma \vdash A \in [\Theta/\omega]\Psi_2 \ ; \ \Gamma \vdash \Theta \leq \Psi_1 \ ; \ \Gamma, \omega \leq \Psi_1, x : \Psi_2 \vdash B \in \Omega}{\Gamma \vdash \textbf{open } (\textbf{pack } A \textbf{ as } \Phi \textbf{ hiding } \Theta) \textbf{ as } [\omega, x] \textbf{ in } B = [A/x][\Theta/\omega]B \in \Omega} \quad (\exists\ \beta)$$

[22] The symbol ≅ is used to denote the relation "≤ in both directions", which is not necessarily the same as the equality relation.

$$\frac{\Gamma \vdash E \in \exists\omega{\leq}\Phi.\Psi}{\Gamma \vdash \textbf{open } E \textbf{ as } [\omega,x] \textbf{ in } (\textbf{pack } x \textbf{ as } \exists\omega{\leq}\Phi.\Psi \textbf{ hiding } \omega) = E \in \exists\omega{\leq}\Phi.\Psi} \quad (\exists\ \eta)$$

The equational theory is augmented with the following rules for existential types:

$$\frac{\begin{array}{l}\Gamma \vdash \Phi' \cong \exists\omega{\leq}\Psi_1.\Psi_2' \ ;\ \Gamma \vdash \Phi \approx \exists\omega{\leq}\Psi_1.\Psi_2 \ ; \\ \Gamma \vdash \Theta,\Theta' \leq \Psi_1 \ ;\ \Gamma \vdash \Omega \leq [\Theta'/\omega]\Phi',[\Theta/\omega]\Phi' \\ \Gamma \vdash \Phi' \leq \Phi \ ;\ \Gamma \vdash A = B \in [\Omega/\omega]\Psi_2\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{pack } A \textbf{ as } \Phi \textbf{ hiding } \Theta = \\ \qquad \textbf{pack } B \textbf{ as } \Phi' \textbf{ hiding } \Theta' \in \Phi\end{array}} \quad \text{(eq packing)}$$

$$\frac{\begin{array}{l}\Gamma \vdash E_1 = E_2 \in \exists\omega{\leq}\Psi_1.\Psi_2 \ ; \\ \Gamma,\omega{\leq}\Psi_1,x{:}\Psi_2 \vdash B_1 = B_2 \in \Phi\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{open } E_1 \textbf{ as } [\omega,x] \textbf{ in } B_1 = \\ \qquad \textbf{open } E_2 \textbf{ as } [\omega,x] \textbf{ in } B_2 \in \Phi\end{array}} \quad \text{(eq open)}$$

Since all of the above rules (except perhaps rule $(\exists\ \eta)$) are valid (see [HoPi94]) under the usual encoding of existential types in terms of universal quantifiers:

$\exists\omega{\leq}\Phi.\Psi := \forall\Theta{\leq}\mathrm{Top}\ (*).(\forall\omega{\leq}\Phi.\Psi{\to}\Theta){\to}\Theta$

existential types are not really an extension to system $F^{\leq}_{\omega}$.

9.3. Recursive types

9.3.1. Example

A recursive type is a type that satisfies a recursive type equation, such as the following equation for defining an infinite list of integers:

```
IntList = {first:Integer; next:IntList}
```

If L is a term of type IntList, then we can retrieve the fourth integer of this list as follows:

```
L#next#next#next#first
```

Note that the above "definition" of `IntList` is is not a "real" definition, since it defines `IntList` in terms of itself. It must be regarded as an equational property that `IntList` must satisfy. However, instead of explicitly using such recursive type equations, we will use terms such as

```
μω:*.{first:Integer; next:ω}
```

to indicate the canonical solution of such type equations. Of course, we will then need an "unfolding" rule formalising the fact that a type of this form indeed satisfies the recursive type equation. For example if we define `Intlist` as

```
IntList = μω:*.{first:Integer; next:ω}
```

then this rule must allow us to prove that

```
IntList ≅ {first:Integer; next:IntList}
```

Another use of recursive types is the encoding of objects as elements of recursive record types. For example, the type of movable, one-dimensional point objects is usually encoded as:

```
Point = μRep:*.{ setX:Rep→Int→Rep;
                 getX:Rep→Int }
```

9.3.2. Syntax and rules

The following extension of the basic $F_{\omega}^{\leq}$ calculus with recursive types (see appendices of [HoPi92] and [HoPi94]) is somewhat informal. The set of types is extended as follows:[23]

```
<types> ::=  ...
      |      μω:𝒦.Φ                                   (least fixed point)
```

The inference rules are extended by the obvious formation rule and two subtyping rules: one for "unfolding" a recursive type and one for (finitely) comparing two recursive types.
The formation rule is obvious:

$$\frac{\Gamma, \omega \leq \mathbf{TOP}(\mathcal{K}) \vdash \Phi \in \mathcal{K}}{\Gamma \vdash \mu\omega:\mathcal{K}.\Phi \in \mathcal{K}} \qquad \text{(recursion)}$$

The unfolding rule allows us to prove that a recursive type of the form $\mu\omega:\mathcal{K}.\Phi(\omega)$ indeed corresponds to the canonical solution of the type equation $\omega = \Phi(\omega)$.

$$\frac{\Gamma \vdash \mu\omega:\mathcal{K}.\Phi \in \mathcal{K}}{\Gamma \vdash \mu\omega:\mathcal{K}.\Phi \cong [\mu\omega:\mathcal{K}.\Phi/\omega]\Phi} \qquad \text{(unfolding)}$$

Indeed, if we define `Intlist` as

```
IntList  = μω:*.{first:Integer;next:ω}
```

then this rule tells us that

```
IntList   = μω:*.{first:Integer;next:ω}
          ≅ [μω:*.{first:Integer;next:ω}/ω]
                {first:Integer;next:ω}
          = {first:Integer;next:μω:*.{first:Integer;next:ω}}
          = {first:Integer; next:IntList}
```

So indeed IntList satisfies the expected type equation

```
IntList ≅ {first:Integer;next:IntList}
```

In fact, the unfolding rule corresponds to the intuition that recursive types are simply denotations for their infinite expansion.

Finally, the subtyping rule for recursive types captures the intuition that a recursive type is a subtype of another one, if all the finite approximations of the first one are subtypes of the corresponding finite approximations of the second one. This can be axiomatised via the following rule:

$$\frac{\Gamma, \varphi_2 \leq \mathrm{TOP}(\mathcal{K}), \varphi_1 \leq \varphi_2 \vdash \Psi_1 \leq \Psi_2}{\Gamma \vdash \mu\varphi_1:\mathcal{K}.\Psi_1 \leq \mu\varphi_2:\mathcal{K}.\Psi_2} \qquad \text{(subtype recursion)}$$

That is, if by assuming $\varphi_1 \leq \varphi_2$ we can verify $\Psi_1 \leq \Psi_2$, then we can deduce the inclusion of the recursive types $\mu\varphi_1:\mathcal{K}.\Psi_1 \leq \mu\varphi_2:\mathcal{K}.\Psi_2$. For example, if we know that `Natural≤Integer`, then the assumption $\varphi_1 \leq \varphi_2$ implies

[23] With [HoPi92], [HoPi94] and others we use the notation **μ** instead of **rec** (as in [CaWe85] and [PiTu92]) to denote recursive types.

$$\{\text{first:Natural;next:}\varphi_1\} \le \{\text{first:Integer;next:}\varphi_2\}$$

and thus we can safely deduce NatList≤IntList, where

$$\text{NatList} = \mu\varphi_1{:}*.\{\text{first:Natural; next:}\varphi_1\}$$
$$\text{IntList} = \mu\varphi_2{:}*.\{\text{first:Integer; next:}\varphi_2\}$$

For more details on the extension of system $F_\omega^{\le}$ with recursive types, see [AmCa90].

9.3.3. Fixed-point operator

[HoPi94] shows how a value-level fixed-point operator can be added to system $F_\omega^{\le}$ with recursive types. First of all a typing rule for the fixed-point combinator *fix* is needed

$$fix \in \forall\psi{\le}\text{Top}(*).(\psi{\to}\psi){\to}\psi \qquad \text{(subtype recursion)}$$

and in addition to the equations implied by its typing, the equational rules

$$\frac{\Gamma \vdash f \in \psi{\to}\psi}{\Gamma \vdash fix[\psi]f = f(fix[\psi]f) \in \psi} \qquad \text{(fix)}$$

$$\frac{\Gamma \vdash \psi_2{\le}\psi_1 \;;\; \Gamma \vdash f_1 \in \psi_1{\to}\psi_1 \;;\; \Gamma \vdash f_2 \in \psi_2{\to}\psi_2 \;;\; \Gamma \vdash f_1{=}f_2 \in \psi_2{\to}\psi_1}{\Gamma \vdash fix[\psi_1]f_1 = fix[\psi_2]f_2 \in \psi_1} \qquad \text{(subtype fix)}$$

which respectively characterise *fix* as a fixed-point combinator and describe its behaviour with respect to subtyping.

10. Conclusion

In section 2, we gave a survey of some extensions of Church's type free lambda calculus. However, it was by no means our intention to present a formal description of all possible extensions of standard lambda calculus. On the contrary, the primary goal of this paper was to obtain a more or less complete description of one specific extension, namely system $F_\omega^{\le}$. This was achieved by starting from type free lambda calculus, and stepwise introducing new syntactic constructs and derivation rules.

The importance of system $F_\omega^{\le}$ is that it can be used as a basis for constructing formal object oriented models. For example [PiTu92] proposes an encoding of a class-based language based on system $F_\omega^{\le}$ extended with existential types, and [Pier93] does the same for a delegation-based language. Hofmann and Pierce ([HoPi92] and [HoPi94]) give a direct type-theoretic characterisation of the basic mechanisms of object oriented programming, by introducing an explicit *Object* type constructor and suitable introduction, elimination and equality rules into $F_\omega^{\le}$.

Acknowledgements

Lots of thanks to Tom Mens, Hendrik Tews, Wolfgang De Meuter and Benjamin Pierce for providing many useful comments when proofreading draft versions of this paper.

References

[AmCa90] : Amadio, R., and L. Cardelli, 1990
Subtyping recursive types
ACM Transactions on Programming Languages and Systems

[Bare84] : Barendregt, H., 1984
The lambda calculus: its syntax and semantics.
Noth-Holland

[CaCo90] : Cardone, F., and M. Coppo, 1990
Two extensions of Curry's type inference system
Academic Press, Logic and Computer Science, pp. 19-75

[CaMi89] : Cardelli, L., and J. Mitchell, 1989
Operations on records
Springer-Verlag, LNCS 442

[Card88] : Cardelli, L., 1988
A semantics of multiple inheritance
Information and Computation, No.76, pp. 138-164

[Card90] : Cardelli, L., 1990
Notes about $F_{\omega}^{\leq}$
Unpublished manuscript

[CaWe85] : Cardelli, L., and P. Wegner., 1985
On understanding types, data abstraction, and polymorphism
ACM Computing Surveys, Vol.17, No.4

[CCHM89] : Canning, P., W. Cook, W. Hill, J. Mitchell, and W. Olthoff, 1989
F-bounded polymorphism for object-oriented programming
ACM Proceedings, 4th International Conference on Functional Programming Languages and Computer Architectures, pp. 273-280

[Chur40] : Church, A., 1940
A formulation of the simple theory of types
Journal of Symbolic Logic, No.5, pp. 56-68

[GiLT90] : Girard, J-Y., Y. Lafont, and P. Taylor, 1990
Proofs and types
Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 7

[Gira72] : Girard, J-Y., 1972
Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.
Ph.D. thesis, Univeristé Paris VII

[HoPi92] : Hofmann, M., and B. Pierce, 1992
An abstract view of objects and subtyping (Preliminary Report)
University of Edinburgh, ECS-LFCS-92-226

[HoPi94] : Hofmann, M., and B. Pierce, 1994
A Unifying type-theoretic framework for objects
University of Edinburgh

[Huet90] : Editor Huet, G., 1990
Logical foundations of functional programming
Addison Wesley

[Mitc92] : Mitchell, J., 1992
Foundations for object-oriented programming
Tutorial at the TOOLS conference

[PiDM89] : Pierce, B., S. Dietzen, and S. Michaylov, 1989
Programming in higher-order typed lambda calculi
Carnegie Mellon University, CMU-CS-89-111

[Pier93] : Pierce, B., 1993
A model of delegation based on existential types
University of Edinburgh

[PiTu92] : Pierce, B., and D. Turner., 1992
Object-oriented programming without recursive types
University of Edinburgh

[Reve88] : Revesz, G., 1988
Lambda calculus, combinators, and functional programming
Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 4

[Reyn74] : Reynolds, J., 1974
Towards a theory of type structure.
Springer-Verlag, LNCS 19

[StPi94] : Steffen, M., and B. Pierce, 1994
Higher-Order Subtyping
University of Edinburgh, ECS-LFCS-94-280

[Wand89] : Wand, M., 1989
Type inference for record concatenation and multiple inheritance
Proceedings 4th IEEE Symposium on Logic in Computer Science, pp.92-97