Chapter 6

Functions, Records and Compatibility in the λN Calculus

Laurent Dami

Abstract Subtyping, a fundamental notion for software reusability, establishes a classification of data according to a compatibility relationship. This relationship is usually associated with records. However, compatibility can be defined in other situations, involving for example enumerated types or concrete data types. We argue that the basic requirement for supporting compatibility is an interaction protocol between software components using *names* instead of *positions*. Based on this principle, an extension of the lambda calculus is proposed, which combines de Bruijn indices with names. In the extended calculus various subtyping situations mentioned above can be modelled; in particular, records are encoded in a straightforward way. Compatibility is formally defined in terms of an operational lattice based on observation of error generation. Unlike many usual orderings, errors are not identified with divergence; as a matter of fact, both are even opposite since they respectively correspond to the bottom and top elements of the lattice. Finally, we briefly explore a second extension of the calculus, providing meet and join operators through a simple operational definition, and opening interesting perspectives for type checking and concurrency.

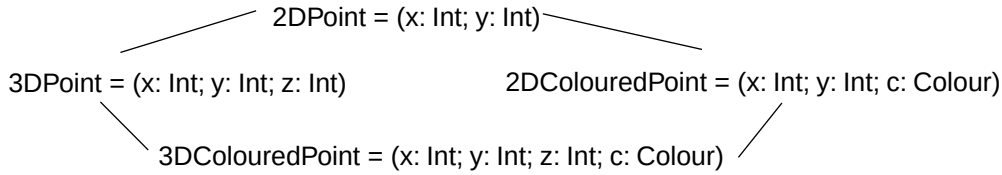
6.1 Introduction

The lambda calculus is a widely used tool for studying the semantics of programming languages. However, there are at least two categories of programming features that cannot be modelled in the lambda calculus. One is *concurrent programming*, in which the non-determinism introduced by operations taking place in parallel cannot be captured by lambda expressions. The other is *subtyping*, which plays a prominent role in object-

Laurent Dami, "Functions, Records and Compatibility in the Lambda N Calculus," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 153-174, Prentice Hall, 1995.

oriented systems, and is interesting for software reuse in general. Subtyping is based on a classification of data according to collections of valid operations; an operation valid for one type is also valid for its subtypes. The term *plug compatibility* is sometimes used to express this relationship. In the lambda calculus, functional application is the only operation, and provides no support for such a classification. Therefore, the lambda calculus must be extended to deal with subtyping: the common approach is to use *records* [9][10]. In this paper we argue that subtyping does not reduce to record systems. We propose an extended lambda calculus λN (lambda calculus with names) which can encode records — and therefore objects as well — but is more general since it also supports plug compatibility on enumerated types and concrete data types.

Our calculus is based on the observation that reusability in record systems is mainly due to the use of *names* for accessing fields, instead of *positions* in simple Cartesian products; the difference is important when considering extensibility. A product type can be extended in one direction, by adding a new component in the last position: any projections valid for the original product are still valid for the extended product. In that view, the type (Int × Int) can be seen as a supertype of (Int × Int× Colour) . However, this ordering based on positions can only have a tree structure. By contrast, an ordering based on names can be any partial order. A well-known example is the ordering of various types of points in a record system:



Like Cartesian products, functions use positional information to identify parameters; this is the basis for the *currying* property, which allows any function of *n* arguments to be encoded as a hierarchy of *n* lambda abstractions, with one single argument at each abstraction level. However, functions cannot be ordered in a tree structure: there is no plug-compatibility relationship between a function with three arguments and a function with only two arguments. This can be illustrated with a simple example: consider the Church encoding of Booleans and the *not* function in standard λ calculus[5]:

True	=	λt.λf.t
False	=	λt.λf.f
Not	=	λb.λt.λf.b f t

and imagine we now want a three-valued logic, with an *unknown* value. We must add a new argument, and everything has to be recoded:

True_U	=	λt.λf.λu.t
False_U	=	λt.λf.λu.f
Unknown_U	=	λt.λf.λu.u
Not_U	=	λb.λt.λf.λu.b f t u

The new encoding is incompatible with the previous one. In particular, it does not make sense to apply Not_U to True: it can only be applied to True_U. In a software reusability per-

spective, this implies that any existing component producing True or False values needs to be modified to be usable with the new logic.

In order to get a compatibility relationship on functions, we propose a simple extension of the lambda calculus, inspired from records: functions are allowed to have multiple parameters *at the same abstraction level*, and those parameters are distinguished by their name. It then becomes necessary to specify which name is being bound in a functional application, but this is precisely the basis for reusability and subtyping: binding more names than those actually used by the function does no harm, and therefore a function with arguments (*x y*) is compatible with a function with arguments (*x y z*), because both can accept a sequence of bindings on names *x*, *y* and *z*.

A consequence of this approach is that names participate in the semantics of functions, and it is no longer possible to consider lambda expressions modulo α-equivalence (renaming of bound variables). However, α-renaming is important in the standard lambda calculus to avoid the well-known problem of name capture in substitutions. The difficulty is avoided by using *de Bruijn indices* [8] to indicate unambiguously the relationship between an applied occurrence of a variable and its corresponding abstraction level, and furthermore using names to distinguish between multiple variables at the same abstraction level. A variable, then, is a pair containing both a name and an index.The λN encoding of Booleans is:

```
True    =   λ (t, 0)
False   =   λ (f, 0)
Not     =   λλ (b, 1)(t→(f, 0))(f→(t, 0))!
```

For example, (t, 0) in True is a variable. The 0 index tells that this variable is bound by the closest abstraction level (the closest 'λ'). The other component of the pair tells that, among the parameters associated with that abstraction level, the one with name *t* is to be chosen. Parameter binding is done through the notation *a*(*x*→*b*), where *a* and *b* are terms, and *x* is a name. So in the Not function, the variable (b, 1), which refers to the outermost abstraction level, receives two bindings on parameters *t* and *f*. The exclamation mark at the end "closes" the sequence of bindings and removes an abstraction level.

As for the de Bruijn calculus, notation involving indices is convenient for machine manipulations, but hard for humans to read. Fortunately, indices can be hidden easily, by using a higher-level syntax with a simple translation function into the low-level representation. This higher-level syntax will be used for all programming examples in this chapter, while the low-level syntax is retained for presenting the semantics of the calculus. In high-level syntax, the expressions above become:

```
True    =   λ(t) t
False   =   λ(f) f
Not     =   λ(b) λ(t, f) b(t→f)(f→t)!
```

Informally, the names in parenthesis following a λ are parameters, so now they are explicitly declared instead of being implicitly recovered from indices. As an example of a derivation, consider the application of Not to True:

```
Not(b→True)!   =   (λ(b) λ (t, f) b(t→f)(f→t)!) (b→λ(t) t)!
```

The outermost binding on *b* is reduced, substituting the internal reference to *b* by True, and removing *b* from the parameter list:

$$(\lambda()\ \lambda(t, f)\ (\lambda(t)\ t)(t{\to}f)(f{\to}t)!)!$$

Then, by reducing the outermost '!', one abstraction level (one 'λ') is removed:

$$\lambda(t, f)\ (\lambda(t)\ t)(t{\to}f)(f{\to}t)!$$

The binding on *t* substitutes *t* by *f* and removes *t* from the parameter list:

$$\lambda(t, f)\ (\lambda()\ f)(f{\to}t)!$$

The binding on *f* is simply dropped, because the abstraction to which it is applied has no *f* parameter:

$$\lambda(t, f)\ (\lambda()\ f)!$$

Finally, one 'λ' is removed because of the '!':

$$\lambda(t, f)\ f$$

and although this final result declares both *t* and *f* as parameters instead of only *f*, it is equivalent to False, because its translation into low-level syntax with indices is also $\lambda(f, 0)$.

Now the interesting point about this calculus is that, in order to get an augmented logic, we just write:

$$\begin{array}{lcl} \mathsf{Unknown} & = & \lambda(u)\ u \\ \mathsf{Not}_U & = & \lambda(b)\ \lambda(t\ f\ u)\ b(t{\to}f)(f{\to}t)(u{\to}u)! \end{array}$$

Not is recoded (which is normal), but we can keep the original encodings of True and False. This cannot be done in the standard lambda calculus, and is interesting for *reusability*: any other module based on the original encoding is still compatible with our new logic and does not need modification.

To the best of our knowledge, the idea of using names in a lambda calculus setting was not studied much in the literature. Two related systems are John Lamping's *unified system of parameterization* [19] and Garrigue and Aït-Kaci's *label-selective lambda calculus* [3][16]. However, both calculi treat names (or "labels") and variables as orthogonal concepts, whereas we unify them through the use of de Bruijn indices.

6.2 A Lambda Calculus with Named Parameters

It is well known that names in the standard lambda calculus are only useful to express a relationship between binding occurrences and applied occurrences of variables. Once that relationship is established, i.e. with bound variables, names can be replaced by other names through α *substitution*, or can even be removed altogether: in [8] de Bruijn proposed a modified lambda calculus in which variables are simply denoted by *indices*. A de Bruijn index is a non-negative integer expressing the distance between an applied occurrence of a variable and the abstraction level to which it refers. For example, the *not* function, written

$$\mathsf{Not} \quad = \quad \lambda b.\lambda t.\lambda f.b\ f\ t$$

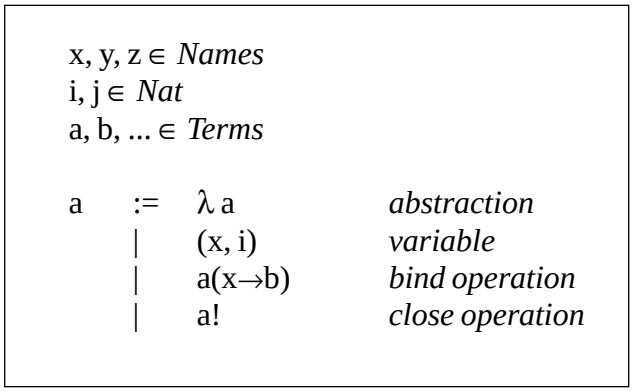


Figure 6.1 *Abstract syntax.*

in the standard calculus, becomes

Not = λλλ 2 0 1

in de Bruijn notation (here we start indices with 0, while some authors start with 1; the difference is not significant). There is a straightforward translation from usual lambda expressions to their de Bruijn version. The de Bruijn notation provides a *canonical* representation: all α-equivalent lambda expressions have the same translation. Furthermore, the well-known problem of *name capture* is avoided. Both in standard and de Bruijn calculi, each abstraction level (each 'λ') introduces exactly one variable. Our proposal is to allow *several* variables at the same abstraction level. To do so, de Bruijn indices are retained, but in addition *names* are used to distinguish between different variables at the same level. This section defines the calculus; the next section shows that this extension provides support for plug-compatibility.

6.2.1 Abstract (Low-level) Syntax

Figure 6.1 presents the abstract syntax of λN. The language is built over a (finite) set of *names*.

An *abstraction* corresponds to the traditional notion of abstraction. Like in the de Bruijn lambda calculus, abstractions need not introduce names for their parameters: the connection between variables and their corresponding abstraction level directly comes from the indices associated with variables (see below).

A *variable* is a name together with a de Bruijn index. This means that an abstraction can have several parameters, all with the same index, which are distinguished by their name. The index indicates the abstraction level (which 'λ') a variable is referring to: an index of 0 refers to the closest abstraction, and higher numbers refer to farther abstraction levels.

A *bind operation* partly corresponds to the usual notion of application. However, since an abstraction may have several parameters, it is necessary to specify which of them is bound in the expression. Therefore the construct *a*(*x*->*b*) means: "bind *b* to the parameter

$$
\begin{array}{lcl}
FV_k((x, i)) & = & \text{if } i = k \text{ then } \{(x, i)\} \text{ else } \{\} \\
FV_k(\lambda a) & = & \{(x, i) \mid (x, i+1) \in FV_{k+1}(a)\} \\
FV_k(a(x \rightarrow b)) & = & FV_k(a) \cup FV_k(b) \\
FV_k(a!) & = & FV_k(a) \\
\\
FV(a) & = & \bigcup_{k \geq 0} FV_k(a) \\
\\
\textit{parameters}(\lambda a) & = & FV_0(a) \\
\\
a \textit{ closed} & \Leftrightarrow & FV(a) = \{\}
\end{array}
$$

Figure 6.2 *Free and bound variables.*

with name *x* in *a*", or, expressed differently: "substitute *b* for every occurrence of (*x*, 0) in *a* (modulo index renumbering, as defined below)". The parameters of an abstraction may be bound separately, and in any order.

A *close operation* closes a sequence of bindings, and removes an abstraction level (removes one 'λ').

Notions of parameters, free and bound variables are as in the de Bruijn calculus; a formal definition is given in figure 6.2.

6.2.2 Reduction Rules

In the de Bruijn calculus, β-reduction involves some renumbering of indices: whenever the number of 'λ's above a subterm changes, its free variables have to be adapted in consequence. One way to express it is

$$(\lambda a)\, b \quad \rightarrow_\beta \downarrow_0 [a\,[0 := \uparrow_0 [b]]$$

where '↑' (lift) is an operation incrementing all free variables by 1, '↓' (unlift) is the reverse operation, and $a[i := b]$ is the substitution of *b* for all occurrences of *i* in *a* (again modulo index renumbering).

The reduction rules for λN, given in figure 6.3, are very similar, since they also involve index manipulation operations. There are two kinds of reductions, called *bind reduction* (β) and *close reduction* (γ). Basically, the operations performed by β-reduction in the standard lambda calculus have been split in two: binding reductions substitute values for variables, and close reductions "remove the lambda" and unlift the result, i.e. they remove an abstraction level. The definitions for lifting and substitution operations are given in figure 6.4

$$(\lambda a)(x \rightarrow b) \quad \rightarrow_\beta \quad \lambda(a[(x, 0) := \uparrow_0[b]])$$
$$(\lambda a)! \quad \rightarrow_\gamma \quad \downarrow_0[a]$$

Figure 6.3 *Reduction rules.*

Lifting/Unlifting

$$\uparrow_k[(x, i)] = \text{if } (i < k) \text{ then } (x, i) \text{ else } (x, i+1)$$
$$\downarrow_k[(x, i)] = \text{if } (i < k) \text{ then } (x, i) \text{ else if } (i=k) \text{ then } \mathbf{err} \text{ else } (x, i-1)$$

$$\mid_k[\lambda a] = \lambda(\mid_{k+1}[a])$$
$$\mid_k[a(x \rightarrow b)] = \mid_k[a](x \rightarrow \mid_k[b])$$
$$\mid_k[a!] = (\mid_k[a])! \qquad \textit{where } `\mid` \textit{ is either } `\downarrow` \textit{ or } `\uparrow`$$

$$\mathbf{err} =_{def} \mathbf{E}(x \rightarrow \mathbf{E})! \qquad \textit{where } \mathbf{E} = \lambda\lambda\,(x, 1)(x \rightarrow (x, 1))!$$

Substitution

$$(y, j)[(x, i) := b] = \text{if } ((x, i) = (y, j)) \text{ then } b \text{ else } (y, j)$$
$$(\lambda a)\ [(x, i) := b] = \lambda(a[(x, i+1) := \uparrow_0[b]])$$
$$(a(y \rightarrow c))\ [(x, i) := b] = (a[(x, i) := b])(y \rightarrow c[(x, i) := b])$$
$$(a!)[(x, i) := b] = (a[(x, i) := b])!$$

Figure 6.4 *Lifting and substitution operations.*

Careful readers will have noticed that in λN we may need to unlift a 0 index, a situation which never occurs in the de Bruijn calculus. Consider de Bruijn's β-reduction rule above: all 0 indices are substituted in *a*, so the expression passed to '↓' contains no 0 index. By contrast, the λN expression $(\lambda\,(x, 0))!$ reduces to $\downarrow_0[\,(x, 0)\,]$, which intuitively corresponds to an error (we are trying to access a parameter that has not been bound). As a matter of fact, in such situations the definition of '↓' yields **err**, a specific term representing errors. This will be discussed in detail in section 6.4; for the time being it suffices to know that **err** is not an additional syntactic construct, but rather is defined as a usual term in the language, with the property that further binding or close operations on **err** yield **err** again.

A binding reduction can never introduce new parameters in an abstraction, because the term passed in the substitution is lifted. Therefore if several successive bindings are done, the final result does not depend on the order of the substitutions. This amounts to say that *bindings are commutative*, i.e. expressions of the form

$a(x \rightarrow b)(y \rightarrow c)$ and $a(y \rightarrow c)(x \rightarrow b)$

derive to the same thing, provided that *x* and *y* are *different names*. If *x* and *y* are the same

name, all references to that name are substituted in the first binding, so the second binding is just ignored, and those bindings are not commutative.

6.2.3 Reduction Example

For illustrating the rules, we use again the expression Not(b→True)!. The derivation was given in an informal way in the introduction, using high-level syntax (without indices). Here, the low-level syntax is used; at each step, the lambda and the bind or close operation involved in the next reduction step are underlined.

1 (λλ (arg, 1) (true→(false, 0))(false→(true, 0))!)(arg→λ(true, 0))!
2 (λλ (λ(true, 0))(true→(false, 0))(false→(true, 0))!) !
3 λ (λ(true, 0))(true→(false, 0))(false→(true, 0))!
4 λ (λ(false, 1))(false→(true, 0))!
5 λ (λ(false, 1)) !
6 λ (false, 0)

The final result is False. Notice at line 4 that the binding of false simply gets eliminated: this is because the abstraction (λ(false, 1)) has *no* parameter called false; it indeed uses a variable with that name, but since the index is not 0 this is a free variable, not a parameter.

At some intermediate stages (e.g. at line 2) several reductions could occur; the sequence shown here corresponds to *normal-order* reduction (choosing leftmost outermost redex first). It is therefore legitimate to ask whether a different reduction sequence would yield the same result (whether the language is confluent). The answer is *yes*, and has been established in [13]. So, as in the standard lambda calculus, results are independent from the reduction sequences through which they were obtained; furthermore, if an expression does have a result, then the normal-order reduction strategy is guaranteed to yield that result (i.e. not to diverge).

Notice that if we "forget" to supply an argument to Not before applying a close operation, as in Not!, we have the reduction

(λλ (arg, 1) (true→(false, 0))(false→(true, 0))!) !
λ **err** (true→(false, 0))(false→(true, 0))!
λ **err** (false→(true, 0))!
λ **err** !
λ **err**

which is equivalent to **err**, i.e. an error is produced.

6.2.4 Higher-level Syntax

Indices were necessary for defining the calculus, but are difficult to read. In order to work practically with the calculus, we will use a higher-level syntax, given in figure 6.5, in which the indices need not be explicitly written. There is a straightforward translation **T** from this syntax into the original syntax, which is formally defined in figure 6.6. In this

v	:=	x	*simple variable*
	|	\v	*"outer" variable*
a	:=	$\lambda(x_1 \ldots x_n)$ a	*abstraction*
	|	v	*variable*
	|	a(x→b)	*bind operation*
	|	a!	*close operation*

Figure 6.5 *Higher-level syntax.*

$\mathbf{T}_V[\lambda(x_1 \ldots x_n)\ a] = \lambda(\mathbf{T}_{V'}[a])$
where $V' = \{(x, i+1) \mid (x, i) \in V\} \cup \{(x_1, 0), ..., (x_n, 0)\}$

$\mathbf{T}_V[\backslash ... \backslash x] = matchVar\ V\ (x, i)$ *where* i *is the number of* '\'

$\mathbf{T}_V[a(x \rightarrow b)] = \mathbf{T}_V[a](x \rightarrow \mathbf{T}_V[b])$

$\mathbf{T}_V[a!] = (\mathbf{T}_V[a])!$

$matchVar\ V\ (x, i) =$ let $J = \{j \mid (x, j) \in V, j \geq i\}$ in
if $(J = \{\})$ then **err**
else $(x, \mathbf{min}(J))$

Figure 6.6 *Translation function.*

new notation, the parameters of an abstraction are *declared* as a list of names in parenthesis. A variable is written simply as a name: the index is recovered by looking for the closest abstraction which declares the same name. In case the same name is used at several abstraction levels, and one wants to override the default variable matching scheme, the name of the variable can be preceded by a collection of backslashes. This tells the translation function to start looking for a declaration, not at the next abstraction level, but one or several levels higher (according to the number of backslashes). The parameter list following a lambda can be empty, as in

λ() Not(arg→True)!

This is like a closure, i.e. a function that needs no arguments but is not evaluated yet (assuming a lazy interpretation as in section 6.4.1). Forcing evaluation is then done with the '!' operator.

The translation **T** from this syntax into the original syntax is like translating the standard lambda calculus into de Bruijn notation (see [12]). The first argument to the translation function is a set of currently declared variables; at each abstraction level this set is updated. The translation is defined for closed terms by taking the initially empty set of variables. Variables which are not declared at any level are translated into an error by the *matchVar* function. As an example of a translation, consider the expression

λ(x y) λ(x z) x + y + z + \x + \y + \z + \\x

(assuming that infix addition is part of the language). After crossing the two abstraction levels, the set V of declared variables is

V={(x, 1), (y, 1), (x, 0), (z, 0)}

and therefore the translation is

λλ (x, 0) + (y, 1) + (z, 0) + (x, 1) + (y, 1) + **err** + **err**

This shows how the backslash can be used to distinguish between parameters with the same name, but at different levels. Notice that *x* and *x* are different variables, while both *y* and *y* are translated into (*y*, 1), because there is no *y* parameter at the inner abstraction level. Furthermore, both *z* and *x* are translated into **err**, because no corresponding variable declaration can be found.

6.3 The Calculus at Work

In this section we show how several common programming constructs are encoded in λN. To make the examples more appealing, we assume that integers, Booleans and strings have been added to the language, with corresponding operations (integer arithmetic, *if* expression, etc.). Such extensions are common for the lambda calculus and can be shown to be conservative, i.e. expressions in the extended language are always convertible into the original language. As a matter of fact, an encoding of Booleans has been seen already, and an encoding of integers is given in section 6.3.4. In consequence, the semantics of the language does not change. We start with a discussion on functions and recursion, just to give a clearer map of the relationship between λN and the standard lambda calculus. Then the specificity of λN, namely the encoding of extensible constructs, is demonstrated through enumerated types, concrete data types and records.

6.3.1 Functions

It can be seen easily that λN contains the usual lambda calculus. Any expression *e* of the pure lambda calculus can be encoded in a straightforward way, by choosing a single arbitrary name (say **arg**) to be associated with variables:

- Take the de Bruijn encoding of *e*.
- Replace every application MN by M(**arg**→N)!, i.e. a binding of **arg** immediately followed by a close operation.
- Replace every variable *i* by (**arg**, i).

For example, the lambda expression λf x y. f(x + y) has de Bruijn encoding λλλ 2(1+0) and becomes here

λλλ(arg, 2)(arg→(arg, 1)+(arg, 0))!

which corresponds to

```
λ(arg) λ(arg) λ(arg) \\arg(arg→\arg+arg)!
```

in the higher-level notation. Now how does this compare to the expression:

```
λ(f x y) f(arg→(x+y))!
```

which intuitively seems more natural? In both formulations, the arguments can be bound and the final result evaluated. The difference appears with partial bindings. When arguments are declared at the same abstraction level, as we do in the second formulation, they can be bound separately, in any order, and even if all arguments are supplied, the internal expression is not evaluated until a close operation takes place. This can be useful, as we will see later, for building lazy data structures. Furthermore, such functions are polymorphic, in the sense that any context which binds more arguments than just *f, x* and *y* will accept this abstraction without generating an error. However, if we want to do partial bindings, leaving the other arguments open, the close operation cannot be inserted, which implies that we lose the currying property, i.e. the possibility to bind one single argument and get in return another function over the remaining arguments. This is because usual functional application corresponds here to a binding *and* a close operation. When writing a function, there is therefore a choice to make about how to organize its arguments. The methodological issues involved in such choices have not been explored yet. Our choices in the coming examples are guided by some heuristics acquired during our various experiences in using the system.

6.3.2 Recursion

A fixed-point operation over a functional λ(*x*)*a* yields a recursive function, as in the lambda calculus; however, the name *x* must be taken into account in the fixed-point operation. So for each name *x* we define a corresponding fixed-point operator

```
Yx  =  λ(x) (λ(x) \x(x→x(x→x)!)!)(x→ (λ(x) \x(x→x(x→x)!)!))!
```

This is like the usual combinator Y, specialized to bind name *x*. It can be checked that for *f*=λ(*x*)*a* we have

```
Yx(x→f)!  →*  f(x→Yx(x→f)!)!
```

In order to facilitate such recursive definitions we introduce some syntactic sugar: an expression with recursion over parameter *x* is written μ(*x*)*a* and is translated into

```
Yx(x→λ(x)a)!
```

With this extension we can write

```
Factorial  =  μ(f) λ(arg) if (arg > 1) then arg*f(arg→(arg-1))! else 1
```

6.3.3 Extensible Enumerated Types and Case Selection

We already have seen an encoding of Boolean values, which is a simple enumerated type with two values. The approach can be generalized to *n*-ary enumerated types:

```
Green   =  λ(green) green
Orange  =  λ(orange) orange
Red     =  λ(red) red
```

Each colour in the encoding above is a kind of identity function on a particular name. The way to use such values is to perform *case selection*:

```
trafficLight  =  λ(colour) colour(green→Go)(orange→Stop)(red→Stop)!
```

Here we assume two defined driving actions Go and Stop. Depending on the colour, the appropriate driving action is chosen. Observe that case selection is just a sequence of bindings. The set of colours can be extended easily:

```
Blue        =  λ(blue) blue
Violet      =  λ(violet) violet
Yellow      =  λ(yellow) yellow
complement  =  λ(colour) colour(green→Red)(blue→Orange)(violet→Yellow)
                                (red→Green)(orange→Blue)(yellow→Violet)!
```

so the first three colours are "reused" here in a different context, without breaking the original encoding of trafficLight. As explained in the introduction, this can *not* be done in the standard lambda calculus.

6.3.4 Extensible Concrete Data Types

A direct extension from previous section is the encoding of concrete data types. Concrete data types are built through a finite number of *constructors*, which can take arguments. Functions using such data types then have to perform case selection over the constructors. We will consider the example of natural numbers, with two constructors:

```
Zero  =  λ(zero) zero
Succ  =  λ(n) λ(positive) positive(pred→n)!
```

The names zero and positive are used to distinguish constructors. Case selection is done as with enumerated types, except that constructors with arguments must be able to pass the corresponding values to the function using the data type, so there must be a convention between the constructor and its users about which name to use for that purpose. In the case of Succ, the conventional name is pred. An example of using the data type is the addition function:

```
Add  =  μ(add) λ(left right) left
            (zero→right)
            (positive→λ(pred) add(left→pred)(right→Succ(n→right)!)!)!
```

which proceeds by decomposition of the left argument.

The encoding can be extended easily to include negative numbers as well:

```
Pred= λ(n) λ(negative) negative(succ→n)!
Inc=  λ(n) n(zero→Succ(n→n)!)(positive→Succ(n→n)!)(negative→λ(succ)succ)!
Dec= λ(n) n(zero→Pred(n→n)!)(positive→λ(pred)pred)(negative→Pred(n→n)!)!
```

```
Add= μ(add) λ(left right)   left(zero→right)
                             (positive→λ(pred) add(left→pred)(right→Inc(n→right)!)!)
                             (negative→λ(succ) add(left→succ)(right→Dec(n→right)!)!)!
```

Again, functions using only positive numbers need *not* be recoded because of that extension.

Generally speaking, the encoding of data types given here is pretty low-level. However, syntactic sugar for data type constructors and pattern matching, as in most modern functional languages, could be added easily.

6.3.5 Records

A more interesting example of extensibility and polymorphism is the encoding of records. We extend the syntax with a record constructor and a field selection operation; the translation of these constructs is given in figure 6.7. The translation can be understood more

T $[\{x_1=a_1 \ldots x_n=a_n\}]$	=	$\lambda(sel)\ sel(x_1 \rightarrow \uparrow_0[a_1]) \ldots (x_n \rightarrow \uparrow_0[a_n])!$
T $[a.x]$	=	$a(sel \rightarrow \lambda(x)x)!$

Figure 6.7 *Records*

easily through a comparison with the encoding of binary products (pairs) in the standard lambda calculus:

```
(a, b) =   λsel. sel a b
fst =      λpair. pair (λfirst. λsecond. first)
snd =      λpair. pair (λfirst. λsecond. second)
```

The encoding of a pair is a function which takes a *selector* and then binds both members of the pair to that selector. A selector is just a function taking two arguments and returning one of them, so the *fst* projection function applies a selector which extracts the first argument, while the *snd* function applies a selector which extracts the second argument. Similarly, a record in λN is a function which takes a selector, and binds all fields to corresponding named parameters in that selector. Since one abstraction level was added because of the *sel* argument, all internal fields are lifted in order to protect free variables from being captured. A selector for field *x* is just an identity function on that name, so a field selection operation simply binds the appropriate selector to the *sel* argument of the record. Here are some examples:

```
{x=5}          =  λ(sel) sel(x→5)!
{x=3 y=2}      =  λ(sel) sel(x→3)(y→2)!
{x=5}.x        =  (λ(sel) sel(x→5)!)(sel→(λ(x)x))!          →*  5
{x=3 y=2}.x    =  (λ(sel) sel(x→3)(y→2)!)(sel→(λ(x)x))!     →*  3
{x=3 y=2}.z    =  (λ(sel) sel(x→3)(y→2)!)(sel→(λ(z)z))!     →*  (λ(z)z)!   → err
```

$$
\begin{aligned}
&\mathbf{T}\,[\langle x_1 \ldots x_n\rangle] \;=\; \mu(rec)\lambda(x_1 \ldots x_n)\ \{\\
&\qquad get=\{x_1=x_1 \ldots x_n=x_n\}\\
&\qquad set=\{x_1=\lambda(arg)\ rec(x_1\rightarrow arg)(x_2\rightarrow x_2)\ldots(x_n\rightarrow x_n)!\\
&\qquad\qquad \ldots\\
&\qquad\qquad x_i=\lambda(arg)\ rec(x_1\rightarrow x_1)\ldots(x_i\rightarrow arg)\ldots(x_n\rightarrow x_n)!\\
&\qquad\qquad \ldots\\
&\qquad\qquad x_n=\lambda(arg)\ rec(x_1\rightarrow x_1)\ldots(x_i\rightarrow x_i)\ldots(x_n\rightarrow arg)!\\
&\qquad\qquad \}\\
&\qquad \}\\
&\mathbf{T}\,[\langle x_1=a_1 \ldots x_n=a_n\rangle] \;=\; (\mathbf{T}\,[\langle x_1 \ldots x_n\rangle])(x_1\rightarrow a_1)\ldots(x_n\rightarrow a_n)!\\
&\mathbf{T}\,[a\langle x := b\rangle] \;=\; a.set.x(arg\rightarrow b)!
\end{aligned}
$$

Figure 6.8 *Updatable records.*

We see that ".*x*" is a polymorphic operation that can be applied to any record containing at least an *x* field.

The same encoding can support more general operations on records, like a form of "execute in context" operation, similar to quoted expressions in LISP or to the blocks of Smalltalk: for example an expression like

$$r.[x + y + z] \;=\; r(sel\rightarrow\lambda(x\ y\ z)x + y + z)!$$

asks record *r* to add its fields *x*, *y* and *z* and return the result.

Moreover recursion can be used to get recursive records:

```
Seasons= μ(rec)   {   spring=   {name="spring"   next= rec.summer}
                      summer=   {name="summer"   next= rec.autumn}
                      autumn=   {name="autumn"   next= rec.winter}
                      winter=   {name="winter"   next= rec.spring}
                  }
```

so for example Seasons.autumn.next.next.name yields "spring". Seasons can be seen as a recursive record, but also as a memory with four locations. Expressions like rec.summer work as "pointers" in the memory fixed by Seasons. Here we have a flat space of memory locations, but the approach can be easily extended to define hierarchical memory spaces with corresponding fixed-point operations at different levels. Pointers in the hierarchical space simply would use variables with different indices (using the '\' syntax).

6.3.6 Updatable Records (Memories)

The next step is to define updatable records, or, seen differently, writable memories. This can be done using the previous constructs, as pictured in figure 6.8. An updatable record is a recursive function, with one named parameter for each field; internally it consists of a simple record with a *get* field, which returns the internal values, and a *set* field, which re-

turns a record of update functions. An update function for field x_i takes one argument *arg*, and uses recursion to return the same updatable record, in which all fields are bound to their current values except the one being updated which takes the new value. Updating a record consists of selecting the appropriate update function, and binding the new value to its *arg* parameter. Functions using this encoding are naturally polymorphic: the function

ZeroX = λ(aRecord) aRecord⟨x := 0⟩

can be applied to *any* record containing an *x* field and returns the original record, with only field *x* being updated.

Updatable records give full flexibility for modelling local state of objects and object identifiers. In languages using a flat domain of object identifiers, like Smalltalk or Objective-C, each object would have its own updatable record, representing local state, and then all objects would be stored in a global record, representing the space of object identifiers. Some other languages have a more complex structure: for example in C++, an object can be contained in the memory space of another object (so the implementation structure reflects the "has-a" relationship). Modelling such structures in λN would involve hierarchical updatable records, in which some fields contain sub-records.

6.3.7 Field Overwriting

The encoding presented in the previous subsection supports modification of an existing field, but not addition of new fields. An alternative approach to updatable records is to consider field overwriting. Here is how it can be done:

r[x←a] = λ(sel) r(sel→sel(x→a))!

This creates a new record from *r* in which field *x* has value *a*, whether or not *x* was already present in *r*. Observe that the encoding is based on the fact that the selector received as a parameter is immediately bound to *a* on name *x*, *without a close operation*, before being passed to the record *r*. This explains why any binding on *x* in *r* will be ignored. Given a field overwriting operation, it is possible to implement record concatenation "for free", following Rémy's technique [26]: one would start with an empty record

λ(sel) sel!

and then consider each record as a "record-modifying function", adding the desired fields; such functions can be combined by functional composition.

6.4 Compatibility Relationship

Several examples of extensible and reusable constructs have been shown, but so far we have no formal definition of a compatibility relationship. In this section such a relationship is studied, through an observational classification of λN expressions. In the standard lambda calculus, the only observable property of terms is their termination behaviour:

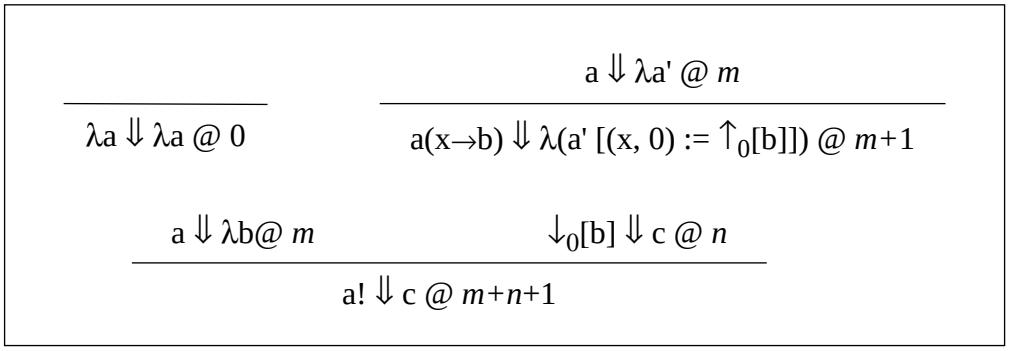


Figure 6.9 *Convergence to weak normal form.*

errors never occur, since all values are functions. Here, we have seen that errors can be generated during a computation, and therefore errors also represent a valuable observation. So, as a complement to the usual approximation ordering, which compares terms on the basis of convergence, we also consider a compatibility ordering, comparing terms on the basis of error generation. This section is mainly inspired from operational orderings in Scott Smith's work [28], who himself draws from a vast body of literature on observational relations (see for example [20][1]). However, Smith identifies errors with divergence, whereas we treat them as distinct observations.

6.4.1 Errors and Lazy Operational Semantics

Now it is time to justify our encoding of errors, as it was given in figure 6.4. The complex expression defining **err** could be written, in high-level notation, as $\mu(x)\ \lambda()\ x$, i.e. as an abstraction without any parameters, containing itself. Such a term can consume any sequence of bind or close operations, but always reduces back to itself. In a classical lambda calculus, a similar behaviour is displayed by the term

$$(\lambda x.\lambda y.xx)(\lambda x.\lambda y.xx)$$

which consumes any input without ever using it. Under a usual interpretation, this is just identified with the bottom element (divergence); however, in a lazy interpretation, it becomes the top element. Boudol [7] calls this an "ogre", while Abramsky and Ong [1] say "a term of order ∞". Usually the "ogre" is not considered very interesting, because it does not interact with its environment. However, this is precisely the behaviour of a run-time error: once it occurs, the "continuation" of the program is ignored, and the final result is the error. So the ogre is a natural choice for representing run-time errors. In consequence, we define in figure 6.9 a lazy convergence relation, where $a{\Downarrow}b\ @\ m$ means "a converges to b in m steps of computation". We simply write $a{\Downarrow}$ if there are a', m such that $a \Downarrow a'\ @\ m$, and $a{\Uparrow}$ if $\neg(a{\Downarrow})$.

Definition 14 A term *a* is *erroneous* (written *a*?) iff it converges and any binding or close operation on it yields an erroneous term again. Formally:

$$a? \quad \Leftrightarrow \quad a\Downarrow \text{ and } (a!)? \text{ and } \forall b.\ (a(x{\to}b))?$$

Another way to state this is to say that *a* is erroneous iff $\forall\overline{o}$, $a\overline{o}\Downarrow$, where $\overline{o}$ is a sequence of bind or close operations. We write $a¿$ whenever $\neg(a?)$. It is an easy exercise to check that (**err**?).

6.4.2 Approximation and Compatibility

Definition 15 The *approximation ordering*, written $\leq_\perp$, is

$$a \leq_\perp b \quad \Leftrightarrow \quad \forall C[-].\ C[a]\Downarrow \Rightarrow C[b]\Downarrow$$

where a *context* C[–] is a term with "holes", which can be filled by another term *a* through the context-filling operation C[a].

Definition 16 The *compatibility ordering*, written $\leq_{\mathbf{err}}$, is

$$a \leq_{\mathbf{err}} b \quad \Leftrightarrow \quad \forall C[-].\ C[b]¿ \Rightarrow C[a]¿$$

Observe that here *a* and *b* are in reverse order in the implication. The first preorder states that whenever *a* converges, *b* also converges. The second preorder states that whenever *b* does not generate an error, *a* does not either. It may seem strange that these definitions are in opposite directions, but this corresponds to standard practice in semantic domains and subtype orderings. In semantic domains, the least defined element (representing the divergent program) is at the bottom, and more defined elements are higher up in the ordering. In type systems, the least defined type (type of anything) is usually at the top, and more refined types are lower. It can be checked, for example, that $\mathsf{Not}_U \leq_{\mathbf{err}} \mathsf{Not}$, i.e. our extended version of the *not* operation for a three-valued logic, is indeed compatible with the *not* operation on Boolean values only.

In [14] we have defined similar orderings for a pure lambda calculus with records (but without extensible records), and we have shown that both orderings coincide, i.e. approxima and compatibility are the same when **err** is chosen as the top element. The proof can be transposed to λN without difficulty. So we have a formal framework for reasoning not only about equivalence of software components, as in usual semantics, but also about their plug-compatibility relationships. Some consequences of this result are discussed in the rest of this section.

6.4.3 Lattice Structure

Define $\bot = \mu(x)\ x$. This is the divergent term [observe the difference with **err** = $\mu(x)\ \lambda()\ x$]. $\bot$ is smaller than any term: a divergent term never generates an error, and never reduces to a WNF in any relevant context. On the other hand, **err** is a greatest element in both orderings, since it never diverges and is an error. This implies that the order is a *lattice* with top element **err** and bottom element $\bot$.

The fact that we get a lattice is interesting in many respects. Lattices were originally considered by Scott for solving domain equations. Then the presence of a top element was criticized, in particular by Plotkin [25], because this element fails to satisfy some intuitively natural identities about the conditional function: for example we expect a phrase like

if *a* **then** *b* **else** *c*

always to give either *b* or *c*; however, this does not hold when *a* is the top element, and it is not clear then what the answer should be: it could be TOP itself, or it could be the upper bound of *b* and *c*, but none of these solutions seems to make sense in usual interpretations. Therefore the semantics community moved to algebraic CPO models instead of lattices.

Since our approach is purely operational, there is no reason here to argue for or against a particular model. Nevertheless, it is worth noticing that the operational lattice has some natural properties. In particular, interpreting the top element as an error, it is quite natural that we should have

if err then *b* **else** *c* = **err**

The answer is neither *b* nor *c*, but this does not contradict our intuitive understanding of the conditional statement: if the first argument is an error, then the whole statement produces an error.

A more recent discussion about lattice models was written by Bloom [6], partially based on Plotkin's previous work. Bloom supports the view that, despite the fact that lattices are mathematically more tractable than CPOs, they have several defects when used as models for programming languages. One of his main criticisms to lattice models is that they are not single-valued: for example if we choose the second solution for the conditional statement above, namely

if TOP **then** *b* **else** *c* = $b \sqcup c$

we get the upper bound of *b* and *c*, which, if not TOP itself, is a "multiple value". However, the justification for taking single-valuedness as an essential criterion is not strongly established. Therefore Boudol [7] criticizes Bloom's position, and argues that under a different notion of observation, multiple values make perfect sense. Parallel functions in Boudol's paper yield a lattice model. Similarly, powerdomains used for modelling concurrency also have a lattice structure. These observations lead us to another extension of the calculus which completes the operational structure by introducing all meets and joins. Full development of these constructs would go beyond the scope of this paper; however, a brief appetizer will be given.

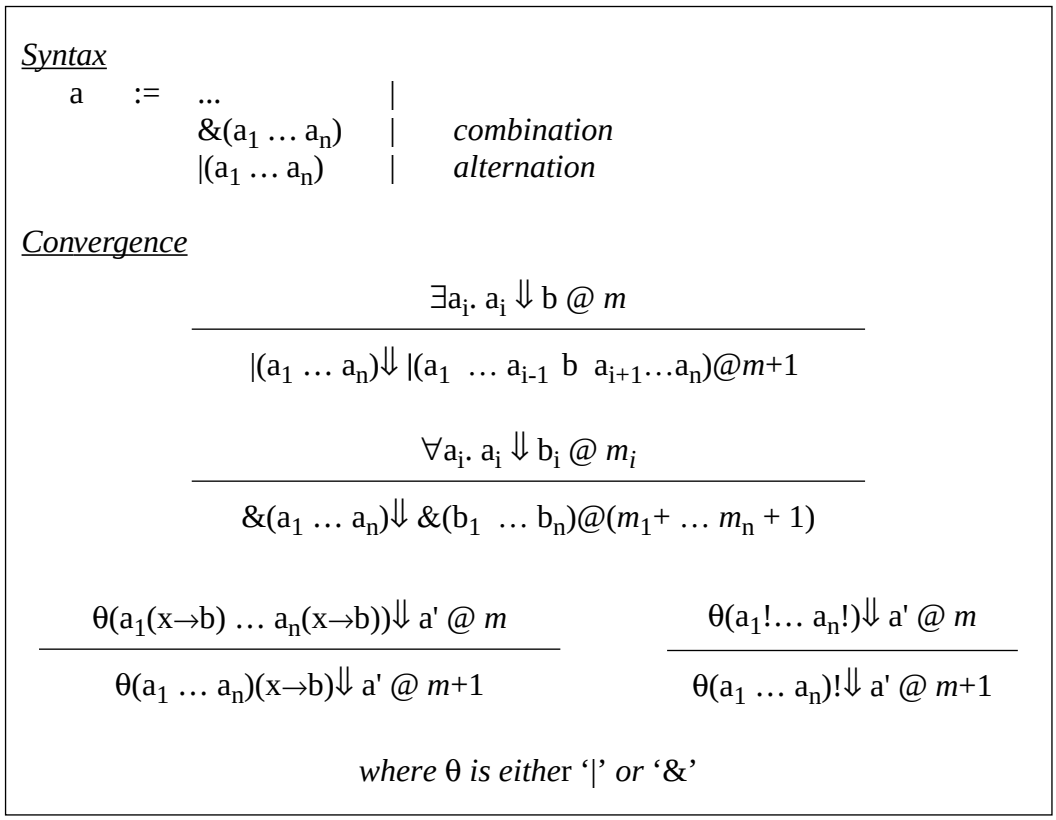


Figure 6.10 *Combinations and alternations.*

6.4.4 Meets and Joins

Figure 6.10 introduces two *n*-ary constructs called *combination* and *alternation*. The reduction rules are exactly the same for both: any binding or close operation is simply distributed to the internal members. Therefore they can be seen as an array of non-communicating processors accepting common operations, in a kind of SIMD architecture. The difference between combinations and alternations comes observationally from the definition of convergence: combinations converge if all their members converge, while alternations converge if at least one member converges. Since convergence is at the foundation of our approximation/compatibility relationship, we have the following properties:

- The combination is a *glb* (greatest lower bound, meet) operator.
- The alternation is a *lub* (least upper bound, join) operator.

This has many interesting applications, all related to various possible uses of *sets* of values.

The alternation operator can be interpreted to model non-determinism. A very similar proposal has been made by Boudol under the name *parallel functions* [7]. Boudol mainly discusses the use of parallel functions for solving the full abstraction problem (relating the

operational ordering with the semantic ordering). Another application is concurrency modelling, where all possible outcomes of a computation are grouped together in an alternation, on which further processes can compute: in [13] we discuss an encoding of shared memory, processes and synchronization primitives using alternations. Yet another possibility is to interpret an alternation as a type, "containing" all its member terms. This opens very interesting perspectives for typing, since the notions of type membership and subtype relationship are both captured by the approximation/compatibility ordering, and therefore values and types are merged into one single concept. Finally, since we deal with sets of values we can directly apply Scott Smith's results [28] for proving theorems like fixed-point induction in a purely operational setting, without going to semantic domains.

Applications of the combination construct, which in a sense is an "overdeterministic" operator, are less intuitive. Remembering that **err** is the top element, combinations can be used to remove errors in a computation, by taking the lower bound of a set of values. This can be applied for operations such as *record concatenation* [10][17]. Moreover, following the idea of unifying types and values, combinations have the same properties as *intersection types*[4][24]. Interestingly, a connection between record concatenation and intersection types as also been proposed by John Reynolds in his Forsythe language[27].

6.5 Conclusion

A lambda calculus with name-based interaction has been described. A few systems using similar ideas have been mentioned in the introduction [19][16]; the original aspect of λN is the unification of names with variables through the use of de Bruijn indices. Not only is this more practical; it also allows us to directly import most of the results established for the standard lambda calculus. Extensible functions in λN are a good basis for studying reusability mechanisms (in particular inheritance and subtyping), and the economy of constructs compares advantageously to other approaches based on records ([9][17]) or extensible methods [23].

The other extension (alternations and combinations) is perhaps more venturing. It touches several hot research areas, like observational equivalences and full abstraction for lambda models [1], parallel functions [7], extensible records [17], and semantics of concurrency. Most of these issues require further investigation. An exciting challenge is to see how the π-calculus[21], also based on names, relates to λN.

The issue of typing was mentioned very briefly, and the development of a full type theory for the calculus is under investigation [13][15]. Using the term ordering as a semantic basis for types seems a promising direction, and has some similarities with type theories based on the Curry–Howard isomorphism (identification of types with logical propositions)[29], in which the usual distinction between terms and types is also blurred. Including name-based interaction in such theories would be a promising step towards an object-oriented logic, and would relate to what Aït-Kaci calls *features* [2]. Related to this, the term ordering in λN can be useful for object-oriented databases, since it gives a query language for free!

Apart from those foundational issues, there are several practical directions in which this work can be extended. One, which in fact was the original motivation for developing the calculus, is to use it for explaining the differences between various forms of inheritance and delegation in object-oriented languages. In addition, many other aspects of programming languages, like modularity, state manipulation or restricted islands of memory locations [18] can be studied in this framework. Ultimately, it is of course tempting to build higher-level syntactic constructs on top of the calculus and make it a full programming language integrating these various aspects.

Finally, it is worth considering implementation issues for this calculus, and perhaps to design a name-based abstract functional machine. As noted by Garrigue [16], names can be translated into offsets in a machine implementation; however, their combination with de Bruijn indices probably raises some technical problems. Combinations and alternations are more challenging. Evaluating a combination can be done by sequentially evaluating all of its members, but evaluating an alternation must be done in some form of parallelism, to be consistent with our notion of WNF.

Acknowledgments

I am grateful to Oscar Nierstrasz, who gave much of his time for examining this work, to Benjamin Pierce, Christian Breiteneder, John Lamping, and Jacques Garrigue, who commented earlier versions, and to Patrick Varone, who read this version carefully and helped to correct several points.

References

[1] Samson Abramsky and C.-H. L. Ong, "Full Abstraction in the Lazy Lambda Calcul," *Information and Computation*, vol. 105, 1993, pp. 159–267.

[2] Hassan Aït-Kaci and Andreas Podelski, "Towards a Meaning of LIFE," *Proceedings PLILP '91, Lecture Notes in Computer Science*, vol. 528, Springer-Verlag, 1991, pp. 255–274.

[3] Hassan Aït-Kaci and Jacques Garrigue, "Label-Selective λ-Calculus, Syntax and Confluence," *Proceedings 13th International Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, vol. 761, Springer-Verlag, 1993, pp. 24–40.

[4] Franco Barbanera and Mariangiola Dezani-Ciancaglini, "Intersection and Union Types," *Proceedings Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science*, vol. 526, Springer-Verlag, 1991, pp. 651–674.

[5] H. P. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1985 (2nd printing).

[6] Bard Bloom, "Can LCF Be Topped? Flat Lattice Models of Typed λ-Calculus," *Information and Computation*, vol. 87, 1990, pp. 264–301.

[7] Gérard Boudol, "Lambda-Calculi for (Strict) Parallel Functions," *Information and Computation*, vol. 108, 1994, pp. 51–127.

[8] N. de Bruijn, "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation," *Indag. Mat.*, vol. 34, 1972, pp. 381–392.

[9] Luca Cardelli, "A Semantics of Multiple Inheritance," *Information and Computation*, vol. 76, 1988, pp. 138–164.

[10] Luca Cardelli and John C. Mitchell, "Operations on Records," *Proceedings Conference on Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science*, vol. 442, Springer-Verlag, 1989, pp. 22–52.

[11] Thierry Coquand and Gérard Huet, "The Calculus of Constructions," *Information and Computation*, vol. 76, 1988, pp. 95–120.

[12] Pierre-Louis Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming*, 2nd edn., Birkhäuser, Boston, 1993.

[13] Laurent Dami, "Software Composition: Towards and Integration of Functional and Object-Oriented Approaches," Ph.D. Thesis, University of Geneva, 1994.

[14] Laurent Dami, "Pure Lambda Calculus with Records: from Compatibility to Subtyping," working paper, 1994.

[15] Laurent Dami, "Type Inference for λN, and Principal Type Schemes for Record Concatenation," working paper, 1994.

[16] Jacques Garrigue and Hassan Aït-Kaci, "The Typed Polymorphic Label-Selective λ-Calculus," *Proceedings 21st ACM Symposium on Principles of Programming Languages*, 1994, pp. 35–47.

[17] Robert Harper and Benjamin Pierce, "A Record Calculus Based on Symmetric Concatenation," *Proceedings 18th ACM Symposium on Principles of Programming Languages*, ACM Press, 1990, pp. 131–142.

[18] John Hogg, "Islands: Aliasing Protection In Object-Oriented Languages," *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, vol. 26, no. 11, Nov. 1991, pp. 271–285.

[19] John Lamping, "A Unified System of Parameterization for Programming Languages," *Proceedings ACM Conference on Lisp and Functional Programming*, 1988, pp. 316–326.

[20] Robin Milner, "Fully Abstract Models of Typed λ-Calculi," *Theoretical Computer Science*, vol. 4, 1977, pp. 1–22.

[21] Robin Milner, "The Polyadic π-Calculus: A Tutorial," Tech. Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[22] Robin Milner, "Elements of Interaction," (Turing Award Lecture), *Communications of the ACM*, vol. 36, no. 1, Jan. 1993, pp. 78–89.

[23] John C. Mitchell, Furio Honsell, Kathleen Fisher, "A Lambda Calculus of Objects and Method Specialization," *Proceedings 8th Annual IEEE Symposium on Logic in Computer Science*, 1993.

[24] Benjamin C. Pierce, "Intersection Types and Bounded Polymorphism," *Proceedings Conference on Typed lambda-calculi and Applications*," *Lecture Notes in Computer Science*, vol. 664, Springer-Verlag, March 1993, pp. 346–360.

[25] Gordon Plotkin, "Domains," Course Notes, Department of Computer Science, University of Edinburgh, 1983.

[26] Didier Rémy, "Typing Record Concatenation for Free," *Proceedings ACM POPL'92*, ACM Press, 1992, pp. 166–176.

[27] John C. Reynolds, "Preliminary Design of the Programming Language Forsythe," Technical Report CMU-CS-88-159, Carnegie-Mellon University, 1988.

[28] Scott F. Smith, "From Operational to Denotational Semantics," *Proceedings Conf on Mathematical foundations of Programming Semantics, Lecture Notes in Computer Science*, vol. 598, Springer-Verlag, 1992, pp. 54–76.

[29] Simon Thompson, *Type Theory and Functional Programming*, International Computer Science Series, Addison-Wesley, Reading, Mass., 1991.