Doctor Thesis

LABEL-SELECTIVE LAMBDA-CALCULI AND TRANSFORMATION CALCULI

Submitted in December 1994

The University of Tokyo, Doctoral School of Science, Department of Information Science

GARRIGUE Jacques

ABSTRACT

The label-selective lambda-calculus, in its different variants, and its offspring, the transformation calculus, are the results of a research on the role of Currying in the lambda calculus.

Currying is the simple trick by which functions of multiple arguments can be written in the lambda calculus, which is essentialy mono-argument. The idea is to transform a function on a pair, into a function whose result, once applied to its first argument, must be applied to its second one. That is $f(a,b) = (\overline{f}(a))(b)$.

In our first system, the label-selective lambda-calculus, we give a method to curry starting from a labeled record, in place of a simple pair. The calculus we encode in has to be more complex than simple lambda-calculus, because of these labels, but it appears to keep the quasi totality of its properties. We have of course confluence, and models similar to lambda calculus; and we can apply both simple and polymorphic typings, for which we get strong normalization, as we had with lambda calculus. An immediate application of such a system is to add out-of-order labeled application to curried functional languages, like ML. Since Currying introduces partial applications, we combinatorically augment the number of possible partial applications.

The second extension, which results in the transformation calculus, introduces the idea of a Currying-compliant composition of terms. Classical functional composition is incompatible with Currying, but ours is, by intuitively making multiple connections at once. Again this system is confluent, and we defined typings for it. Selective lambda-calculus is already included, but the new power of this calculus is in viewing arguments as a flow of data, which can be manipulated by composed *transformations*. It allows one to write imperative algorithms, changing variables, in a purely functional framework. By this it provides a syntactical view on previous works done on the semantics of Algol, and other imperative extensions of functional languages.

Finally we present a third calculus, that of symmetrical transformations, which extends the transformation calculus to n-ary relations. This can give interesting insights on problems like distributed computation or, more mathematically, function inversion.

These three systems are studied from the three points of view of their syntactial properties, their typing, and their semantics.

Aknowledgements

I shall thank here all those who, by their collaboration, help, understanding, or simple interest, permited me to write this thesis.

In the first place come of course my supervisors. Hassan Ait Kaci, for my D.E.A. in Universite Paris VII – Denis Diderot, was the origin of part of the ideas of this thesis, and label-selective λ-calculus started as a collaborative work with him. Akinori Yonezawa, supervisor of this thesis, gave me his confidence and support to pursue this work.

Mitsuhiro Okada, one of the first to show interest in the ideas exposed here, kindly advised me for the tricky proof of confluence, and checked it with all his knowledge of the field.

Atsushi Ohori was, during these three years, an always available interlocutor to discuss the progression of sometimes strange ideas. I received a lot in the interaction with him. He also helped me in giving better formalizations to some concepts. I am specially grateful to him.

Christophe Bonnet was there at the beginning of many of the ideas presented here, and his constant interest and comments were a support.

I thank also all the members of the Yonezawa laboratory who, eventhough their specialties were different, offered me fruitful exchanges. I think particularly of Jeff MacAffer, Hidehiko Masuhara and Satoshi Matsuoka, who introduced me to object-oriented philosophy and concurrent programming, and Kentaro Torisawa, for other philosophical discussions on language. Masami Hagiya, Shin'ya Nishizaki and Daisuke Suzuki of the Hagiya laboratory were good interlocutors for more specialized questions.

Masako Takahashi, Therese Hardin, Sundeep Oberoi, Guy Cousineau, Pierre-Louis Curien, Jean-Jacques Levy and Giuseppe Longo also showed interest in this work, and their comments were precious.

Last, I thank Denis Diderot, author of "Jacques le fataliste et son maitre", a book which gave me a way of thinking, following tracks just by fantasy, and not trying to contradict fate.

I am forgetting many others, of this world or the other, who gave me hints and comfort. Let them all be thanked too.

Table of Contents

1. Introduction **1**

1.1 Implicit currying . . . 1
1.2 Selective λ-calculus . . . 2
1.3 Transformation calculus . . . 3
1.4 Symmetrical transformations . . . 5
1.5 Organization of this thesis . . . 5

2. Selective λ-calculus : Syntax **7**

2.1 Introducing selective λ-calculi . . . 7
2.1.1 Generic syntax . . . 7
2.1.2 Relative and absolute positions . . . 7
2.1.3 A lambda-calculus with multiple channels . . . 9
2.1.4 A lambda-calculus with moving indexes . . . 10
2.1.5 A first way to combine these two systems . . . 11
2.2 The selective λ-calculus . . . 11
2.2.1 Definition . . . 11
2.2.1.1 Syntax . . . 11
2.2.1.2 Substitutions . . . 12
2.2.1.3 Reductions . . . 12
2.2.2 Entity syntax . . . 12
2.3 Proof of confluence . . . 14
2.3.1 Pseudo-reduction . . . 14
2.3.1.1 Rules . . . 14
2.3.1.2 Pseudo-reduced form equivalence (PRF) . . . 14
2.3.2 Combined systems and restricted reductions . . . 15
2.3.3 Confluence of the reordering system . . . 15
2.3.4 Confluence of β-reordering . . . 19
2.3.5 Confluence of selective λ-calculus . . . 25

3. Streams **28**

3.1 Stream monoid . . . 28
3.2 Stream syntax of selective λ-calculus . . . 31
3.2.1 Terms . . . 31
3.2.2 β-reduction . . . 31
3.2.3 η-reduction . . . 32
3.2.4 Bohm separation theorem . . . 32

4. Typed selective λ-calculus **34**
4.1 Simple types . . . 34
4.2 Polymorphic selective λ-calculus . . . 36
4.2.1 Syntax and types . . . 37
4.2.2 Typing rules . . . 37
4.2.3 Type unification . . . 38
4.2.4 Type inference . . . 40
4.3 Application to programming . . . 41
4.3.1 Keywords: an enhancement for clarity . . . 41
4.3.2 Relative positions versus combinators . . . 42

5. Semantics **43**
5.1 Model of the selective λ-calculus . . . 43
5.2 Typed model . . . 44
5.2.1 Definition . . . 44
5.2.2 Construction of a model . . . 44
5.2.3 Correctness of this model . . . 45
5.3 Multi-application model . . . 46
5.4 Abstraction models . . . 46
5.4.1 Definition . . . 46
5.4.2 Properties . . . 47
5.4.3 Restricted abstraction model . . . 48
5.5 Categorical models . . . 48
5.5.1 Label Category . . . 48
5.5.2 Record category . . . 50
5.5.3 Record closed category . . . 51
5.5.4 Multi-categorical model . . . 52
5.5.5 Mono-categorical model . . . 53
5.6 Final remarks on semantics . . . 55

6. The transformation calculus **56**
6.1 Introduction . . . 56
6.2 Composition and streams . . . 57
6.2.1 Implicit currying . . . 57
6.2.2 Composition . . . 58
6.2.3 Selective currying . . . 59
6.2.3.1 Indexed streams . . . 59
6.2.3.2 Naming positions . . . 60
6.2.4 Stream behaviour . . . 61
6.2.5 Scope-free variables . . . 62
6.3 Syntax of transformation calculus . . . 64
6.4 Applicative translation and confluence . . . 66
6.5 Scope-free variable encoding . . . 67
6.6 Transformational combinators . . . 69
6.6.1 First formulation : $A^1U^2C^2D^2K^1$ . . . 69
6.6.2 Second formulation : $A^1K^1U^2S^2$ . . . 69

7. Typed transformation caculus and extensions **71**
7.1 Simply typed transformation calculus . . . 71
7.2 Denotational semantics . . . 74
7.3 Polymorphic types . . . 75
7.3.1 Syntax and types . . . 76
7.3.2 Typing rules . . . 77
7.3.3 Monotype unification . . . 79
7.3.4 Type inference . . . 80
7.3.5 Polymorphic records . . . 82
7.4 FIML: a typed language based on the transformation calculus . . . 82
7.5 Transformations and stateful objects . . . 82
7.5.1 The scoped transformation calculus . . . 82
7.5.1.1 Syntax . . . 83
7.5.1.2 Simple types . . . 84
7.5.2 Stateful objects . . . 84

8. Symmetric Transformations **88**
8.1 Logical version . . . 88
8.1.1 Grammar . . . 88
8.1.2 Structural rules . . . 89
8.1.3 Reduction rules . . . 90
8.1.4 Lambda calculus . . . 91
8.1.5 Types . . . 92
8.1.6 Denotational semantics . . . 93
8.2 Functional version . . . 95
8.2.1 Syntax . . . 95
8.2.2 Typed functional relations . . . 96
8.2.3 Generalized inversion . . . 96

9. Conclusion **98**
9.1 Applications . . . 98
9.2 Related works . . . 99
9.2.1 Selective λ-calculus . . . 99
9.2.2 Transformation calculus . . . 100
9.3 Future works . . . 101

Appendix

A. Other results **103**
A.1 Applicative polymorphism . . . 103
A.1.1 Terms and types . . . 103
A.1.2 Let polymorphism . . . 103
A.1.3 Applicative polymorphism . . . 104

B. FIML **107**
B.1 FIML V0.4 Manual . . . 107
B.1.1 Installation . . . 107
B.1.2 Syntax . . . 108
B.1.3 Toplevel . . . 109
B.1.4 Primitives and pragmas . . . 110

B.1.5 Error handling . . . 110
B.1.6 Programming examples . . . 110
B.1.7 Bugs and future developments . . . 111
B.2 Sample FIML session . . . 111
B.2.1 Starting with it . . . 111
B.2.2 New notations . . . 112
B.2.3 The dialogue mode . . . 114
B.2.4 Imperative features . . . 115
B.2.5 Final remarks . . . 116
B.3 Programming samples . . . 116
B.3.1 prelude.fm . . . 116
B.3.2 prelude2.fm . . . 117
B.3.3 unif.fm . . . 117
B.3.4 examples.fm . . . 118
B.3.5 types.fm . . . 119
B.3.6 combinators.fm . . . 119
B.3.7 graph.fm . . . 120

References **121**

Chapter 1

Introduction

This thesis is — from its origin to its achievement — a study of a "phenomenon" of lambda calculus, *Currying*[1]. It is, probably, the first one to make it its main subject, since this phenomenon was never paid much attention. Currying takes its origin in the necessity of expressing multi-argument function in the lambda calculus — or in its algebraic version, combinatory logic — which is mono-argument. The trick is simple: make a function that takes the first argument and gives back a function takes the second argument and gives back *etc...* to the final result. With only two argument that is $f_u(x,y) = (f_c(x))(y)$.

The principal reason to this poor interest, whereas it is widely used, is the apparent extreme simplicity of the conversion. Why study such a straightforward "trick". However we will see that, while intuitively evident in the case of tuples, the problem gets more complex in practice, or with extended data structures. A simple example is the impossibility of writing a function doing this transformation (in the general n-ary case) in ML[2].

Another remark is that most works which can be related to it are not explicitly so. For instance, Curien's *Categorical Combinators* [Cur93] contain a currying combinator Λ. More recently, Dami proposed a partly curried calculus of records [Dam94]. But none of them consider currying as a real object of study.

We have done it in three ways, by extending its role.

1. A study of currying in the presence of *selective* arguments, that is currying *records* in place of tuples. (Chapters 2 to 5)

2. A study of output currying, that is currying not only the input side of functions, but they result too. (Chapters 6 and 7)

3. A study of currying on n-ary relations, which is rather currying the two-sides of binary relation. (Chapter 8)

1.1 Implicit currying

All the systems we develop here are based on a slight reformulation of currying, which makes easier to relate intuition and reductions, we call it *implicit currying*. The idea is just to describe currying not as a transformation but as an equality, which lets us work with the uncurried form of expressions.

[1] This name is subject to discussion since Curry himself, who defined the concept for combinatory logic [Cur30], remarked in [Cur80] that Schonfinkel, an other father of λ-calculus, had the idea before him [Sch24].

[2] Recently a solution was proposed for an extended type system [DRW95]

In the usual case it just means that we allow to write

$$(\lambda(x_1, \ldots, x_n).M)(M_1, \ldots, M_m)$$

for

$$(\lambda x_1.\cdots.\lambda x_n.M)\ M_1\ \ldots\ M_m.$$

The extension of this notation to records, and the problems it incurs, is the starting point of our study.

1.2 Selective λ-calculus

In the beginning, a simple remark : in λ-calculus currying yields a privileged order of application on the arguments.

$$(A \times B) \to C \simeq A \to (B \to C)$$

That means that applying $f : A \times B \to C$ to its second argument necessitates an asymmetrical use of combinators :

$$\lambda x.f(a, x) = f\ a\ ^{3}$$

but

$$\lambda x.f(x, b) = S\ f\ (K\ b)\ ^{4}$$

This is linked with another problem : since the only way left to specify an argument is to know its position in the application sequence we must always keep in mind in which order we should do the application.

An ideal solution would be to solve these two problems at once, by introducing a notion that simultaneously abstracts application order and lets arguments commute, that is introduces the isomorphism

$$(A \times B) \to C \simeq (B \times A) \to C$$

aka

$$A \to (B \times C) \simeq B \to (A \to C).$$

Finally we should add a condition, which eliminates the trivial solution of not distinguishing parameters, as for instance in Berry's and Boudol's Chemical Abstract Machine [BB90], by requiring determinism.

Our answer to this problem is the use of labels. Most programming languages have labeled records. This seems a necessity to handle complex data structures. Some of them, including Common LISP [Ste84], ADA [Led81], and LIFE [AKG93b], extend this possibility to parameters of functions. However, if we want mix this last possibility with currying, problems arise.

Our strategy is to define this new possibility of labeling arguments in the presence of currying through commutation equalities on these arguments. The roles of our arguments are determined by their labels, which interact with their order. Selective λ-calculus introduces two types of commutation equalities.

[3] by η-equality $\lambda x.g\ x = g$

[4] since $S\ f\ (K\ b)\ x = f\ x(K\ b\ x) = f\ x\ b)$

The first, and most natural one, is between symbolic labels. By analogy with tuples, when currying a function $f_u\{p \Rightarrow a, q \Rightarrow b, \ldots\}$ we obtain a function $((f_c\{p \Rightarrow a\})\{q \Rightarrow b\})\ldots$. But since we have no reason to apply f_c in this specific order, we want to use the abstraction provided by labels and be able to write $((f_c\{q \Rightarrow b\})\{p \Rightarrow a\})\ldots$ too. Suppressing superfluous parentheses, and limiting ourselves to two arguments, we will need in our calculus the equality:

$$f\{p \Rightarrow a\}\{q \Rightarrow b\} = f\{q \Rightarrow b\}\{p \Rightarrow a\}.$$

However we must had a restriction to this: p and q should be different labels. By speaking of currying, we are thinking of languages where functions are first-class objects; so this may be the case that we have to apply a function more than once on the same label, and we should keep the order of these applications decidable.

Here is an example of the use of these symbolic labels for the list constructor, in an ML-like language [5] , together with inferred types.

```
#let cons car=>a cdr=>b = a::b;;
 cons : {car=>'a,cdr=>'a list} -> 'a list

#cons cdr=>[1];;
 it : {car=>int} -> int list
```

The second type of equality comes from a reversion of the analogy with tuples. That is, we can see a tuple as a record labeled with numbers: $(a, b, \ldots) = \{1 \Rightarrow a, 2 \Rightarrow b, \ldots\}$. If we apply directly the equality above we would have $f\{1 \Rightarrow a\}\{2 \Rightarrow b\} = f\{2 \Rightarrow b\}\{1 \Rightarrow a\}$. But, based on the idea that we can abbreviate the label 1, and the current definition of currying, we would rather write $f\{1 \Rightarrow a\}\{1 \Rightarrow b\}$, or, abbreviating 1, $f\ a\ b$. To make it possible, we will define commutations differently on numbers, and have $f\{1 \Rightarrow a\}\{1 \Rightarrow b\} = f\{2 \Rightarrow b\}\{1 \Rightarrow a\}$. This can be generalized in:

$$f\{m \Rightarrow a\}\{n \Rightarrow b\} = f\{n+1 \Rightarrow a\}\{m \Rightarrow b\} \qquad \text{if } m \leq n.$$

For instance we can use it as follows, omitting `1=>`.

```
#let sub x y = x-y;;
 sub : {1=>int,2=>int} -> int

#let minus15 = sub 2=>15;;
 minus15 : {1=>int} -> int
```

This second equality is in fact orthogonal to the first one. Commutation on symbolic labels expresses the intuitive possibility of taking input on multiple *channels*, while the numeric form gives a control on the order on each *channel*.

1.3 Transformation calculus

A natural extension to selective λ-calculus is applying currying to the output of functions. The idea takes its origin in the definition of composition in selective λ-calculus. Since we have labels, each time we compose two functions we must know from which label each of the functions takes its input. That is $f \overset{p,q,r}{\circ} g = f\,\widehat{p}\,(g\,\widehat{q}\,x)\ \overset{\vee}{r}\ x$, the function taking its

[5] We use a notation close to CAML [W+90]. "`let`" denotes a definition, "`::`" the list constructor. Since "`=>`" is left unused (abstraction uses "`->`"), we use it for labeling.

input on r, giving it to g on q, and feeding the result to f on p. This is complex, and this is rather weak. Particularly when we think of the powerful out-of-order currying power of our system.

The solution is have labels on the output: if know that $g\{q \Rightarrow x\}$ returns output on p, then we can simply write $f \circ g$. Moreover, introducing currying here means that if f gets input on labels p, q, r and g returns output on the same labels, in $f \circ g$ all connections are done. And even more: if f needs more input, resp. g returns more output, then those can be reported to the input side of g, resp. the output side of f.

This principle appears to be quite powerful. Not only it extends record calculi with a certain form of concatenation/composition, but it introduces the notion of *transformation*, that is functions with labeled output. Such transformations can be composed together in a way that describes changes on a mutable state combined with functional computations.

The notion of mutable state is essential in many algorithms. There are ways to turn it, with recursion and infinite lists, but this is not always very natural. A more intuitive notion of state is helpful; we can see it even in mathematics, where algorithms are often defined imperatively. Lambda-calculus is enough to define functions, but not always practical for algorithms.

One way to use a state in lambda-calculus with pairing is simply considering this state as a normal value we pass to functions and get back from them. This is the intuition you have in "folding" functions: iterating a function on a list to get a final result. However such a method presents a problem. We must know the structure of our state when accessing it, and there is no standard way of combining two pieces of state into one. Particularly, this makes difficult to get "state polymorphic" functions, that would be able to change some defined parameter in a range of differently structured states. This means that we must do all the scoping by hand, sending only necessary parts of the state to each function, and recombining the result.

A natural answer to this problem is the use of mutable variables, as we have in imperative languages. A first way to do it, by directly using a store, like references in ML, breaks referential transparency, and loses confluence. Some formal systems introduce them more cautiously [Lam88, ORH93, CO94, Sat94]. These systems include *named abstractions* for reading the value of such a variable, along with *named definitions* providing destructive assignments. These two operations are orthogonal to λ-calculus' β-reduction. What they essentially do is forcing the operations on the store to be sequential, but the intuition is still there. With that we can easily get functions modifying only one variable in a state, and do not need to know about its complete structure. The scoping of variables can be done by simple name scoping. Advantages of this method are its closeness to the classical computational model, and possibilities of specific typing of the store through effects [GL86, TJ92].

We find two defaults to stores.

- The extension is completely orthogonal to λ-calculus, we have some redundant concepts between the two, like reading a variable in the store and getting a value through an abstraction.
- It is linked with a particular model including memory, and we may want a stronger abstraction if we are to work with new computational models.

Our transformations are an answer to these two problems. They are integrated into the core of the calculus, and do not suppose any computational model. In fact, the notion of *scope-free variable* they define is stronger than classical stores, and have

other potential applications. Moreover, denotational semantics of scope-free variable is, thanks to this integration, straightforward, while classical variables are still a problem.

Here is a small example of transformations:

$$add_sub = \lambda\{1 \Rightarrow x, 2 \Rightarrow y\}.\{1 \Rightarrow x - y, 2 \Rightarrow x + y\} : \{1 \Rightarrow int, 2 \Rightarrow int\} \rightarrow \{1 \Rightarrow int, 2 \Rightarrow int\}$$

$$add_sub\ \{1 \Rightarrow 5, 2 \Rightarrow 3, ok \Rightarrow true\} = \{1 \Rightarrow 2, 2 \Rightarrow 8, ok \Rightarrow true\} : \{1 \Rightarrow int, 2 \Rightarrow int, ok \Rightarrow bool\}$$

As you see here our proposal includes typing. Moreover, some constructors are introduced for supporting stateful objects and prototypes.

1.4 Symmetrical transformations

The last system we present in this thesis extends transformation calculus towards mathematical relations.

The basic goal of symmetric transformations is to provide an algebra for computing with n-ary relation. While some relational calculi including composition have been proposed for binary relations, in a categorical framework for instance, the n-ary case is more complex.

The solution chosen here is again to curry binary relations. That is, we see relations as being two-sided, with part of their contents on each side.

$$(A_1, \ldots, A_k, A_{k+1}, \ldots, A_n)$$

becomes

$$\{\overline{l_1 \Rightarrow A_1, \ldots, l_k \Rightarrow A_k}\}\{l_{k+1} \Rightarrow A_{k+1}, \ldots, l_n \Rightarrow A_n\}$$

where $\overline{\{\ldots\}}$ "looks" to the left and $\{\ldots\}$ "looks" to the right.

This is in fact reminiscent of

$$\lambda\{l_1 \Rightarrow A_1, \ldots, l_k \Rightarrow A_k\}.\{l_{k+1} \Rightarrow A_{k+1}, \ldots, l_n \Rightarrow A_n\}$$

in the transformation calculus. Then we can, for instance, use the curried composition of transformation calculus to compute joints of relations.

This calculus is extended in various ways to make it a tool for the specification of the geometry of communicating systems, or the syntactical study of inversion.

1.5 Organization of this thesis

We already explained above the distribution in three parts — selective λ-calculus, transformation calculus and symmetric transformations. They are in natural progression.

Two appendices complete those: the first one gives some peripheric results, obtained during the study of the above calculi; the second one presents a small strongly typed functional programming language, FIML, based on transformation calculus.

Sequential reading is possible. One may however want to skip part of the contents. Here is the dependency structure of this thesis.

- The first two sections of Chapter 2, defining a monadic version of selective λ-calculus, are useful to the understanding of the whole thesis. The rest is only the proof for confluence.
- Chapter 3, where *streams* and the stream (polyadic) version of selective λ-calculus are defined, is necessary for all the following chapters.

- Chapters 4 to 8 are to a great extent mutually independant.
- Appendices A and B are self-contained.

So the reader is left free to chose his order of exploration, but will probably have to go to the first section of chapter 3 sooner or later.

Last, there is no “Background” part in this thesis. The initial definition of Currying does not require long explanations. For more general bases we refer the reader to any good book on lambda calculus [Bar81, HS86].

Chapter 2

Selective λ-calculus : Syntax

We define here the (monadic) selective λ-calculus as the combination of a calculus where labels are names with one where they are indexes. More than half of this part is devoted to the confluence proof.

2.1 Introducing selective λ-calculi

2.1.1 Generic syntax

Selective λ-terms are formed by variables taken from a set $\mathcal{V}$, and two labeled constructions: abstraction and application. The labeling is done with labels taken from a set of *position labels* $\mathcal{L}$.

We will denote variables by x, y, labels in $\mathcal{L}$ by p, q, and λ-expressions by capitals.

We can define the syntax of λ-terms as:

$$\begin{array}{rcll} M & ::= & x & \text{(variables)}, \\ & | & \lambda_p x.M & \text{(abstractions)}, \\ & | & M \,\widehat{}_p\, M & \text{(applications)}. \end{array}$$

We will say of a term $\lambda_p x.M$ that it *"abstracts x on p in M,"*, and of the term $M \,\widehat{}_p\, N$, that it *"applies M to N on p."*

It will often be convenient to break the atomicity of an abstraction or an application. In the abstraction $\lambda_p x.M$, the part $\lambda_p x$ will be called its *abstractor*, and M its *body*. In the application $M \,\widehat{}_p\, N$, the part $\,\widehat{}_p\, N$ will be called the *applicator*. By *entity*, we will mean either an abstractor or an applicator.

2.1.2 Relative and absolute positions

Before we look at different selective λ-calculi, let us give some intuition to justify this syntax, thinking of two possible sets of labels, symbolic and numeric ones.

Symbolic labels, or "keywords", are simple names. A useful way of thinking of these symbols is to see them as *channel names* used for process communication [Mil92]. Here, a process is a λ-term, where *sending* is performed by applicators and *receiving* by abstractors. If an application is performed ("sends arguments") through two different channels p and q, then clearly there cannot be any ambiguity as far as which abstractor will "receive" them. Hence, these reductions ("communications") may be done in any order, with the same end result. However, if that situation arises with $p = q$, then clearly the order in which they are performed will matter. In this case, the rules will insure that reduction will respect the order specified syntactically. In other words, several arguments sent through the same channel are "buffered" in sequence.

If numeric labels are always kept explicit, then the above view applies to them as well. Indeed, recall from the introduction that the free syntax of function application to several arguments at a time uses their positions as Cartesian projections; *e.g.* $f(a_1,\ldots,a_n)$ may be seen as the more explicit $f(1\Rightarrow a_1,\ldots,n\Rightarrow a_n)$. However, numeric labels do not quite behave like symbolic labels in that a number is always *implicitly* seen as the *first* position *relatively* to the form on its left. More precisely, currying works by seeing each argument as the first one relatively to the form on its left. This has the benefit of simplifying the rule of functional reduction to be a *local* rule never needing to consider more than a single argument at a time. So, clearly, we do want to allow using relative argument positions.

Nevertheless, it is more natural to use absolute positions "packaged" as labeled Cartesian tuples. For instance, it is easier to write $(\lambda\{1\Rightarrow x,2\Rightarrow y,4\Rightarrow z\}.M)\,\{1\Rightarrow a,4\Rightarrow b\}$ rather than $(\lambda_1 x.\lambda_1 y.\lambda_2 z.M)\,\widehat{1}\,a\,\widehat{3}\,b$. However, the latter fully curried form is needed to express reduction with local rules. Fortunately, translation from the notation with absolute labels to a fully curried one with relative labels is in fact systematic: one need simply subtract from each numeric label the number of numeric-labeled components, smaller than it, and appearing to its left in the labeled Cartesian product. Namely,

$$M\,\{n_1\Rightarrow N_1,\ldots,n_k\Rightarrow N_k\} = M\,\widehat{n'_1}\,N_1\ldots\widehat{n'_k}\,N_k$$
$$\text{where } n'_k = n_k - |\{i \mid i<k, n_i<n_k\}|.$$

Conversely, one may go back from relative syntax to the absolute one by inserting iteratively entities in an abstraction or application tuple. That is,

$$(M\,\widehat{m}\,N)\,\{n_1\Rightarrow N_1,\ldots,n_k\Rightarrow N_k\} = M\,\{m\Rightarrow N, n'_1\Rightarrow N_1,\ldots,n'_k\Rightarrow N_k\}$$
$$\text{where } n'_i = \begin{cases} n_i & \text{if } n_i < m \\ n_i+1 & \text{if } n_i \geq m \end{cases}$$

These two rules apply directly for abstractions too, and one may verify that they just do opposite work.

For the absolute and relative notations to be effectively coherent, we will expect $M\,\{n_1\Rightarrow N_1,\ldots,n_k\Rightarrow N_k\}$ and $M\,\{n_{\sigma(1)}\Rightarrow N_{\sigma(1)},\ldots,n_{\sigma(k)}\Rightarrow N_{\sigma(k)}\}$ to be convertible terms for any permutation σ of $[\![1,k]\!]$, that is, the order of the pairs in a record should be semantically irrelevant.

With this, we are justified to limit our syntax to that of relative-labeling lending itself to simpler local reduction rules, while still keeping the freedom of a flexible surface syntax with Cartesian tuples using absolute position labeling.

Now, a reasonable question that one may have is whether we could not also treat symbolic labels as we do numeric labels. That is, we could envisage using a function associating each symbol to its predecessor in the linear order of symbols, thus doing away with names altogether. This, however, would be possible only if the order on symbols were not dense. Since, in practice, symbols are the free monoid, generated by a subset of the ASCII alphabet, and is densely ordered by lexicographic ordering, this is ruled out. Hence, symbolic labels always designate *absolute* positions of arguments. In other words, packaging symbolic-labeled arguments in labeled Cartesian tuples is always safe since they are not concerned with relative positioning. In fact, the ordering on symbols is only necessary as a trick to avert non-termination so that rules may perform well-founded label commutation.

Reciprocally one could think of getting rid of numeric labels. However, simply forgetting about numeric labels, just because they are a little cumbersome, would reduce the generality of the calculus. With only symbolic labels we can directly send values to abstractions *as long as they have different labels*. An abstraction can still be hidden

β – reduction

(β) $(\lambda_a x.M)\,\widehat{}_a\, N \to [N/x]M$

Reordering

(1) $\lambda_a x.\lambda_b y.M \to \lambda_b y.\lambda_a x.M \qquad a > b$

(2) $M\,\widehat{}_a\, N_1\,\widehat{}_b\, N_2 \to M\,\widehat{}_b\, N_2\,\widehat{}_a\, N_1 \qquad a > b$

(3) $(\lambda_a x.M)\,\widehat{}_b\, N \to \lambda_a x.(M\,\widehat{}_b\, N) \qquad a \neq b, x \notin FV(N)$

Figure 2.1: Reduction rules for symbolic selective λ-calculus

by another abstraction with same label. However, with symbolic labels, we have this property *in all cases*. This is certainly useful if you want, for instance, to construct a model of this calculus: intuitively all curried functions become flat, while they would still be partly hierarchized in an only keyword calculus.

2.1.3 A lambda-calculus with multiple channels

This is the first possibility, using keywords as label. We define an extension of the lambda calculus, the *symbolic selective λ-calculus*, with symbolic labels.

Following the above syntax, we take our labels from a totally ordered set of symbols $\mathcal{S}$. We will denote these labels by a, b.

To keep compatibility with the classical λ-calculus, we have a default label, ϵ, such that an unlabeled abstraction or application is interpreted as being labeled by ϵ.

The reduction rules for this calculus are given in Figure 2.1. β-reduction only happens on abstractor-applicator pairs with the same label. Otherwise they commute by rule (3). Rules (1) and (2) normalize the order of abstractors and applicators. The condition $x \notin FV(N)$ in rule (3) can always be satisfied through α-conversion.

Definition 2.1 (symbolic) *We call* symbolic selective λ-calculus *the free combination of rules in Figure 2.1.*

This calculus is meaningful, in that it is confluent.

Corrolary 2.1 *The symbolic selective λ-calculus is confluent.*

PROOF Consequence of the proof for selective λ-calculus. □

Example 2.1 *We suppose that $a < b < c < d$,*
(For keywords the notations $\lambda\{a \Rightarrow x, \ldots\}$ and $M\ \{a \Rightarrow N, \ldots\}$ are only shorthands.)

$$
\begin{array}{ll}
 & (\lambda\{a \Rightarrow x, b \Rightarrow y, c \Rightarrow z\}.M)\,\{c \Rightarrow N_1, d \Rightarrow N_2, a \Rightarrow N_3\} \\
= & (\lambda_a x.\lambda_b y.\lambda_c z.M)\,\widehat{}_c\, N_1\,\widehat{}_d\, N_2\,\widehat{}_a\, N_3 \\
\to_3 & (\lambda_a x.((\lambda_b y.\lambda_c y.M)\,\widehat{}_c\, N_1))\,\widehat{}_d\, N_2\,\widehat{}_a\, N_3 \\
\to_2 & (\lambda_a x.((\lambda_b y.\lambda_c y.M)\,\widehat{}_c\, N_1))\,\widehat{}_a\, N_3\,\widehat{}_d\, N_2 \\
\to_\beta & (\lambda_b y.\lambda_c z.[N_3/x]M)\,\widehat{}_c\, N_1\,\widehat{}_d\, N_2 \\
\to_3 & (\lambda_b y.((\lambda_c z.[N_3/x]M)\,\widehat{}_c\, N_1))\,\widehat{}_d\, N_2 \\
\to_\beta & (\lambda_b y.([N_3/x][N_1/z]M))\,\widehat{}_d\, N_2) \\
\to_3 & \lambda_b y.([N_3/x][N_1/z]M\ \widehat{}_d\, N_2)
\end{array}
$$

$\beta-$ reduction

(β) $(\lambda_n x.M)\,\widehat{}_n\, N \rightarrow [N/x]M$

Reordering

(1) $\lambda_m x.\lambda_n y.M \rightarrow \lambda_n y.\lambda_{m-1} x.M \qquad m > n$

(2) $M\,\widehat{}_m\, N_1\,\widehat{}_n\, N_2 \rightarrow M\,\widehat{}_n\, N_2\,\widehat{}_{m-1}\, N_1 \qquad m > n$

(3) $(\lambda_m x.M)\,\widehat{}_n\, N \rightarrow \lambda_{m-1} x.(M\,\widehat{}_n\, N) \qquad m > n, x \notin FV(N)$

(4) $(\lambda_m x.M)\,\widehat{}_n\, N \rightarrow \lambda_m x.(M\,\widehat{}_{n-1}\, N) \qquad m < n, x \notin FV(N)$

Figure 2.2: Reduction rules for numerical selective λ-calculus

2.1.4 A lambda-calculus with moving indexes

In this calculus we can selectively apply a function on any of its arguments, according to its *apparent position*.

For an unlabeled expression, the apparent position of an abstractor is intuitively defined as the number of times we have to apply this expression in order to have the abstractor applied to the desired argument. For instance, in $\lambda x.\lambda y.\lambda z.M$, the apparent position of the abstractor of z is 3, but in $\lambda x.(\lambda y.\lambda z.M)N$ it is 2. As a consequence, apparent positions do not change when we reduce an expression. When we add labels, we want to keep this property.

Definition 2.2 (numeric) Numerical selective λ-calculus *takes its labels from* $\mathcal{N} = \mathbb{N} - \{0\}$. *Reduction rules on terms modulo* α*-conversion are given in Figure 2.2.*

Definition 2.3 (apparent position) *The* apparent position *of an abstractor in a term* M *is* n *such that* $M\,\widehat{}_n\, N$ associates *this abstractor and* N *(makes them to be* β*-reduced together eventually).*

Well-definedness of apparent positions is guaranteed by confluence. Of course, if an abstractor is already linked with an applicator in the term, or appears in the right-hand of an application, it has no apparent position.

Corrolary 2.2 *The numerical selective* λ*-calculus is confluent.*

PROOF Consequence of the proof for selective λ-calculus. □

We can now relate apparent positions to the absolute positions of our relative *vs.* absolute dichotomy. The idea is that when we apply M to the tuple $\{n_1 \Rightarrow N_1, \ldots, n_k \Rightarrow N_k\}$, the n_i's, which are absolute positions in the above definition, are the apparent positions in M of the abstractors they aim at. Similarly, in $\lambda\{n_1 \Rightarrow x_1, \ldots, n_k \Rightarrow x_k\}.M$, x_i has apparent position n_i. As a result, we have

$$(\lambda\{n_1 \Rightarrow x_1, \ldots, n_k \Rightarrow x_k\}.M)\,\{n_1 \Rightarrow N_1, \ldots, n_k \Rightarrow N_k\} \xrightarrow{*} [N_i/x_i]_{i=1}^{k} M$$

and the order of bindings in records is free, as one would expect.

Example 2.2 *Numerical indexes*

$$
\begin{array}{ll}
 & (\lambda\{2\Rightarrow x, 1\Rightarrow y, 4\Rightarrow z\}.M)\,\{4\Rightarrow N_1, 6\Rightarrow N_2, 2\Rightarrow N_3\} \\
= & (\lambda_2 x.\lambda_1 y.\lambda_2 z.M)\,\widehat{4}\,N_1\,\widehat{5}\,N_2\,\widehat{2}\,N_3 \\
\rightarrow_4 & (\lambda_1 y.\lambda_1 x.\lambda_2 z.M)\,\widehat{4}\,N_1\,\widehat{5}\,N_2\,\widehat{2}\,N_3 \\
\rightarrow_7 & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\,\widehat{3}\,N_1))\,\widehat{5}\,N_2\,\widehat{2}\,n_3 \\
\rightarrow_5 & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\,\widehat{3}\,N_1))\,\widehat{2}\,N_3\,\widehat{4}\,N_2 \\
\rightarrow_7 & (\lambda_1 y.\lambda_1 x.((\lambda_2 z.M)\,\widehat{2}\,N_1))\,\widehat{2}\,N_3\,\widehat{4}\,N_2 \\
\rightarrow_\beta & (\lambda_1 y.\lambda_1 x.[N_1/z]M)\,\widehat{2}\,N_3\,\widehat{4}\,N_2 \\
\rightarrow_7 & (\lambda_1 y.((\lambda_1 x.[N_1/z]M)\,\widehat{1}\,N_3))\,\widehat{4}\,N_2 \\
\rightarrow_\beta & (\lambda_1 y.[N_3/x][N_1/z]M)\,\widehat{4}\,N_2 \\
\rightarrow_7 & \lambda_1 y.([N_3/x][N_1/z]M\,\widehat{3}\,N_2)
\end{array}
$$

2.1.5 A first way to combine these two systems

Intuitively it would be interesting to get in one calculus both the power of symbolic and numeric selective λ-calculi.

For this we take $\mathcal{L}$ to be the disjoint union of the two sets $\mathcal{N}$ of numeric, and $\mathcal{S}$ of symbolic labels. Namely, $\mathcal{N}$ is ordered with the natural number ordering, that we shall write $<_{\mathcal{N}}$; $\mathcal{S}$ is ordered with a linear order that we write $<_{\mathcal{S}}$; and, $\mathcal{L}$ is ordered by the order $<_{\mathcal{L}}$ such that $<_{\mathcal{L}} = <_{\mathcal{N}}$ on $\mathcal{N}$, $<_{\mathcal{L}} = <_{\mathcal{S}}$ on $\mathcal{S}$, and $\forall (n,p) \in \mathcal{N} \times \mathcal{S},\ n <_{\mathcal{L}} p$. In other words, all numeric labels are less than all symbolic labels.

Our set of rule is the union of symbolic (Fig. 2.1) and numeric (Fig. 2.2) rules applied to their respective sets of labels, and the three following rules, which just generalize symbolic ones to handle conflicts with numeric labels, in conformity with our new order.

$$
\begin{array}{ll}
(1') & \lambda_a x.\lambda_n y.M \rightarrow \lambda_n y.\lambda_a x.M \\
(2') & M\,\widehat{a}\,N_1\,\widehat{n}\,N_2 \rightarrow M\,\widehat{n}\,N_2\,\widehat{a}\,N_1 \\
(3') & (\lambda_a x.M)\,\widehat{n}\,N \rightarrow \lambda_a x.(M\,\widehat{n}\,N) \quad x \notin FV(N)
\end{array}
$$

We call this system *flat selective λ-calculus*. Again it is confluent.

Corrolary 2.3 *The flat selective λ-calculus is confluent.*

PROOF Consequence of the proof for selective λ-calculus. □

We considered for a long time the flat calculus as the best balanced of these calculi, since it includes both symbolic and numeric calculi in a coherent way. Indeed most of the terms one would write can be expressed in it. However it appears that a stronger one includes it conservatively, and we will rather chose that one as "fundamental" calculus.

2.2 The selective λ-calculus

2.2.1 Definition

2.2.1.1 Syntax

Selective λ-calculus combines orthogonally symbolic and numerical selective λ-calculi. Its set of labels is $\mathcal{L} = \mathcal{S} \times \mathcal{N}$. [1] The order induced on labels is the lexicographical one: $a <_{\mathcal{S}} b \Rightarrow am <_{\mathcal{L}} bn$ and $m <_{\mathcal{N}} n \Rightarrow am <_{\mathcal{L}} an$.

$$M ::= x \mid \lambda_{an} x.M \mid M\,\widehat{an}\,M'$$

[1] In [AKG93b] this was defined as a product system, and selective λ-calculus as the sum system $\mathcal{L} = \mathcal{S} \cup \mathcal{N}$. Properties of the two systems being similar, we work here on the most general one.

The set of selective λ-terms (considered modulo α-conversion) is Λ. We use a, b for symbols (or *channels*), m, n for numbers (or *indexes*), and p, q for labels formed of a couple (channel,index).

2.2.1.2 Substitutions

Substitution of variables by λ-expressions needs the same precautions as in λ-calculus and obeys exactly the same rules. As usual, we use the equal sign $(=)$ to mean syntactic equality modulo α-conversion, defining α-conversion as for classical λ-calculus.

Let FV(M) be the set of free variables in M, defined as usual in lambda calculus. The expression $[N/x]M$ denotes the term obtained by replacing all the free occurrences of a variable x by N in (an appropriate α-renaming of) M. That is,

$$
\begin{array}{lcll}
[N/x]x & = & N & \\
[N/x]y & = & y & \text{if } y \in \mathcal{V},\ y \neq x \\
[N/x](M_1 \mathbin{\widehat{p}} M_2) & = & ([N/x]M_1) \mathbin{\widehat{p}} ([N/x]M_2) & \\
[N/x](\lambda_p x.M) & = & \lambda_p x.M & \\
[N/x](\lambda_p y.M) & = & \lambda_p y.[N/x]M & \\
 & & \multicolumn{2}{l}{\text{if } y \neq x \text{ and } y \notin \text{FV}(N)} \\
[N/x](\lambda_p y.M) & = & \lambda_p z.[N/x][z/y]M & \\
 & & \multicolumn{2}{l}{\text{if } y \neq x \text{ and } y \in \text{FV}(N),} \\
 & & \multicolumn{2}{l}{\text{and } z \notin \text{FV}(N) \cup \text{FV}(M).}
\end{array}
$$

2.2.1.3 Reductions

The reduction system is the combination in Figure 2.3. We call *weak reordering* the system excepting β-reduction.

This may look complex, but one can see reordering rules as structural equalities, and then we have β-reduction as unique reduction rule. One might wonder about why then we do not adopt this view, and separate completely reordering from β-reduction. The answer is that to define from the beginning reordering as a structural equivalence, we would need a good understanding of what is a term modulo reordering. As a matter of fact, we can define it, using a monoid of records. But it would be as complex, and term structure harder to grasp. So, it is probably more convincing to first verify the confluence on a simple structure, and then trivially extend it to the equational one.

Since the combination is orthogonal (symbolic and numeric labels work on two independent levels), confluence is inherited from the two previous systems.

Theorem 2.1 *The selective λ-calculus is confluent.*

PROOF in Section 2.3. □

To let this system include the symbolic and numerical sub-calculi, we will identify a symbol a with the label $(a, 1)$ and an index n with the label (ϵ, n). Remark that the default label to encode classical λ-calculus terms is now $(\epsilon, 1)$.

2.2.2 Entity syntax

To emphasize the similarity between abstraction and application we define a new notation for the first.

$$\lambda_p x.M = M \mathbin{\overset{\vee}{p}} x$$

β – reduction

(β) $(\lambda_p x.M)\,\widehat{}_p\, N \to [N/x]M$

Symbolic reordering

(1) $\lambda_{am} x.\lambda_{bn} y.M \to \lambda_{bn} y.\lambda_{am} x.M$ $\quad a > b$

(2) $M\ \widehat{}_{am}\ N_1\ \widehat{}_{bn}\ N_2 \to M\ \widehat{}_{bn}\ N_2\ \widehat{}_{am}\ N_1$ $\quad a > b$

(3) $(\lambda_{am} x.M)\,\widehat{}_{bn}\, N \to \lambda_{am} x.(M\ \widehat{}_{bn}\ N)$ $\quad a \neq b, x \notin FV(N)$

Numeric reordering

(4) $\lambda_{am} x.\lambda_{an} y.M \to \lambda_{an} y.\lambda_{am-1} x.M$ $\quad m > n$

(5) $M\ \widehat{}_{am}\ N_1\ \widehat{}_{an}\ N_2 \to M\ \widehat{}_{an}\ N_2\ \widehat{}_{am-1}\ N_1$ $\quad m > n$

(6) $(\lambda_{am} x.M)\,\widehat{}_{an}\, N \to \lambda_{am-1} x.(M\ \widehat{}_{an}\ N)$ $\quad m > n, x \notin FV(N)$

(7) $(\lambda_{am} x.M)\,\widehat{}_{an}\, N \to \lambda_{am} x.(M\ \widehat{}_{an-1}\ N)$ $\quad m < n, x \notin FV(N)$

Figure 2.3: Reduction rules for selective λ-calculus

for any $p \in \mathcal{L}$, $x \in \mathcal{V}$, $M \in \Lambda$.

As a result, β-reduction becomes:

$$M\ \overset{\vee}{}_p\ x\ \overset{\wedge}{}_p\ N \to_\beta M[x \backslash N],$$

where it is natural to write substitutions on the right side of terms.

We can redefine more clearly the notions we introduced for the generic syntax, and introduce some new ones.

Definition 2.4 (entity) *An* entity *(in Γ), either an* applicator *or* an abstractor*, is a pair of an* operator *($\overset{\wedge}{}_p$ or $\overset{\vee}{}_p$ for some $p \in \mathcal{L}$) and a term for applications, a variable for abstractions.*

We chose here the term abstractor rather than the usual *binder* to emphasize on the presence of a label. Then we can call binder the variable of an abstractor, that is the x in $\overset{\vee}{}_p\ x$.

Definition 2.5 (head and spine) *We distinguish in a term its* head *and its* spine*. Any selective λ-term can be written*

$$x \odot P = (\ldots(x\ e_1)\ldots)e_n$$

where the head x is a variable and the spine P an entity sequence $(e_1, \ldots, e_n) \in \Gamma^*$.

$\iota_P(e_k) = n - k$ is the position *of e_n in P (counting from the right).*

- For any entity sequence P in Γ^* we define $BV(P)$, the set of variables *bounded by* P, that is the set of variables abstracted by the roots of the entities forming P. For any M in Λ, any occurrence of a variable of $BV(P)$ in M is bound in $M \odot P$.
- We note $P \cdot Q$ the concatenation of two entity sequences, and have $M \odot (P \cdot Q) = (M \odot P) \odot Q$.

Example 2.3 *In the term $x\ \overset{\wedge}{}_p\ z\ \overset{\vee}{}_q\ x\ \overset{\vee}{}_q\ y$, x is the* head *and $P = \overset{\wedge}{}_p\ z\ \overset{\vee}{}_q\ x\ \overset{\vee}{}_q\ y$ the spine. $BV(P)$ is $\{x, y\}$ and $FV(P)$ is $\{z\}$.*

This notation will simplify many proofs. We will use indifferently the two notations in the following.

2.3 Proof of confluence

2.3.1 Pseudo-reduction

2.3.1.1 Rules

Pseudo-reduction rules are intended to make reordering systems confluent in the absence of β-reduction. They promote the formation of new reordering redexes by commutation over β-redexes. The idea is that, without β-reduction, β-redexes just sit there, presenting "obstacles" to the formation of reordering redexes. Hence, we need pseudo-reduction rules to simulate the promotion of reordering redexes that would appear if the β-reduction had been performed. We simulate that effect by having a labeled entity "jump" into, or out of, the body of the abstraction part of a β-redex through its "λ-membrane" and seek reordering on the other side of that membrane. There are two cases: (a) one corresponding to having an applicator jump "into" the body of the abstraction part of a β-redex, and (b) the other corresponding to having an abstractor jump "out of" it. Namely,

$$(a) \quad M \overset{\vee}{}_{p}\, x \overset{\wedge}{}_{p}\, N_1 \overset{\wedge}{}_{q}\, N_2 \to M \overset{\wedge}{}_{q}\, N_2 \overset{\vee}{}_{p}\, x \overset{\wedge}{}_{p}\, N_1 \qquad x \notin FV(N_2)$$
$$(b) \quad M \overset{\vee}{}_{q}\, y \overset{\vee}{}_{p}\, x \overset{\wedge}{}_{p}\, N \to M \overset{\vee}{}_{p}\, x \overset{\wedge}{}_{p}\, N \overset{\vee}{}_{q}\, y \qquad y \notin FV(N)$$

Example 2.4 *If we use only reordering rules,* $(\lambda_1 x.\lambda_2 y.x \overset{\wedge}{}_{1}\, y) \overset{\wedge}{}_{2}\, a \overset{\wedge}{}_{1}\, b$ *can be reduced by Rules* (7) *and* (6) *yielding* $A = (\lambda_1 x.\lambda_1 y.x \overset{\wedge}{}_{1}\, y \overset{\wedge}{}_{1}\, a) \overset{\wedge}{}_{1}\, b$*. It can also be reduced by Rule* (5) *to* $B = (\lambda_1 x.\lambda_2 y.x \overset{\wedge}{}_{1}\, y) \overset{\wedge}{}_{1}\, b \overset{\wedge}{}_{1}\, a$*. Both terms* A *and* B *are normal forms with respect to reordering. We need pseudo-reduction to recover confluence. Namely, applying Rule* (a) *to* B *promotes the appearance of a redex for Rule* (6)*, which yields* A*.*

2.3.1.2 Pseudo-reduced form equivalence (PRF)

Since some critical pairs appear between reordering rules, we introduce an equivalence relation that unifies their results. Confluence of the reordering part of the system can only be considered under this equivalence.

Definition 2.6 (PRF-equivalence) *If* $x \notin FV(N_1)$ *and* $y \notin FV(N_2)$ *then*

$$M \overset{\vee}{}_{q}\, y \overset{\wedge}{}_{q}\, N_1 \overset{\vee}{}_{p}\, x \overset{\wedge}{}_{p}\, N_2 \leftrightarrow M \overset{\vee}{}_{p}\, x \overset{\wedge}{}_{p}\, N_2 \overset{\vee}{}_{q}\, y \overset{\wedge}{}_{q}\, N_1$$

For any M, x, N*,*

$$M \overset{\vee}{}_{am}\, x \overset{\wedge}{}_{am}\, N \leftrightarrow M \overset{\vee}{}_{an}\, x \overset{\wedge}{}_{an}\, N$$

Example 2.5 *Consider the term* $A = M \overset{\vee}{}_{1}\, x \overset{\vee}{}_{2}\, y \overset{\wedge}{}_{2}\, N \overset{\wedge}{}_{1}\, N'$*. If we exclude* β*-reduction, we can still apply either rule (4) or (5), yielding to* $B = M \overset{\vee}{}_{1}\, y \overset{\vee}{}_{1}\, x \overset{\wedge}{}_{2}\, N \overset{\wedge}{}_{1}\, N'$ *or* $C = M \overset{\vee}{}_{1}\, x \overset{\vee}{}_{2}\, y \overset{\wedge}{}_{1}\, N' \overset{\wedge}{}_{1}\, N$*, which by (7) and (6) give* $B' = M \overset{\vee}{}_{1}\, y \overset{\wedge}{}_{1}\, N \overset{\vee}{}_{1}\, x \overset{\wedge}{}_{1}\, N'$ *and* $C' = M \overset{\vee}{}_{1}\, x \overset{\wedge}{}_{1}\, N' \overset{\vee}{}_{1}\, y \overset{\wedge}{}_{1}\, N$*. No reordering rule can recover confluence between* B' *and* C'*, but they are PRF equivalent.*

The necessity for the second equation is more subtle, but is linked to the use of pseudo-reductions.

Combining these two we can see that PRF equivalence makes both order and symbolic part of labels irrelevant for *locally associated pairs* (*cf.* Def. 2.8), as long as variable dependencies are satisfied.

2.3.2 Combined systems and restricted reductions

We will first distinguish reordering rules, and prove properties of those alone. To do this we have to add some rules to preserve confluence: pseudo-reductions (a) and (b). So the system we are really interested in is weak reordering combined with pseudo-reduction. We will call it the *reordering system*.

An *order normal form* is a normal form for the reordering system.

A *stable reordering* is a reordering where pseudo-reduction is prefered to weak reordering for critical pairs.

Intermediate statements will be done on the system called β*-reordering*. It is β-reduction on order normal forms, where the result of each step is normalized by the reordering system.

A last system, the *label-parallel system*, which makes the link between selective λ-calculus and β-reordering, is the combination of all rules, and the *stable label-parallel system* includes the same restriction as stable reordering.

For each of these reduction systems, we shall use the symbol $\rightarrow$ to indicate a single reduction step using any of system's rules, and $\rightarrow_r$ if the rule uses Rule (r). When unconcerned by termination, we shall accept the step $(M \rightarrow M)$ in this relation. As usual, $\stackrel{*}{\rightarrow}$ is the reflexive and transitive closure, also possibly subscripted. Given a reduction strategy ϱ, we will use the symbol $\triangleright_\varrho$ to denote the subrelation of $\stackrel{*}{\rightarrow}$ using only ϱ-reduction steps. For example, $\triangleright_{stb}$ for stable reorderings, $\triangleright_{mcd}$ for minimal complete developments, *etc...*

For many of these systems confluence will be considered **modulo** pseudo-reduced form equivalence. Here is the definition, as given in [Hue80].

Definition 2.7 (confluent modulo) *We say that the relation $\rightarrow$ is* confluent modulo $\sim$ *iff*

$$(\forall xyx'y')\ x \sim y \wedge x \stackrel{*}{\rightarrow} x' \wedge y \stackrel{*}{\rightarrow} y' \Rightarrow (\exists \overline{xy})\ x' \stackrel{*}{\rightarrow} \overline{x} \wedge y' \stackrel{*}{\rightarrow} \overline{y} \wedge \overline{x} \sim \overline{y}.$$

2.3.3 Confluence of the reordering system

Before going on it may be good to have an idea of what an order normal form looks like, but first we need a basic definition.

Definition 2.8 (associated) *An abstractor and an applicator are said to be* locally associated *when they match β-reduction, namely $\stackrel{\vee}{p}\ x$ and $\stackrel{\wedge}{p}\ N$ in $M \stackrel{\vee}{p}\ x \stackrel{\wedge}{p}\ N$. They are* locally free *if they do not. They are* associated *if a reordering can bring them to this state,* free *otherwise.*

Proposition 2.1 *The general structure of an order normal form is:*

$$x \stackrel{\wedge}{p_1} N_1 \ldots \stackrel{\wedge}{p_j} N_j \stackrel{\vee}{q_k} y_k \stackrel{\wedge}{q_k} K_k \ldots \stackrel{\vee}{q_1} y_1 \stackrel{\wedge}{q_1} K_1 \stackrel{\vee}{r_l} x_l \ldots \stackrel{\vee}{r_1} x_1$$

where $p_i \leq p_{i+1}$, $r_i \leq r_{i+1}$, N_i's and K_i's are in order normal form.

PROOF Thanks to rules (3),(6),(7) and pseudo-reductions, in each spine, all locally free applicators (resp. abstractors) have to be on the left of all absractors (resp. on the right of all applicators).

By rules (1) and (4) free abstractors must have decreasing labels; and by rules (2) and (5) free applicators must have growing ones.

Locally associated abstractors and applicators stay by pair with the same label. □

Then next lemma is essential, since it allows us to consider reordering as a reduction on independent spines rather than terms.

Lemma 2.1 (Stability of entities) *The reordering rules (1)–(7),$(a), (b)$ do not produce any labeled entities nor do they destroy any. Moreover, a labeled entity stays on the same spine after any reordering rule application, and only the numeric part of its label may change.*

PROOF A quick look at the rules shows that none moves an entity from a spine in the set of spines of a term to another one. Moreover, it is even possible to track entities through the transformations, considering that those entities corresponding to the same label occurrence on the two sides of the rule are in fact identical up to variable renaming details (by "same label occurrence" we include m and $m-1$, n and $n-1$). □

Now we can give a new, and more general definition of the notion of apparent position. We base it on *shifting*.

Definition 2.9 (shifting) *To each spine P we associate a* shifting function $\phi(P)$ *defined as follows. It corresponds to pushing right $\overset{\vee}{}_{am}\ x$ through P, using reordering rules bidirectionally.*

$$\begin{array}{lcll}
\phi()(am) & = & am & \\
\phi(\overset{\vee}{}_{bk}\ y \cdot P)(am) & = & \phi(P)(am) & a \neq b \\
\phi(\overset{\wedge}{}_{bk}\ N \cdot P)(am) & = & \phi(P)(am) & a \neq b \\
\phi(\overset{\vee}{}_{an}\ y \cdot P)(am) & = & \phi(P)(am) & m < n \\
\phi(\overset{\vee}{}_{an}\ y \cdot P)(am) & = & \phi(P)(am+1) & m \geq n \\
\phi(\overset{\wedge}{}_{an}\ N \cdot P)(am) & = & \phi(P)(am) & m < n \\
\phi(\overset{\wedge}{}_{an}\ N \cdot P)(am) & = & \phi(P)(am-1) & m > n
\end{array}$$

$\phi(P)(am)$ is undefined iff our abstractor bumps into an applicator with same label in P. Similarly $\phi^{-1}(P)(am)$ is uniquely defined and corresponds to pushing left $\overset{\wedge}{}_{am}\ N$ through P; it is undefined iff our applicator bumps into an abstractor with same label.

Proposition 2.2 (shifting) *a. ϕ is compositional: $\phi(P \cdot Q) = \phi(Q) \circ \phi(P)$.*

b. $\phi^{-1}(P) = \phi(\overline{P})$ where $\overline{P}$ is the dual of P.

$$\begin{array}{lcl}
\overline{P \cdot Q} & = & \overline{Q} \cdot \overline{P} \\
\overline{\overset{\vee}{}_{an}\ x} & = & \overset{\wedge}{}_{an}\ x \\
\overline{\overset{\wedge}{}_{an}\ N} & = & \overset{\vee}{}_{an}\ x_N
\end{array}$$

PROOF

a. by the recursivity of the definition.

b. by verifying cases 5 and 7 of the definition exchange correctly, and by compositionality of ϕ.

□

Definition 2.10 (apparent position) *The apparent position of $\overset{\vee}{}_{an}\ x$ in $P = P_1 \cdot \overset{\vee}{}_{an}\ x \cdot P_2$ is $\pi_P(\overset{\vee}{}_{an}\ x) = \phi(P_2)(an)$. That of $\overset{\wedge}{}_{an}\ N$ is $P = P_1 \cdot \overset{\wedge}{}_{an}\ N \cdot P_2$ is $\pi_P(\overset{\wedge}{}_{an}\ N) = \phi^{-1}(P_1)(an)$.*

We will have to verify that this new definition matches the precedent. This is in fact equivalent to having the following *static association* match previous association.

Definition 2.11 (static association) *An abstractor $\overset{\vee}{}_{am}\ x$ and an applicator $\overset{\wedge}{}_{an}\ N$ are said to be* statically associated *when they belong to the same spine $P \cdot \overset{\vee}{}_{am}\ x \cdot Q \cdot \overset{\wedge}{}_{an}\ N \cdot R$ and $\phi(Q)(an) = am$.*

What intuitionally this definition does is using ϕ to simulate a reordering moving ${}^{\vee}_{am}\ x$ outwards. If this simulation succeeds in meeting ${}^{\wedge}_{an}\ N$ with an abstractor ${}^{\vee}_{an}\ x$, then they are statically associated.

The following lemma is the most important of this subsection. It proves all at once that apparent positions are invariant, static association too, and that it is association.

Lemma 2.2 *Apparent positions are conserved by reordering.*

PROOF We only study the case for an abstractor, since we can extend to applicators by duality (Proposition 2.2b, all reordering rules being symmetrical w.r.t. abstractors and applicators).

In $P\cdot {}^{\vee}_{am}\ x\cdot Q$, we have three possible positions of the reordering redex:

1. either the redex is included in P, and has no effect on the apparent position,
2. either it is included in $Q = Q_1\cdot R\cdot Q_2$, with R the redex, and we need to prove that $\phi(R') = \phi(R)$, since $\phi(Q) = \phi(Q_2)\circ\phi(R)\circ\phi(Q_1)$,
3. either it contains ${}^{\vee}_{am}\ x$.

We first prove that for any reordering redex R, $\phi(R')(ck) = \phi(R)(ck)$. We proceed by case on the rules:

- Symbolic rules. We only detail the first one: ${}^{\vee}_{bn}\ y\ {}^{\vee}_{am}\ x \to {}^{\vee}_{am}\ x\ {}^{\vee}_{bn}\ y$. If $c\neq a$ and $c\neq b$ then $\phi(R')(ck) = \phi(R)(ck) = ck$. If $c = a$ then $\phi(R')(ak) = \phi(R)(ak) = \phi({}^{\vee}_{am}\ x)(ak)$. Resp. for $c = b$.

  Others use the same argument (one interference in maximum, and invariant).

- Numeric rules. If $c\neq a$ then there is no interference: $\phi(R')(ck) = \phi(R)(ck) = ck$. Otherwise we calculate the interference case by case.

(4)	$\phi({}^{\vee}_{an}\ y\ {}^{\vee}_{am}\ x)(ak)$				$\phi({}^{\vee}_{am-1}\ x\ {}^{\vee}_{an}\ y)(ak)$
$k\geq m-1\geq n$	$\phi({}^{\vee}_{am}\ x)(ak+1)$	$=$	$ak+2$	$=$	$\phi({}^{\vee}_{an}\ y)(ak+1)$
$m-1>k\geq n$	$\phi({}^{\vee}_{am}\ x)(ak+1)$	$=$	$ak+1$	$=$	$\phi({}^{\vee}_{an}\ y)(ak)$
$m-1\geq n>k$	$\phi({}^{\vee}_{am}\ x)(ak)$	$=$	ak	$=$	$\phi({}^{\vee}_{an}\ y)(ak)$
(5)	$\phi({}^{\wedge}_{am}\ N_1\ {}^{\wedge}_{an}\ N_2)(ak)$				$\phi({}^{\wedge}_{an}\ N_2\ {}^{\wedge}_{am-1}\ N_1)(ak)$
$k>m>n$	$\phi({}^{\wedge}_{an}\ N_2)(ak-1)$	$=$	$ak-2$	$=$	$\phi({}^{\wedge}_{am-1}\ N_1)(ak-1)$
$m>k>n$	$\phi({}^{\wedge}_{an}\ N_2)(ak)$	$=$	$ak-1$	$=$	$\phi({}^{\wedge}_{am-1}\ N_1)(ak-1)$
$m>n>k$	$\phi({}^{\wedge}_{an}\ N_2)(ak)$	$=$	ak	$=$	$\phi({}^{\wedge}_{am-1}\ N_1)(ak)$
(6)	$\phi({}^{\vee}_{am}\ x\ {}^{\wedge}_{an}\ N)(ak)$				$\phi({}^{\wedge}_{an}\ N\ {}^{\vee}_{am-1}\ x)(ak)$
$k\geq m>n$	$\phi({}^{\wedge}_{an}\ N)(ak+1)$	$=$	ak	$=$	$\phi({}^{\vee}_{am-1}\ x)(ak-1)$
$m>k>n$	$\phi({}^{\wedge}_{an}\ N)(ak)$	$=$	$ak-1$	$=$	$\phi({}^{\vee}_{am-1}\ x)(ak-1)$
$m>n>k$	$\phi({}^{\wedge}_{an}\ N)(ak)$	$=$	ak	$=$	$\phi({}^{\vee}_{m-1}\ x)(ak)$
(7)	$\phi({}^{\vee}_{am}\ x\ {}^{\wedge}_{an}\ N)(ak)$				$\phi({}^{\wedge}_{an-1}\ N\ {}^{\vee}_{am}\ x)(ak)$
$k>n-1\geq m$	$\phi({}^{\wedge}_{an}\ N)(ak+1)$	$=$	ak	$=$	$\phi({}^{\vee}_{am}\ x)(ak-1)$
$n-1>k\geq m$	$\phi({}^{\wedge}_{an}\ N)(ak+1)$	$=$	$ak+1$	$=$	$\phi({}^{\vee}_{am-1}\ x)(ak)$
$n>m>k$	$\phi({}^{\wedge}_{an}\ N)(ak)$	$=$	ak	$=$	$\phi({}^{\vee}_{m-1}\ x)(ak)$

- Pseudo-reductions. For any pair $P = {}^{\vee}_{an}\ x\ {}^{\wedge}_{an}\ N$, $\phi(P) = Id$: this is clear for a label with a different symbolic part. Otherwise, either $m < n$, and $\phi(P)(am) = \phi({}^{\wedge}_{an}\ N)(am) = am$, or $m \geq n$, and $\phi(P)(am) = \phi({}^{\wedge}_{an}\ N)(am + 1) = am$.

  As a result, since pseudo-reductions do not change indexes, $\phi(R') = \phi(R) = \phi({}^{\wedge}_{q}\ N_2)$ (resp. $\phi({}^{\vee}_{q}\ y)$).

When the reordering redex contains ${}^{\vee}_{am}\ x$, we verify that they stay associated. Pseudo-reductions and symbolic reordering are clearly not a problem, since the former moves a locally assocated pair, with no effect on ϕ, and the later move an entity with a different symbolic part.

For the numeric rules we just remark that ϕ simulates a reduction moving ${}^{\vee}_{am}\ x$ outwards, using the rules bidirectionally. As such all steps are either the actual simulated step (outward) or its opposite (inward), and naturally static association is conserved. □

Corrolary 2.4 *Static associations are conserved by reordering. Static association is association.*

PROOF The spine is $P \cdot {}^{\vee}_{am}\ x \cdot Q \cdot {}^{\wedge}_{an}\ N \cdot R$.

When the reordering redex contains both ${}^{\vee}_{am}\ x$ and ${}^{\wedge}_{an}\ N$, this is a pseudo-reduction, and they stay locally associated, which is statically associated.

When the reordering redex is contained in $P \cdot {}^{\vee}_{am}\ x \cdot Q$, we apply Lemma 2.2 on it.

Otherwise we apply the lemma on $Q \cdot {}^{\wedge}_{an}\ N \cdot R$.

Moreover, by Proposition 2.1, in an order normal form all entities are either free or locally associated. So, statically associated entities associated.

Reciprocally, since statically free entities have an apparent position out of the spine, they cannot be associated. As such all associated entities are statically associated. □

With these lemma and corollary we easily prove the confluence of reordering modulo PRF equivalence, once we have termination.

Proposition 2.3 *The reordering system in Noetherian.*

PROOF We define a mesure on spines by the following ordered pair (remember that positions (ι_P) in a spine start from the right):

$$\begin{aligned}\mu(P) = \quad & (|\{(e,e') \in P \mid e \text{ abstractor}, e' \text{ applicator}, \iota_P(e) > \iota_P(e')\}|, \\ & |\{(e,e') \in P \mid e,e' \text{ abstractors}, \iota_P(e) > \iota_P(e'), \pi_{P[e,e']}(e) < \pi_{P[e,e']}(e')\}| \\ & +|\{(e,e') \in P \mid e,e' \text{ applicators}, \iota_P(e) > \iota_P(e'), \pi_{P[e,e']}(e) > \pi_{P[e,e']}(e')\}|)\end{aligned}$$

where $P[e,e']$ is the sub-spine extracted from P between e an e' (included). When one of the π's is not defined, the inequality is considered false.

Now we must prove that this mesure, as lexicographical ordering, decreases.

For the first term this is easy: only rules (3),(6),(7) and pseudo-reductions may change it, and they reduce it.

For the second one we must be more careful, because of the changing sub-spine in π. But we remark that $\phi(P)$ is strictly monotonous for any P, and by compositionality we can extend our sub-spine to the totality of the redex and get the same order. However, a special case arises when this is a pseudo-reduction implying e or e' as locally associated. We can no longer use ϕ. Actually, this may increase the term. However since the first term increased, this does not matter.

Moreover, as needed, rules (1),(4) and (2),(5) decrease respectively the left and right side of the sum, while clearly not changing the other.

Since all rules decrease that mesure, which is well-founded, reordering terminates. □

Theorem 2.2 *The reordering system is confluent modulo PRF equivalence.*

PROOF We prove this property spine by spine, since reordering keeps entities on the same spine (*cf.* Lemma 2.1).

Since by Proposition 2.3 we know that reordering terminates, we just have to prove that for any spine taken modulo PRF equivalence, its order normal form is unique modulo PRF equivalence.

By Lemma 2.2 and Corrolary 2.4, we know that both apparent positions for free entities, and associations for associated ones, are invariant by reordering. We verify easily that they are invariant by PRF equivalence too.

Moreover Proposition 2.1 gives us the structure of order normal forms. First free abstractor and free applicator parts are entirely specified by there apparent positions, and the ordering on labels. Then PRF equivalence says that the order of locally associated pairs is irrelevant, and the numeric part of their labels too. Since we know by Lemma 2.1 that the symbolic part does not change, that specifies the associated part modulo PRF equivalence. Since all reordering rules respect variable dependencies, they are kept.

As a result order normal forms are completely specified, modulo PRF equivalence, by the original term. □

2.3.4 Confluence of β-reordering

Definition 2.12 (β-Reordering) *A* β-reordering *step is a* β*-reduction step immediately followed by a stable reordering to order normal form.*

Since stable reordering can reduce every time reordering reduces, and reordering is Noetherian and confluent modulo PRF-equivalence, this completely defines a reduction rule on the quotient of order-normal λ-terms modulo PRF-equivalence. The one-step β-reordering relation is denoted as $\rightarrow_{\beta\downarrow}$.

Not to worry about renaming problems during reordering, we will assume that all free variables and abstraction variables have distinct names.

For Definition 2.12 to stand we have to prove that PRF-equivalent order-normal expressions are still equivalent after β-reordering. This is immediate, since PRF-equivalence and β-reduction are orthogonal: β-reduction do not separate any association pair, and PRF-equivalence do not separate any β-redex, nor change variable scoping.

This justifies us in the rest of this subsection, to consider no longer terms, but their equivalence classes modulo PRF equivalence.

From here on, the proof of β-reordering confluence follows the Martin-Lof-Tait scheme as in [HS86]. By *contracting* a β-redex, we mean applying the corresponding step of β-reduction.

Still, the definition of β-redex has to be changed to let us work with equivalence classes: with the classical one, two redexes could be different and mutually included in one another.

Definition 2.13 (β-redex) *In* $(\lambda_p x.(z \odot P))\,\widehat{}_{\,p}$ *only are included in the* β*-redex the associated pair,* z *and the entitities of* P *selected by the following process.*

1. *We start with* $V_0 = \{x\}$*, and parse from the right.*

2. *If the n^{th} entity is an applicator $\widehat{}_q N$, and $FV(N)$ contains variables in V_{n-1}, then we select it. Otherwise it is not part of the redex.*

3. *If the n^{th} entity is an abstractor $\overset{\vee}{}_q y$, if it is associated in P with a selected applicator, then we select it and $V_n = V_{n-1} \cup \{y\}$. Otherwise it is not part of the redex, and $V_n = V_{n-1} \setminus \{y\}$.*

With this definition we get out of the redex all entities which could escape by a reordering (for abstractors), or be inserted by a reordering (for applicators).

As a result, this definition allows us to use any selective λ-term as a representative for the equivalence class of its order normal form modulo PRF equivalence: reordering does not modify such β-redexes.

We note $M\downarrow$ this equivalence class for a selective λ-term M. However we will abbreviate it in simply M for the rest of this subsection (except Lemma 2.3, Corollary 2.5 and Proposition 2.4), and work modulo reordering, proving in Lemmas 2.3, 2.4, 2.6 and Corollary 2.5 that this does not give us wrong intuitions on the structure of reductions: Lemmas 2.3 and Corollary 2.5 proves that substitution and β-reordering commute with full reordering, Lemma 2.4 that we can still induce on the structure of terms before reoprdering, and Lemma 2.6, enunciated later, a property about postponment of reductions.

Lemma 2.3 (Substitution) *Reordering before or after a substitution does not change the result.*

$$[N/x]M = [N/x]M\downarrow$$

PROOF We shall only consider whether heads of spines will be substituted or not. In each spine where it is substituted, we can conclude by confluence of reordering (reordering the outer part of the spine and then introducing the end is equivalent to reordering directly the whole spinpe). In spines where it is not, there is no problem since they are left unmodified. □

Lemma 2.4 (Induction)
$$\begin{array}{l} M \to_{\beta\downarrow} M' \Rightarrow \lambda_p x.M \to_{\beta\downarrow} \lambda_p x.M' \\ M \to_{\beta\downarrow} M' \Rightarrow M \,\widehat{}_p N \to_{\beta\downarrow} M' \,\widehat{}_p N \\ N \to_{\beta\downarrow} N' \Rightarrow M \,\widehat{}_p N \to_{\beta\downarrow} M \,\widehat{}_p N' \end{array}$$

PROOF

1. $M = \lambda_{p_1} x_1 \ldots \lambda_{p_n} x_n.M_1$, with M_1 an order-normal form starting with an abstraction and $p_i \leq p_{i+1}$. So that $M' = \lambda_{p_1} x_1 \ldots \lambda_{p_n} x_n.M'_1$, with M'_1 any order-normal expression, and $M_1 \to_{\beta\downarrow} M'_1$. Hence

$$\begin{array}{lll} \lambda_p x.M & = & \lambda_{p_1} x_1 \ldots \lambda_{p'} x \ldots \lambda_{p_n} x_n.M_1 \\ & \to_{\beta\downarrow} & \lambda_{p_1} x_1 \ldots \lambda_{p'} x \ldots \lambda_{p_n} x_n.M'_1 \\ & = & \lambda_p x.M' \end{array}$$

2. If $\widehat{}_p N$ is associated then

$$\begin{array}{lll} M \,\widehat{}_p N & = & (\lambda_{p_1} x_1 \ldots \lambda_{p'} x \ldots \lambda_{p_n} x_n.M_1) \,\widehat{}_p N \quad M_1 \text{ as before}, p_i \leq p_{i+1} \\ & = & \lambda_{p_1} x_1 \ldots \lambda_{p_n} x_n.((\lambda_{p'} x.M_1) \,\widehat{}_{p'} N) \quad \text{(order normal form)} \\ & \to_{\beta\downarrow} & \lambda_{p_1} x_1 \ldots \lambda_{p_n} x_n.((\lambda_{p'} x.M'_1) \,\widehat{}_{p'} N) \\ & = & (\lambda_{p_1} x_1 \ldots \lambda_{p'} x \ldots \lambda_{p_n} x_n.M'_1) \,\widehat{}_p N \\ & = & M' \,\widehat{}_p N \end{array}$$

If $\widehat{}_p N$ is not associated then

$$\begin{array}{lll} M \widehat{}_p N & = & (\lambda_{p_1} x_1 \ldots \lambda_{p_n} x_n . (z \widehat{}_{q_1} N \ldots \widehat{}_{q_m} N_m \odot A)) \widehat{}_p N \\ & = & \lambda_{p_1} x_1 \ldots \lambda_{p'_n} x_n . (z \widehat{}_{q_1} N \ldots \widehat{}_{p'} N \widehat{}_{q'_m} N_m \odot A) \\ & \rightarrow_{\beta\downarrow} & \lambda_{p_1} x_1 \ldots \lambda_{p'_n} x_n . (Z' \widehat{}_{q_1} N'_1 \ldots \widehat{}_{p'} N \ldots \widehat{}_{q'_m} N'_m \odot A') \\ & = & (\lambda_{p_1} x_1 \ldots \lambda_{p_n} x_n . (Z' \widehat{}_{q_1} N'_1 \ldots \widehat{}_{q_m} N'_m \odot A')) \widehat{}_p N \\ & = & M' \widehat{}_p N \end{array}$$

where A is the entity associations, A' the reduced associations, and N is unchanged because all its variables are free.

3. By independence of spines.

□

Corrolary 2.5 (β-reduction) *Reordering before β-reduction does not change the order-normal form modulo PRF quivalence.*

$$M \rightarrow_R N \wedge M\downarrow \rightarrow_R N' \Rightarrow N\downarrow = N'\downarrow$$

where R is the redex reduced in $\rightarrow_R$.

PROOF Let $M_{\backslash R}$ be the context M without R, that is $M_{\backslash R}[R] = M$. By Lemma 2.4, $R \rightarrow_\beta R'$ implies $M_{\backslash R}[R]\downarrow \rightarrow_{\beta\downarrow} M_{\backslash R}[R']\downarrow$. But $M_{\backslash R}[R'] = N$ and $M_{\backslash R}[R']\downarrow = N'\downarrow$, so the equality stands. □

Definition 2.14 (Residuals) *Let R, S be β-redexes in a selective λ-term P. When R is contracted, let P change to P'. The* residuals *of S with respect to R are redexes in P', defined as follows:*

- *R, S are* non-overlapping parts *of P. Then contracting R leaves S unchanged. This unchanged S in P' is called the residual of S.*
- *$R = S$. Then contracting R is the same as contracting S. We say S has no residual in P'.*
- *R* is part of *S and $R \neq S$. Then S has form $(\lambda_p x.M) \widehat{}_p N$ and R is in M or in N. Contracting R changes M to M' or N to N', and S to $(\lambda_p x.M') \widehat{}_p N$ or $(\lambda_p x.M) \widehat{}_p N'$; this is the residual of S.*
- *S* is part of *R and $S \neq R$. Then R has form $(\lambda_p x.M) \widehat{}_p N$ and S is in M or in N. If S is in M, then $S' = [x/N]S$. If S is in N, then there are as many residuals S' of S as there were occurences of x in M, and $S' = S$.*

  This case will not happen in our proof.
- *$R \neq S$, but R is not part of S and S is not part of R. We can handle this as R part of S and R in M, by PRF equivalence.*

We can read this definition in two ways. If we are working with only β-reduction, without PRF equivalence nor reordering, then we adopt the usual definition of β-redex and do not need the last case.

Otherwise, we read all terms as non order-normal representatives of order-normal equivalence classes, and use the new definition of β-redex.

Note that anyway, in the first three cases (and the last too, since this is the third) S has at most one residual.

Before going on with our proof about β-reordering, we enunciate finite developments for β alone.

Proposition 2.4 (finite developments) *Let* R *be a set of β-redexes in M. Then reducing, in any order, of all residuals of redexes in* R *terminates and converges to the same M'.*

PROOF Since we use only β-reduction, finite developments for classical lambda calculus does apply. □

Now we are back to our proof about β-reordering.

Let $R_1, \ldots, R_n$ $(n \geq 0)$ be redexes in a term P. An R_i is called *minimal* iff it properly contains no other R_j (using our new definition of β-redex).

A *minimal complete development* (MCD) of $\{R_1, \ldots, R_n\}$ in P is a sequence of contractions on P performed as follows:

- First, contract any minimal R_i (say $i = 1$ for convenience). This leaves at most $n-1$ residuals $R'_2, \ldots, R'_n$, of $R_2, \ldots, R_n$.
- Then, contract any minimal R'_j. This leaves at most $n-2$ residuals.
- Repeat the above two steps until no residuals are left.

Note that this process is non-deterministic, and thus there are more than one such sequence of contractions.

Definition 2.15 (MCD) *Let P be a term as above, and Q a term. We write $P \triangleright_{mcd} Q$ iff Q is obtained from P by* minimal complete development *of the set $\{R_1, \ldots, R_n\}$.*

Note that if $M \triangleright_{mcd} M'$ and $N \triangleright_{mcd} N'$, then $M \mathbin{\widehat{p}} N \triangleright_{mcd} M' \mathbin{\widehat{p}} N'$. (*cf.* , Lemma 2.4)

Lemma 2.5 *If $M \triangleright_{mcd} M'$ and $N \triangleright_{mcd} N'$, then*

$$[N/x]M \triangleright_{mcd} [N'/x]M'.$$

PROOF We proceed by induction on M. Let $R_1, \ldots, R_n$ be the redexes developed in the given MCD of M.

1. $M = x$. Then n=0 and $M' = x$, so

$$[N/x]M = N \triangleright_{mcd} N' = [N'/x]M'.$$

2. $x \notin \mathrm{FV}(M)$. Then $x \notin \mathrm{FV}(M')$, so

$$[N/x]M = M \triangleright_{mcd} M' = [N'/x]M'.$$

3. $M = \lambda_p y.M_1$. Then each β-redex in M is in M_1, so M' has form $\lambda_p y.M'_1$ where $M_1 \triangleright_{mcd} M'_1$. Hence

$$\begin{array}{lcll} [N/x]M & = & [N/x](\lambda_p y.M_1) & \text{Lemma 2.3} \\ & = & \lambda_p y.[N/x]M_1 & \text{since } y \notin \mathrm{FV}(xN) \\ & \triangleright_{mcd} & \lambda_p y.[N'/x]M'_1 & \text{by induction hypothesis} \\ & = & [N'/x]M' & \text{since } y \notin \mathrm{FV}(xN') \end{array}$$

4. $M = M_1 \mathbin{\widehat{p}} M_2$ and each R_i is in M_1 or M_2. Then M' has form $M'_1 \mathbin{\widehat{p}} M'_2$ where $M_j \triangleright_{mcd} M'_j$ for $j = 1, 2$. Hence

$$\begin{array}{lcll} [N/x]M & = & ([N/x]M_1) \mathbin{\widehat{p}} ([N/x]M_2) & \text{Lemma 2.3} \\ & \triangleright_{mcd} & ([N'/x]M'_1) \mathbin{\widehat{p}} ([N'/x]M'_2) & \text{by ind. and note above} \\ & = & [N'/x]M'. & \end{array}$$

5. $M = (\lambda_p y.L)\,\widehat{p}\,Q$ and one R_i, say R_1, is M itself and is contracted last, and the others are in L or Q. (If it is not contracted last then we have $M = (\lambda_q z.K)\,\widehat{q}\,O$ too, and this one is contracted last). Hence the MCD has form

$$\begin{array}{lll} M = (\lambda_p y.L)\,\widehat{p}\,Q & \triangleright_{mcd} & (\lambda_p y.L')\,\widehat{p}\,Q' \quad (L \triangleright_{mcd} L', Q \triangleright_{mcd} Q') \\ & \rightarrow_{\beta\downarrow} & [Q'/y]L' \\ & = & M'. \end{array}$$

By induction hypothesis we have MCD's of $[N/x]L$ and $[N/x]Q$. Hence

$$\begin{array}{lll} [N/x]M & = & (\lambda_p y.[N/x]L)\,\widehat{p}\,([N/x]Q) \qquad \text{since } y \notin \mathrm{FV}(xN) \\ & \triangleright_{mcd} & (\lambda_p y.[N'/x]L')\,\widehat{p}\,([N'/x]Q') \quad \text{induction} \\ & \rightarrow_{\beta\downarrow} & [([N'/x]Q')/y][N'/x]L' \\ & = & [N'/x][Q'/y]L' \\ & = & [N'/x]M'. \end{array}$$

This reduction is an MCD, as required.

□

The following lemma is necessary because we are working on order normal forms: it is unclear wether $(\lambda_p x.M)\,\widehat{p}\,N$ will still be most external after reordering, but thanks to our definition of redex, it cannot be included in any other, so can be reduced last.

Lemma 2.6 (Proof induction) *If there is an MCD*

$$\begin{array}{lllll} P & = & (\lambda_p x.M)\,\widehat{p}\,N & \xrightarrow{*}_{\beta\downarrow} & (\lambda_p x.M')\,\widehat{p}\,N' \\ & & & \rightarrow_{\beta\downarrow} & [N'/x]M' \quad = \quad Q \\ & & & \xrightarrow{*}_{\beta\downarrow} & Q' \end{array}$$

then there is an MCD

$$\begin{array}{lllll} P & = & (\lambda_p x.M)\,\widehat{p}\,N & \xrightarrow{*}_{\beta\downarrow} & (\lambda_p x.M'')\,\widehat{p}\,N'' \\ & & & \rightarrow_{\beta\downarrow} & [N''/x]M'' \quad = \quad Q' \end{array}$$

That is, reduction of a potentially most external redex may be done last.

PROOF Since this is an MCD, new reductions do not apply on redexes created in the substitution, and Q' has form $[N''/x]M''$.

We should then just show that there are MCD's $M \triangleright_{mcd} M''$ and $N \triangleright_{mcd} N''$, which proves that $(\lambda_p x.M)\,\widehat{p}\,N \triangleright_{mcd} [N''/x]M$, by Lemma 2.5.

Each step of the original MCD after $[N'/x]M'$ only modifies either N' or M' at a time. So that we can write $M' \rightarrow_{\beta\downarrow} M_1 \rightarrow_{\beta\downarrow} \ldots \rightarrow_{\beta\downarrow} M''$, and since it is an MCD, $M' \triangleright_{mcd} M''$. Similarly $N' \triangleright_{mcd} N''$. And all the reductions performed are on the external level, that is permutable with our reduction on p in an MCD. So that $M \triangleright_{mcd} M''$ and $N \triangleright_{mcd} N''$. □

Lemma 2.7 (confluence of MCD) *If $P \triangleright_{mcd} A$ and $P \triangleright_{mcd} B$, then there exists T such that $A \triangleright_{mcd} T$ and $B \triangleright_{mcd} T$.*

PROOF By induction on P.

1. $P = x$. Then $A = B = P$. Choose $T = P$.

2. $P = \lambda_p x.P_1$. Then all β-redexes in P are in P_1, and

   $$A = \lambda_p x.A_1, \qquad B = \lambda_p x.B_1,$$

   where $P_1 \rhd_{mcd} A_1$ and $P_1 \rhd_{mcd} B_1$. By induction hypothesis there is a T_1 such that

   $$A_1 \rhd_{mcd} T_1, \qquad B_1 \rhd_{mcd} T_1.$$

   Choose $T = \lambda_p x.T_1$.

3. $P = P_1 \mathbin{\widehat{p}} P_2$ and all the redexes developed in the MCD's are in P_1, P_2. Then the induction hypothesis gives us T_1, T_2, and we choose $T_1 \mathbin{\widehat{p}} T_2$.

4. $P = (\lambda_p x.M) \mathbin{\widehat{p}} N$ and just one of the given MCD's involves contracting P's residual; say it is $P \rhd_{mcd} A$. Then, by Lemma 2.6, there is an MCD with form

   $$\begin{array}{lll} P & = & (\lambda_p x.M) \mathbin{\widehat{p}} N \\ & \rhd_{mcd} & (\lambda_p x.M') \mathbin{\widehat{p}} N' \quad (M \rhd_{mcd} M',\ N \rhd_{mcd} N') \\ & \rightarrow_{\beta\downarrow} & [N'/x]M' \\ & = & A. \end{array}$$

   And the other MCD has form

   $$\begin{array}{lll} P & = & (\lambda_p x.M) \mathbin{\widehat{p}} N \\ & \rhd_{mcd} & (\lambda_p x.M'') \mathbin{\widehat{p}} N'' \quad (M \rhd_{mcd} M'',\ N \rhd_{mcd} N'') \\ & = & B. \end{array}$$

   The induction hypothesis applied to M, N gives us M^+, N^+ such that

   $$\begin{array}{llll} M' \rhd_{mcd} M^+, & M'' \rhd_{mcd} M^+; \\ N' \rhd_{mcd} N^+, & N'' \rhd_{mcd} N^+. \end{array}$$

   Choose $T = [N^+/x]M^+$. Then there is an MCD from A to T, thus, by Lemma 2.5

   $$A = [N'/x]M' \rhd_{mcd} [N^+/x]M^+.$$

   And for B,

   $$\begin{array}{lll} B & = & (\lambda_p x.M'') \mathbin{\widehat{p}} N'' \\ & \rhd_{mcd} & (\lambda_p x.M^+) \mathbin{\widehat{p}} N^+ \\ & \rightarrow_{\beta\downarrow} & [N^+/x]M^+ \end{array}$$

5. $P = (\lambda_p x.M) \mathbin{\widehat{p}} N$ and both the given MCD's contract P's residual. Then (Lemma 2.6) we can give these MCD's form

   $$\begin{array}{lll} P & = & (\lambda_p x.M) \mathbin{\widehat{p}} N \\ & \rhd_{mcd} & (\lambda_p x.M') \mathbin{\widehat{p}} N' \\ & \rightarrow_{\beta\downarrow} & [N'/x]M' \\ & = & A, \end{array} \qquad \begin{array}{lll} P & = & (\lambda_p x.M) \mathbin{\widehat{p}} N \\ & \rhd_{mcd} & (\lambda_p x.M'') \mathbin{\widehat{p}} N'' \\ & \rightarrow_{\beta\downarrow} & [N''/x]M'' \\ & = & B. \end{array}$$

   Apply the induction hypothesis to M and N in case 4, and choose $T = [N^+/x]M^+$. Then Lemma 2.5 gives the result, as above.

□

Theorem 2.3 *β-reordering is confluent modulo PRF equivalence.*

$$P \xrightarrow{*}_{\beta\downarrow} M,\ P \xrightarrow{*}_{\beta\downarrow} N \Rightarrow (\exists T)\ M \xrightarrow{*}_{\beta\downarrow} T,\ N \xrightarrow{*}_{\beta\downarrow} T.$$

PROOF By induction on the length of the reduction from P to M, it is enough to prove

$$P \rightarrow_{\beta\downarrow},\ P \xrightarrow{*}_{\beta\downarrow} N \Rightarrow (\exists T)\ M \xrightarrow{*}_{\beta\downarrow} T,\ N \xrightarrow{*}_{\beta\downarrow} T.$$

Since a single β-reordering step is an MCD, it is sufficient to have

$$P \triangleright_{mcd} M,\ P \xrightarrow{*}_{\beta\downarrow} N \Rightarrow (\exists T)\ M \xrightarrow{*}_{\beta\downarrow} T, N \triangleright_{mcd} T.$$

which is shown by an induction on the number of β-steps from P to N. □

2.3.5 Confluence of selective λ-calculus

In this section, $\rightarrow$ (or $\rightarrow_\lambda$) denotes the union of β-reduction and ordering rules (label-selective λ-calculus), and $\rightarrow_\omega$ is the union of all rules (label-parallel system). We will now *no longer* consider terms modulo PRF equivalence, except in the β-reordering diamond of Figure 2.4.

Definition 2.16 (Normalized reduction) *For each label-parallel reduction $M_0 \rightarrow_\omega M_1 \rightarrow_\omega \ldots \rightarrow_\omega M_n$ we define its* normalized reduction $N_0 \rightarrow_{\beta\downarrow} N \rightarrow_{\beta\downarrow} \ldots \rightarrow_{\beta\downarrow} N_n$ *by taking for each N_i the order-normal form $M_i\downarrow$.*

Proposition 2.5 *Normalized reduction is a β-reordering.*

PROOF We should verify that we really obtain a β-reordering by this process.

We can first remark that, since we have Corollary 2.4, all β-redexes in M_i are still β-redexes in N_i.

If $M_i \rightarrow M_{i+1}$ is a reordering step, then $N_i = N_{i+1}$. Else, $M_i \rightarrow M_{i+1}$ is a β-step, and we should show $N_i \rightarrow_{\beta\downarrow} N_{i+1}$. From our remark, we have $N_i \rightarrow_{\beta\downarrow} N_i'$, reducing the same redex. We will in fact construct two parallel reorderings of M_i and M_{i+1}. First, a stable reordering of M_i, from $M_i^0 = M_i$ to $M_i^k = N_i$. With such a reordering, we have at each step $M_i^j \rightarrow M_i'^j$ by a β-step. Then we define a reordering of M_{i+1} going through all $M_i'^j$'s. By definition $M_i^j \rightarrow M_i^{j+1}$ does not separate two locally associated entities. There are four cases to consider:

1. If it is external to the reduced redex, then we can do the same reduction $M_i'^j \rightarrow_c M_i'^{j+1}$.
2. If it is internal, the β-reduction may only substitute some variables, but the reduction can still be applied. $M_i'^j \rightarrow_c M_i'^{j+1}$.
3. If it was an (a) or (b) reordering step over the redex, then it is superfluous after reduction, $M_i'^j = M_i'^{j+1}$.

Finally we can go from $M_i'^k$ to N_i' by a stable reordering. By confluence it gives $N_i' = N_{i+1}$, and the normalized reduction is correctly constructed. □

Theorem 2.4 (Confluence of label-parallel reduction) *The label-parallel system is confluent. Moreover, the converging reductions are* stable,

$$P \xrightarrow{*}_\omega M,\ P \xrightarrow{*}_\omega N \ \Rightarrow\ (\exists T)\ M \triangleright_{\omega stb} T,\ N \triangleright_{\omega stb} T.$$

PROOF We have

$$\begin{array}{llll} P & \to_\omega M_1 & \to_\omega \ldots \to_\omega M_m & = M, \\ P & \to_\omega N_1 & \to_\omega \ldots \to_\omega N_n & = N. \end{array}$$

So that we obtain normalized reductions

$$\begin{array}{ll} P' \to_{\beta\downarrow} M'_1 & \to_{\beta\downarrow} \ldots \to_{\beta\downarrow} M'_m, \\ P' \to_{\beta\downarrow} N'_1 & \to_{\beta\downarrow} \ldots \to_{\beta\downarrow} N'_n. \end{array}$$

And by confluence of β-reordering modulo PRF-equivalence,

$$\begin{array}{llll} M'_m & = R_0 & \to_{\beta\downarrow} R_1 \quad \to_{\beta\downarrow} \ldots \to_{\beta\downarrow} R_r & = T, \\ N'_n & = S_0 & \to_{\beta\downarrow} S_1 \quad \to_{\beta\downarrow} \ldots \to_{\beta\downarrow} S_s & = T'. \end{array}$$

with T and T' PRF-equivalent.

Since normalized (β-reordering) reductions are confluent, and all steps used here are in the stable label-parallel system, the label-parallel system is confluent modulo PRF equivalence, using stable reductions.

We can then reduce all the β-redexes present in the resulting term, thanks to Proposition 2.4, and obtain full confluence. All differences masked by PRF equivalence are contained in the redexes, and since in this last stage we do not use pseudo-reduction, we do not create new differences. □

Theorem 2.1 (Confluence of label-selective λ-calculus) *The label-selective λ-calculus is confluent. That is,*

$$P \xrightarrow{*} M,\ P \xrightarrow{*} N \ \Rightarrow\ (\exists T)\ M \xrightarrow{*} T,\ N \xrightarrow{*} T.$$

PROOF By Theorem 2.4,

$$\begin{array}{llll} M & = R_0 & \to_{\omega stb} R_1 \quad \to_{\omega stb} \ldots \to_{\omega stb} R_r & = T', \\ N & = S_0 & \to_{\omega stb} S_1 \quad \to_{\omega stb} \ldots \to_{\omega stb} S_s & = T'. \end{array}$$

But the absence of pseudo-reduction rules makes it impossible to follow these paths. Each time we have a (a) or (b) reduction, we should have a β-reduction in place.

We first define the set B_k of all residuals of β-redexes which where implied in an (a) or (b) pseudo-reduction. That is $B_0 = \emptyset$, $B_{k+1} = \{$residuals of B_k in $R_k \to R_{k+1}\}$ if this was not a pseudo-reduction, $B_{k+1} = \{$residuals of B_k in $R_k \to R_{k+1}\} \cup \{$the skipped β-redex$\}$ if it was. Since reductions are stable, residuals do not disappear.

Then we define R'_k as R_k where all redexes in B_k were reduced; Proposition 2.4 makes this definition correct. We have $R'_k \xrightarrow{*} R'_{k+1}$ where $\to$ is either the original $R_k \to R_{k+1}$ step applied on all its residuals, either $\to_\beta$ applied on $B_{k+1} \setminus B_k$ if it was a pseudo-reduction.

We define similarly S'_k.

We will finally have two expressions, coming from T' by β-reduction only. The number of β-reductions done may differ, but reducing all the redexes which were present in T' is enough, since the B_k's contain only residuals of β-redexes. That is,

$$\begin{array}{llll} M & \xrightarrow{*} R'_0 & \xrightarrow{*} R'_1 \quad \xrightarrow{*} \ldots \xrightarrow{*} R'_r & \xrightarrow{*}_\beta T, \\ N & \xrightarrow{*} S'_0 & \xrightarrow{*} S'_1 \quad \xrightarrow{*} \ldots \xrightarrow{*} S'_s & \xrightarrow{*}_\beta T. \end{array}$$

So that finally, $M \xrightarrow{*} T$ and $N \xrightarrow{*} T$. □

Figure 2.4 shows a schematic diagram of the process.

Corrolary 1,2,3 *Symbolic, numeric and flat selective λ-calculi are confluent.*

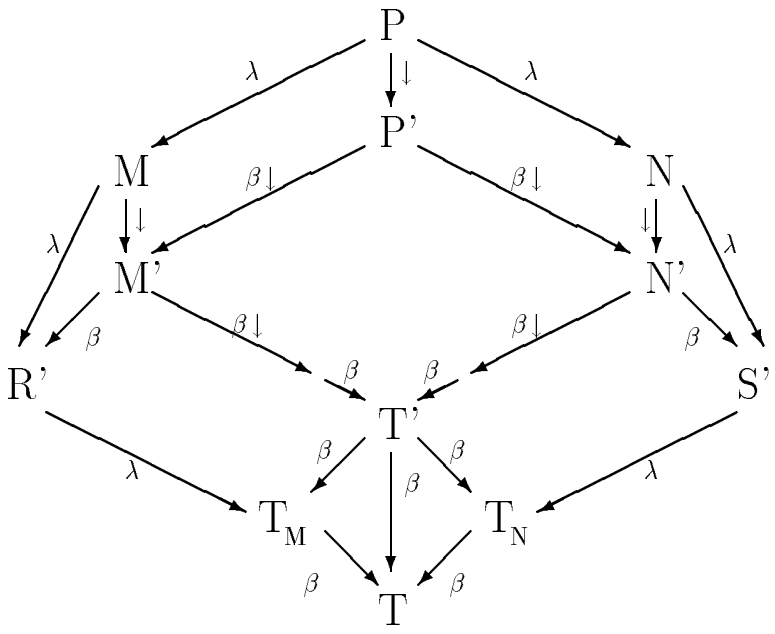


Figure 2.4: Schematic confluence of label-selective λ-calculus

PROOF For the numerical case, just take $\mathcal{S}$ with only one element.

For symbols, you just add the index 1 to all of them. No rule increases the index, and since all index are equal, no rule recreases them. As a consquence any reduction on a symbolic selective λ-term in selective λ-calculus uses only steps of the symbolic calculus, and we get confluence by injection.

For the flat case, take $\mathcal{S}' = \mathcal{S} \dot{\cup} \{\epsilon\}$, and define ϵ to be the least element of $\mathcal{S}'$. On the $\mathcal{S}$ part we can apply the argument for symbols, and get indexes on the extra symbol ϵ. Rules (1'), (2'), (3') are ensured by the extended order. □

Chapter 3

Streams

In the previous chapter we adopted a monadic notation for application in the selective λ-calculus. Since we have currying, this is enough to express any function, but we had to relate them to a more intuitive polyadic notation in paragraph 2.1.2. In this chapter we define a structure of *stream*, which then allows us to have a selective λ-calculus working directly in the polyadic notation.

3.1 Stream monoid

Selective λ-calculus, and its offsprings, can be defined in terms of operations on streams, that is a special kind of record allowing multiple occurences of different values on the same name. We define their set here, but we could in fact use any "reversible" monoid: the monoid operation, or concatenation, gives us uncurrying, while its inverse, or extraction, gives us currying.

We will note the concatenation on streams by a simple dot "$\cdot$", and the monoid of streams is $(\mathcal{S}, \cdot)$.

Preliminaries

- $\mathcal{L}_s$ is an ordered set of names, $\mathcal{N} = \mathbb{N} \setminus \{0\}$.
- $\mathcal{L} = \mathcal{L}_s \times \mathcal{N}$ is the set of labels, lexicographically ordered.
- In this section, l represents an element of $\mathcal{L}$; p, q elements of $\mathcal{L}_s$; m, n elements of $\mathcal{N}$.

Definition 3.1 (stream) *The set $\mathcal{S}(\mathcal{L}, \mathcal{A})$ of streams on a domain $\mathcal{A}$ is the set of finite partial functions from $\mathcal{L}$ to $\mathcal{A}$.*

$$\mathcal{S}(\mathcal{L}, \mathcal{A}) = \{s \in \mathcal{A}^{\mathcal{L}} \mid |\mathcal{D}_s| \in \mathbb{N}\}$$

Notations

- $\mathcal{D}_s$ is the definition domain of a stream s.
- $\{\}$ is the function defined nowhere ($\mathcal{D}_{\{\}} = \emptyset$).
- We note labels pn, and defining pairs $\{pn \Rightarrow a\}$ with $a \in \mathcal{A}$.
- The following equivalences of notation are admitted for labels and streams:
  - n denotes ϵn, p denotes $p1$
  - if $l = pn$ then $l + m = p(n + m)$
  - $(a_1, \ldots, a_n) = \{1 \Rightarrow a_1, \ldots, n \Rightarrow a_n\}$

Example 3.1 (stream monoid) *The simplest instanciation of $\mathcal{S}$ is the monoid of tuples: $\mathcal{L}_s = \{\epsilon\}$ (a singleton) and $\mathcal{S} = \bigcup_{n\geq 0} \mathcal{A}^{[\![1,n]\!]}$. Then concatenation is*

$$(a_1, \ldots, a_m) \cdot (b_1, \ldots, b_n) = (a_1, \ldots, a_m, b_1, \ldots, b_n),$$

and is reversible.

This example gives implicit currying, without selectivity. In the general case, we need a more complex definition, permitting label operations. It is based on a notion of n^{th} free position in a stream, which, while intuitively clear —just imagine that undefined labels point to free positions—, looks a little complex once formalized.

Definition 3.2 (free position) *1. The n^{th} position on p in a stream r is said to be* occupied *if $pn \in \mathcal{D}_r$. It is* free *otherwise, and $\mathcal{F}_r = \mathcal{L} \setminus \mathcal{D}_r$ is the set of these free positions.*

2. The n^{th} free position *for p in r is the n^{th} element of $\{i \mid pi \in \mathcal{F}_r\}$. Namely $\phi_{r,p}(n) = \min\{m \mid |\{pi \in \mathcal{F}_r \mid i \leq m\}| = n\}$.*

3. The relative index *of pn in r is the number of free positions preceding n on p plus one. Namely $\psi_{r,p}(n) = |\{pi \in \mathcal{F}_r \mid i < n\}| + 1$.*

One notices immediately the inversion relation between free positions, used for concatenation, and relative indexes, used for extraction.

$$\begin{array}{rcl} \phi_{r,p}(n) & = & \min\{m \mid |\{pi \in \mathcal{F}_r \mid i \leq n\}| = n\} \\ & = & \max\{m \mid |\{pi \in \mathcal{F}_r \mid i < n\}| = n-1\} \\ & = & \max \psi_{r,p}^{-1}(n) \end{array}$$

We extend ϕ to streams by $\phi_r(\{p_i n_i \Rightarrow a_i\}_{i=1}^k) = \{p_i \phi_{r,p_i}(n_i) \Rightarrow a_i\}_{i=1}^k$, and respectively for ψ.

Example 3.2 (free positions) *In $\{p1 \Rightarrow a, p3 \Rightarrow b, p5 \Rightarrow c, q2 \Rightarrow d\}$, relative indexes are respectively 2 for $p4$ and $q3$, and 3 for $p5$ and $q4$. Free positions are $\{2,4,6,7,\ldots\}$ on p, and $\{1,3,4,\ldots\}$ on q. As a result, the second free position on p is 4, and on q this is 3.*

Proposition 3.1 (reversibility) *ϕ_r is a bijection from $\mathcal{S}$ to $\{s \in \mathcal{S} \mid \mathcal{D}_r \cap \mathcal{D}_s = \emptyset\}$. $\psi_r \circ \phi_r = id_{\mathcal{S}}$.*

PROOF For each label pn, by definition of ϕ we have,

$$\psi_{r,p}(\phi_{r,p}(n)) = \psi_{r,p}(\max\{i \mid \psi_{r,p}(i) = n\}) = n.$$

Moreover, since $\psi_{r,p}$ is an increasing surjection, we have,

$$\begin{array}{rl} & (\exists m)\ \phi_{r,p}(m) = n \\ \Leftrightarrow & \psi_{r,p}(n) < \psi_{r,p}(n+1), \text{ by definition of } \phi_{r,p} \\ \Leftrightarrow & |\{pi \in \mathcal{F}_r \mid i < n\}| < |\{pi \in \mathcal{F}_r \mid i < n+1\}| \\ \Leftrightarrow & pn \in \mathcal{F}_r \\ \Leftrightarrow & pn \notin \mathcal{D}_r, \end{array}$$

and we can conclude that $\phi_{r,p}(\mathcal{N}) = \{n \in \mathcal{N} \mid pn \notin \mathcal{D}_r\}$, to obtain $\phi_r(\mathcal{S}) = \{s \in \mathcal{S} \mid \mathcal{D}_r \cap \mathcal{D}_s = \emptyset\}$ by extension. □

Definition 3.3 (concatenation and extraction)

1. *Stream concatenation is defined as* $r \cdot s = r \uplus \phi_r(s)$ *where "*$\uplus$*" denotes union of (set represented) functions on disjoint domains.*

2. *Sub-stream extraction is defined as* $r \uplus s = r \cdot \psi_r(s)$*, where* r *is the extracted sub-stream and* $\psi_r(s)$ *is the rest after extraction.*

Proposition 3.2 (monoid) *Concatenation as in definition 3.2 is an associative application* $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$*, accepting* $\{\}$ *as neutral element.*

PROOF Associativity comes from the equality $\phi_r \circ \phi_s = \phi_{r \uplus \phi_r(s)}$. For this we reason on inverses, pn being a free position of $r \cdot s$:

$$\begin{array}{lll}
 & \psi_{s,p}(\psi_{r,p}(n)) & \\
= & 1 + |\{pi \in \mathcal{F}_s \mid i < \psi_{r,p}(n)\}| & \\
= & 1 + |\{pi \in \mathcal{F}_s \mid \phi_{r,p}(i) < \phi_{r,p}(\psi_{r,p}(n))\}|, & \\
 & \qquad \text{since } \phi_{r,p} \text{ is strictly growing} & \\
= & 1 + |\{pi \in \phi_r(\mathcal{F}_s) \mid i < \phi_{r,p}(\psi_{r,p}(n))\}|, & \\
 & \qquad\qquad \text{since } \phi_r \text{ is an injection} & \\
= & 1 + |\{pi \in \phi_r(\mathcal{F}_s) \mid i < n\}|, & \\
 & \qquad \text{since } pn \text{ is a free position of } r & \\
= & 1 + |\{pi \in \mathcal{F}_r \cap \mathcal{F}_{\phi_r(s)} \mid i < n\}| & \\
= & 1 + |\{pi \in \mathcal{F}_{r \cdot s} \mid i < n\}| & \\
= & \psi_{r \cdot s,p}(n) &
\end{array}$$

We then have $r \cdot (s \cdot t) = r \uplus \phi_r(s \uplus \phi_s(t)) = r \uplus \phi_r(s) \uplus \phi_r(\phi_s(t)) = r \uplus \phi_r(s) \uplus \phi_{r \uplus \phi_r(s)}(t) = (r \cdot s) \cdot t$. $r \cdot \{\} = r = \{\} \cdot r$ is immediate ($\phi_{\{\}} = id$). □

The intuition behind these definitions is that when we do $r \cdot s$ we insert elements of s at free positions in r: for each name p we insert the element whose index is n in s at the n^{th} free position for p in r. ϕ_r is the function which does this shifting. Reciprocally, extraction uses ψ_r to shift back positions in the rest to their relative indexes w.r.t r.

A consequence of the above propositions is that we can equivalently work only on "exploded" streams since they generate all streams. Indexes vary according to their ordering.

$$\begin{array}{lcll}
\{pm \Rightarrow a\} \cdot \{qn \Rightarrow b\} & = & \{qn \Rightarrow b\} \cdot \{pm \Rightarrow a\} & p \neq q \\
\{pm \Rightarrow a\} \cdot \{pn \Rightarrow b\} & = & \{pn \Rightarrow b\} \cdot \{pm - 1 \Rightarrow a\} & m > n
\end{array}$$

This gives us back our original monadic definition of selective λ-calculus, with its strange, but now explained, shifts on the numerical part of labels.

Last, two small examples of stream concatenation, to get used to it.

Example 3.3 (Concatenation)

$$\begin{array}{l}
\{2 \Rightarrow a\} \cdot (b,c) \cdot \{p \Rightarrow d\} \cdot \{q \Rightarrow e\} \cdot \{p \Rightarrow f\} \\
\qquad = (b,a,c) \cdot \{p \Rightarrow d\} \cdot \{p \Rightarrow f\} \cdot \{q \Rightarrow e\} \\
\qquad = \{\epsilon 1 \Rightarrow b, \epsilon 2 \Rightarrow a, \epsilon 3 \Rightarrow c, p1 \Rightarrow d, p2 \Rightarrow f, q1 \Rightarrow e\}.
\end{array}$$

$$\begin{array}{l}
\{p1 \Rightarrow a, p3 \Rightarrow b, r1 \Rightarrow c\} \cdot \{p1 \Rightarrow d, q_2 \Rightarrow e\} \\
\qquad = \{p1 \Rightarrow a, p3 \Rightarrow b\} \cdot \{p1 \Rightarrow d\} \cdot \{q2 \Rightarrow e\} \cdot \{r1 \Rightarrow c\} \\
\qquad = \{p1 \Rightarrow a, p2 \Rightarrow d, p3 \Rightarrow b, q2 \Rightarrow e, r1 \Rightarrow c\}
\end{array}$$

$$
\begin{array}{rcll}
l & ::= & pn & p \in \mathcal{S}, n \in \mathcal{N} \\
M & ::= & x & \text{variable} \\
 & | & \lambda\{l \Rightarrow x, \ldots\}.M & \text{abstraction} \\
 & | & M\ \{l \Rightarrow M, \ldots\} & \text{application}
\end{array}
$$

Figure 3.1: Stream syntax of selective λ-calculus

3.2 Stream syntax of selective λ-calculus

We define here a new syntax for selective λ-calculus. It takes advantage of the previous confluence proof to consider basic reorderings as equivalences, and reduces it to a single rule. By using streams in the syntax it is more intuitive, and makes easier the definition of a typing system in the next chapter.

3.2.1 Terms

In the following definitions we will use the abbreviation $A \not\frown B$ for $A \cap B = \emptyset$.

Definition 3.4 (stream syntax) *Terms of selective λ-calculus in stream syntax are given in figure 3.1. We consider them modulo α conversion and the following three structural equivalences:*

$$
\begin{array}{rcll}
S.R.M & \equiv_{\cdot} & (R \cdot S).M & \\
\lambda R.\lambda S.M & \equiv_{\lambda} & \lambda(S \cdot R).M & V(R) \not\frown V(S) \\
R.\lambda S.M & \equiv_{\cdot\lambda} & \lambda\psi_R(S).\psi_S(R).M & FV(R) \not\frown V(S), \mathcal{D}_R \not\frown \mathcal{D}_S
\end{array}
$$

For a stream of variables, V is its image (second projection).

As we had entity syntax for the original calculus, we can write application in stream form as $\{l \Rightarrow N, \ldots\}.M$ for a left to right flow.

These three equalities are just generalizing the three reordering rules of the symbolic selective λ-calculus (*cf.* 2.1.3) to streams.

Last we redefine the notions of head and spine, which we will use in proofs.

Definition 3.5 (head and spine) *Any selective λ-term in stream syntax can be divided in its* head *(x) and* spine *($e_1.\cdots.e_n$),*

$$e_1.\cdots.e_n.x = e_1.\cdots.e_n{:}x$$

where e_i's are either applicators $\{l \Rightarrow N, \ldots\}$ *or* abstractors $\lambda\{l \Rightarrow x, \ldots\}$*, and $e_1.\cdots.e_n$ an* entity sequence*. We note $s.s'$ the concatenation of two entity sequences s and s'.*

3.2.2 β-reduction

Using that syntax, we can make evident that the only real reduction rule of selective λ-calculus is β-reduction.

Definition 3.6 (polyadic selective λ-calculus) *The polyadic selective λ-calculus is the following β-reduction applied to equivalence classes of stream syntax selective λ-terms by their structural rules.*

$$(\beta) \qquad \{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M \to [N/x]M.$$

Confluence in this new syntax is immediate.

Theorem 3.1 *Selective λ-calculus in stream syntax is confluent.*

$$\forall M,P,Q\ (M \xrightarrow{*} P \wedge M \xrightarrow{*} Q) \Rightarrow (\exists T\ P \xrightarrow{*} T \wedge Q \xrightarrow{*} T)$$

PROOF We already proved the confluence of the calculus including reordering rules. Making them bidirectional keeps confluence, and so is it when we make them equivalences. □

Here are some examples of reductions in the new syntax.

Example 3.4 (Stream syntax) *We use here the left-to-right entity notation.*

$$\begin{array}{ll} & \{p3 \Rightarrow N, p5 \Rightarrow N'\}.\lambda\{p1 \Rightarrow x, p3 \Rightarrow y, p4 \Rightarrow z\}.M \\ \equiv & \{p4 \Rightarrow N'\}.\{p3 \Rightarrow N\}.\lambda\{p3 \Rightarrow y\}.\lambda\{p1 \Rightarrow x, p3 \Rightarrow z\}.M \\ \rightarrow & \{p4 \Rightarrow N'\}.\lambda\{p1 \Rightarrow x, p3 \Rightarrow z\}.[N/y]M \\ \rightarrow & \lambda\{p1 \Rightarrow x, p3 \Rightarrow z\}.\{p2 \Rightarrow N'\}.[N/y]M \end{array}$$

We omitted eventual substitutions of bound variables.

3.2.3 η-reduction

η-reduction cannot be defined in the selective λ-calculus of Chapter 2, since the classical definition

$$M \hat{l} x \overset{\vee}{l} x \rightarrow_\eta M \qquad\qquad x \notin FV(M)$$

gives a system that is clearly not Church-Rosser when combined with other rules. The reason is that reordering rules may introduce entities between the two concerned.

However, since in the stream syntax we use structural equivalences in place of reordering, this can be defined.

Definition 3.7 (η-reduction) *For a selective λ-term M, variable x and label p, such that x is not free in M, η-reduction is*

$$(\eta) \qquad \lambda\{l \Rightarrow x\}.\{l \Rightarrow x\}.M \rightarrow_\eta M.$$

3.2.4 Bohm separation theorem

Theorem 3.2 *For any two selective λ-terms M and N in $\beta\eta$-normal form, such that $M \not\equiv N$, there is an entity sequence P such that, for two different variables x and y,*

$$\begin{array}{l} (P{:}M) \downarrow = x, \\ (P{:}N) \downarrow = y. \end{array}$$

PROOF We prove it by induction on the depth of the first different head, that is either a free and a bound variable, or different free variables, or variables bound by different labels.

- M and N have different heads.

  In a first step we un-abstract the two terms, by giving them a context of applications to different fresh variables for each label contained in one of them. That is, if $M = \lambda\{l_1 \Rightarrow x_1, \ldots l_m \Rightarrow x_m\}.M'$ and $N = \lambda\{l'_1 \Rightarrow y_1, \ldots, l'_n \Rightarrow y_n\}.N'$, then $P = \{l_1 \Rightarrow z_1, \ldots, l_m \Rightarrow z_n, l'_1 \Rightarrow z'_1, \ldots l'_n \Rightarrow z'_n\}$ (l'_i does not appear if there is l_j such that $l'_i = l_j$).

The two resulting terms are normalizable since we apply only on variables, and heads of $(P_1{:}M)\downarrow$ and $(P_1{:}N)\downarrow$ are now linked to distinct free variables.

Say $(P_1{:}M)\downarrow = \lambda\{l_1 \Rightarrow M_1, \ldots, l_m \Rightarrow M_m\}.u$. The context P_2 we should apply then is $\{l \Rightarrow \lambda\{l_1 \Rightarrow r_1, \ldots, l_m \Rightarrow M_m\}.u\}.\lambda\{l \Rightarrow u\}$, r_i's being fresh. After this operation we have $(P_2.P_1{:}M)\downarrow = u$ and $(P_2.P_1{:}N)\downarrow = \lambda\{p_1 \Rightarrow y_1, \ldots, p_k \Rightarrow y_k\}.\{q_1 \Rightarrow N_1, \ldots q_n \Rightarrow N_n\}.v$, where v is free. (we have still a normal form since we only applied on variables.)

In the third step we construct P_3 like $P_2.P_1$ to eliminate abstractors in $P_2.P_1{:}N$ with applicators, and substitute u and v to eliminate applicators.

We have finally $(P{:}M)\downarrow = u$ and $(P{:}N)\downarrow = v$ for $P = P_3.P_2.P_1$.

- M and N have same head x.

  They have form $\lambda\{p_1 \Rightarrow x_1, \ldots, p_m \Rightarrow x_m\}.\{p'_1 \Rightarrow M_1, \ldots, p'_{m'} \Rightarrow M_{m'}\}.x$ and $\lambda\{q_1 \Rightarrow y_1, \ldots, q_n \Rightarrow y_n\}.\{q'_1 \Rightarrow N_1, \ldots, q'_{n'} \Rightarrow N_{n'}\}.x$, and either $x = x_i = y_j$ and $p_i = q_j$, or x is free in both M and N. In the first case, we free it by applying to $\{p_i \Rightarrow x\}$. Then, in both case, we substitute it with $\lambda\{h \Rightarrow x\}.x$, where h is a name not used in M and N: this gives us a way to change the head variable without changing its other occurences. We now speak of M and N modified in this way.

  The difference is in some entity. We look for it.

  - There is some p'_i which differs from all q'_j's. We then use the entity sequence $P = \{r \Rightarrow \lambda\{p_i \Rightarrow x\}.x\}$. Then $(P{:}M)\downarrow$ and $(P{:}N)\downarrow$ have different heads (we are in η-normal form), and we can use the previous method.
  - $m' = n'$ and for all i, $p'_i = q'_i$, but there is some p_i which differs from all q_j's. We then use the entity sequence $P = \{p_i \Rightarrow u, h \Rightarrow \lambda\{\phi_{\{q'_i\}_{j=1}^{n'}}(\psi_{\{q_j\}_{j=1}^{n}}(p_i)) \Rightarrow v\}.v\}$. Then $(P{:}M)\downarrow$ and $(P{:}N)\downarrow$ have different heads (u for the second, an abstracted variable for the first), and we can use the previous method.
  - For all labels M and N have the same spine structure. We can suppose $p_i = q_i$ and $x_i = y_i$. First we free these variables, by applying to $P_1 = \{p_1 \Rightarrow x_1, \ldots, p_n \Rightarrow x_n\}$. Then the difference must be in some applicator, say the application on p'_i. We select it by $\{h \Rightarrow \lambda\{p_i \Rightarrow x\}.\{l_1 \Rightarrow z_1, \ldots, l_k \Rightarrow z_k\}.x$. The fresh variables z_i are there not to let the value passed through x "eat" other applications. Either the new head variables are different, and we are finished, either they are identical, but we can go on, having lowered the level of the difference by one.

□

Theorem 3.3 *For two closed selective λ-terms, that is selective combinators, in $\beta\eta$-normal form, if $M \not\equiv N$, then there is an entity sequence P, composed only of applications on closed terms, such that*

$$\begin{aligned}(\{p \Rightarrow x, q \Rightarrow y\}.P{:}M)\downarrow &= x,\\ (\{p \Rightarrow x, q \Rightarrow y\}.P{:}N)\downarrow &= y.\end{aligned}$$

PROOF We construct first an environment P_0 by the preceding theorem. Since we will apply it on a closed term, we can normalize it directly by the rule β on entity sequences: $P.\{p \Rightarrow M\}.\lambda\{p \Rightarrow x\}.Q \to P.[M/x]Q$, and normalization of subterms. The result $P_1 = P_0\downarrow$ will not contain abstractions, since they would be external, and wouldn't disappear in $(P{:}M)\downarrow$. We substitute in this result x by $\lambda\{p \Rightarrow x, q \Rightarrow y\}.x$, y by $\lambda\{p \Rightarrow x, q \Rightarrow y\}.y$ and all other free variables by any closed term, and get P. □

Chapter 4

Typed selective λ-calculus

Now that we have the stream syntax, introducing types become easy. Basically, we just have to replace usual types by stream types on the left side of arrows, and everything becomes very similar to typed λ-calculus. We get easily a strong normalization theorem, and are even able to infer polymorphic types, *a la* ML. We conclude showing how such a parameter passing mechanism could be used in a strongly typed functional programming language.

4.1 Simple types

We introduce here simple types like in classical λ-calculus. In doing so we have two goals. The first one is to gain a better understanding of the calculus itself, by seeing which type structure it involves. The second one is to verify that selective λ-calculus keeps all good properties of the classical one, like strong normalization for typable terms.

We define our types by a grammar distinguishing between base types and general types,

l	::=	pn	labels
u	::=	$u_1 \mid u_2 \mid \ldots$	base types
t	::=	$\{l \Rightarrow t, \ldots\} \rightarrow u$	types

where the set $\{l \Rightarrow t, \ldots\}$ denotes a finite partial function from $\mathcal{L}$ to types. We identify base types and their images $\{\} \rightarrow u$ in general types.

The idea in writing types like that, is that an application can be done indifferently on any label present in the type, on a value of corresponding type. This makes type inference quite intuitive.

The original syntax of terms is extended in

$$M ::= x \mid \lambda\{l \Rightarrow x{:}t, \ldots\}.M \mid M\ \{l \Rightarrow M, \ldots\}.$$

which requires any abstracted variable to be explicitly typed.

A term M is well typed if there is a mapping Γ from the free variables of M to types and a type τ such that

$$\Gamma \vdash M : \tau$$

is deducible in the type inference system of figure 4.1. It supposes the following definition of stream type concatenation.

Simply typed selective λ-calculus verifies the two fundamental properties of typed lambda-calculi.

$$\Gamma[x \mapsto \tau] \vdash x : \tau \tag{I}$$

$$\frac{\Gamma[x_1 \mapsto \theta_1, \ldots] \vdash M : r \to \tau}{\Gamma \vdash \lambda\{l_1 \Rightarrow x_1{:}\theta_1, \ldots\}.M : \{l_1 \Rightarrow \theta_1, \ldots\} \cdot r \to \tau} \tag{II}$$

$$\frac{\Gamma \vdash M : \{l_1 \Rightarrow \theta_1, \ldots\} \cdot r \to \tau \quad \Gamma \vdash N_i : \theta_i}{\Gamma \vdash M\ \{l_1 \Rightarrow N_1, \ldots\} : r \to \tau} \tag{III}$$

Figure 4.1: Typing rules for the simply typed calculus

Proposition 4.1 (subject reduction) *If* $\Gamma \vdash M : \tau$ *and* $M \to N$ *then* $\Gamma \vdash N : \tau$.

PROOF We first remark that structural equalities are compatible with typing rules: we use the stream structure in the same way.

However one case is not included in the simple stream structure: the third structural equivalence, when abstraction and application exchange their positions.

In that case, M is of the form $R{:}\lambda S.P$, with $S = \{l_1 \Rightarrow x_1{:}\theta_1, \ldots\}$ and $R = \{l'_1 \Rightarrow Q_1, \ldots\}$, $\mathcal{D}_R \not\parallel \mathcal{D}_S$. The basis of the proof tree is

$$\frac{\dfrac{\Gamma[x_1 \mapsto \theta_1, \ldots] \vdash P : \psi_S(\{l'_1 \Rightarrow \theta'_1, \ldots\}) \cdot r \to \tau}{\Gamma \vdash \lambda S.P : \{l_1 \Rightarrow \theta_1, \ldots, l'_1 \Rightarrow \theta'_1, \ldots\} \cdot r \to \tau} \quad \Gamma \vdash Q_i : \theta'_i}{\Gamma \vdash M : \psi_R(\{l_1 \Rightarrow \theta_1, \ldots\}) \cdot r \to \tau}$$

which gives after reordering

$$\frac{\dfrac{\Gamma[x_1 \mapsto \theta_1, \ldots] \vdash P : \psi_S(\{l'_1 \Rightarrow \theta'_1, \ldots\}) \cdot r \to \tau \quad \Gamma \vdash Q_i : \theta'_i}{\Gamma[x_1 \mapsto \theta_1, \ldots] \vdash P \cdot \psi_S(R) : r \to \tau}}{\Gamma \vdash \lambda\psi_R(S).\psi_S(R){:}P : \psi_R(\{l_1 \Rightarrow \theta_1, \ldots\}) \cdot r \to \tau}$$

Then we only need to prove this property when M is a redex and N is the result of this reduction. We can then generalize by substitution and repetition.

If M is a β-redex it is of the form $(\lambda\{l_1 \Rightarrow x_1{:}\theta_1, \ldots\}.P)\ \{l_1 \Rightarrow Q_1, \ldots\}$. Then the basis of the proof tree is:

$$\frac{\dfrac{\Gamma[x_1 \mapsto \theta_1, \ldots] \vdash P : r \to \tau}{\Gamma \vdash \lambda\{l_1 \Rightarrow x_1{:}\theta_1, \ldots\}.P : \{l_1 \Rightarrow \theta_1, \ldots\} \cdot r \to \tau} \quad \Gamma \vdash Q_i : \theta_i}{\Gamma \vdash M : r \to \tau}$$

After reduction the result is $N = P[x_1 \backslash Q_1, \ldots]$. We obtain a derivation tree for $\Gamma \vdash P[x_1 \backslash Q_1, \ldots]$ from those of $\Gamma[x_1 \mapsto \theta_1, \ldots] \vdash P : r \to \tau$ and $\Gamma \vdash Q_i : \theta_i$ by (1) do all α-conversions necessary to the substitution of x_i's by Q_i's; (2) suppressing x_i in the environments (except where it is redefined by an abstraction); (3) where x_i appears without being defined in the environment, replace $\Gamma' \vdash x_i : \theta_i$ by the derivation tree of $\Gamma' \vdash Q_i : \theta_i$ (no problem since $\forall y \in FV(Q_1 \ldots Q_n)\ \Gamma(y) = \Gamma'(y)$). □

Theorem 4.1 *The simply typed selective* λ*-calculus is strongly normalizing.*

PROOF The idea is to actually construct a function that gives the longest reduction of a term in function of its input. By reduction steps we only count here β-reductions, since we already know that reordering is Noetherian.

Before doing that, we define zero functions, and the rectification of a function. In fact we use *selective functions* in place of classical functions, labeling arguments. They are only a practical notation since we know that selection is deterministic by the confluence theorem; and we could translate them to classical functions using their types and the order on labels.

$\tau = \{l_1 \Rightarrow \tau_1, \ldots, l_n \Rightarrow \tau_n\} \rightarrow u$ is a simple type. The *zero-function* for τ, noted 0^τ, is the function $\lambda\{l_1 \Rightarrow x_1{:}\tau_1^*, \ldots, l_n \Rightarrow x_n{:}\tau_n^*\}.0$, of type τ^*, where $*$ is defined by induction as $(\{l_1 \Rightarrow \tau_1, \ldots\} \rightarrow u)^* = \{l_1 \Rightarrow \tau_1^*, \ldots\} \rightarrow int$ (we replace every base type with int).

To *rectify* a function f of type $\tau = \{l_1 \Rightarrow \tau_1, \ldots, l_n \Rightarrow \tau_m\} \cdot r \rightarrow u$ to $r \rightarrow u$ you apply it to the corresponding zero-functions: $rect(r \rightarrow u, f : \tau) = f\ \{l_1 \Rightarrow 0^{\tau_1}, \ldots, l_n \Rightarrow 0^{\tau_n}\}$.

We define our function $T_\Gamma(M)$ by induction on the structure of the term M, annotated with types in some typing environment Γ. We suppose that keywords $\mathcal{S}$ and variables $\mathcal{V}$ are independent, and use $\mathcal{V} \cup \mathcal{S}$ as symbols for the respective selective functions. We only consider abstraction and applications on one label, by structural equivalence.

1. for a variable $\Gamma \vdash x : \tau$, the associated function is $\lambda\{x \Rightarrow x{:}\tau^*\}.x$.

2. for an abstraction $\Gamma \vdash \lambda\{l \Rightarrow x{:}\theta\}.M : \{l \Rightarrow \theta\} \cdot r \rightarrow \tau$, the associated function is $\lambda\{l \Rightarrow x{:}\theta^*\}.\{x \Rightarrow x\}{:}T_{\Gamma[x \mapsto \theta]}(M)$ if $x \in FV(M)$, $\lambda\{l \Rightarrow x{:}\theta^*\}.T_\Gamma(M)$ otherwise.

3. for an application $\Gamma \vdash M\ \{l \Rightarrow N\} : r \rightarrow \tau$, with $\Gamma \vdash N : \theta$, the associated function is, $(x_1, \ldots x_n)$ being the free variables of N, of which $(x_1, \ldots x_k)(k \leq n)$ are free in M too,

$$T_\Gamma(M\{l \Rightarrow N\}) = \lambda\{x_1 \Rightarrow x_1{:}\Gamma(x_1), \ldots, x_n \Rightarrow x_n{:}\Gamma(x_n)\}.$$
$$(T_\Gamma(M)\ \{x_i \Rightarrow x_i \mid 1 \leq i \leq k, x_i \in FV(M)\}\ \{l \Rightarrow N^a\} + rect(int, N^a : \theta^*) + 1)$$

   where $N^a = T_\Gamma(N)\ \{x_1 \Rightarrow x_1, \ldots, x_n \Rightarrow x_n\}$, and, for $f : \{l_1 \Rightarrow \theta_1, \ldots, l_n \Rightarrow \theta_n\} \rightarrow int$ and $a : int$, $f + a = \lambda\{l_1 \Rightarrow x_1{:}\theta_1, \ldots, l_n \Rightarrow x_n{:}\theta_n\}.(f\ \{l_1 \Rightarrow x_1, \ldots, l_n \Rightarrow x_n\} + a)$.

   This sum of three terms expresses that N may be reduced after substitution in M, or before, and that there may be one step of β-reduction.

In this function we make two approximations. The first one is that we count one step for each application, should there be an abstraction to reduce with or not. The second one is that we take the sum of the call-by-name and call-by-value strategies, and not their maximum. Since these are only over-estimations, our function gives an upper-bound of the longest reduction path.

If $\Gamma \vdash M : \{l_1 \Rightarrow \theta_1, \ldots, l_n \Rightarrow \theta_n\} \rightarrow \tau$ and $\Gamma|_{FV(M)} = \{x_1 \mapsto \tau_1, \ldots, x_m \mapsto \tau_m\}$, then $T_\Gamma(M)$ is a total function from $\theta_1^* \times \cdots \times \theta_n^* \times \tau_1^* \times \cdots \times \tau_m^*$ to int. This means that on any complete input coherent with its typing, M will terminate. As an independent term, we have an upper bound of its longest reduction path by $rect(int, T_\Gamma(M) : (r \rightarrow \tau)^*)$. □

4.2 Polymorphic selective λ-calculus

Whereas more powerful typing systems exist, like second order λ-calculus, the ML trend for typing is the more simple (because more restrictive) let-polymorphism. Type quantification appears outside of the type itself, and instantiation is done implicitly while applying to arguments. The principal advantage of this type system is that, for λ-calculus, any term has a most generic type, which avoids explicit declarations of type, since a simple unification algorithm gives this type.

We will show here that such an algorithm exists for selective λ-calculus too. This means that, from a typing point of view, the addition of labels is coherent with polymorphically typed λ-calculus.

$$\Gamma[x \mapsto \sigma] \vdash x : \sigma \tag{I}$$

$$\frac{\Gamma[x \mapsto \theta] \vdash M : r \to \tau}{\Gamma \vdash \lambda\{l_1 \Rightarrow x_1, \ldots\}.M : \{l_1 \Rightarrow \theta_1, \ldots\} \cdot r \to \tau} \tag{II}$$

$$\frac{\Gamma \vdash M : \{l_1 \Rightarrow \theta_1, \ldots\} \cdot r \to \tau \quad \Gamma \vdash N_i : \theta_i}{\Gamma \vdash M\ \{l_1 \Rightarrow N_1, \ldots\} : \tau} \tag{III}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma} \quad \alpha \text{ not free in } \Gamma \tag{IV}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha \backslash \tau]} \tag{V}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma[x \mapsto \sigma] \vdash N : \tau}{\Gamma \vdash \text{let } x = M \text{ in } N : \tau} \tag{VI}$$

Figure 4.2: Typing rules for polymorphic selective λ-calculus

4.2.1 Syntax and types

The syntax is that of untyped selective λ-calculus with a *let* construct to introduce polymorphism, types being provided by inference.

$$M ::= x \mid \lambda\{l \Rightarrow x, \ldots\}.M \mid M\ \{l \Rightarrow M, \ldots\} \mid \text{let } x = M \text{ in } M'$$

We add a reduction rule, corresponding to the new construct:

$$\text{let } x = M \text{ in } N \to N[x \backslash M].$$

Like in Damas and Milner's definition [DM82] types are divided into monotypes ranged by t, and polytypes ranged by σ.

w	::=	$u \mid v$	return types
t	::=	$\{l \Rightarrow t, \ldots\} \to w$	monotypes
σ	::=	$t \mid \forall v.\sigma$	polytypes

where u stands for base types and v for type variables.

The distinction we introduce here between return types and monotypes is specific to selective λ-calculus. Since we want types to be as flat as possible, return types should not be functional. This means that when we substitute a variable that appears as return type with a functional type, we will need to modify the structure of the type.

4.2.2 Typing rules

The typing rules are given in figure 4.2. The rules (IV)-(VI) are in no way specific to selective λ-calculus. Since type quantifiers are external they are independent from the structure of monotypes, they are exactly the same as for classical λ-calculus. Their roles are IV: Generalize, V: Instantiate and VI: Let introduction.

Proposition 4.2 (subject reduction) *If $\Gamma \vdash M : \tau$ in polymorphically typed selective λ-calculus, and $M \to N$, then $\Gamma \vdash N : \tau$.*

PROOF Since polymorphism can only be used in conjunction with **let**, the proof for simple types is enough except for let-reduction.

In this last case, the derivation tree starts with:

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma[x \mapsto \sigma] \vdash N : \tau}{\Gamma \vdash \text{let } x = M \text{ in } N : \tau}$$

We first make all α-conversions necessary to the substitution of x by M. After reduction we obtain a tree with root $\Gamma \vdash N[x \backslash M] : \tau$ from the derivation tree of $\Gamma[x \mapsto \sigma] \vdash N : \tau$ by replacing every occurrence of the axiom $\Gamma'[x \mapsto \sigma] \vdash x : \sigma$ by the derivation tree of $\Gamma' \vdash M : \sigma$ ($\forall y \in FV(M)\ \Gamma(y) = \Gamma'(y)$). □

Still there is an important difference between these rules and rules for classical λ-calculus. It is hidden in the $\sigma[\alpha \backslash \tau]$ of rule (V). The substitution rule for types is

$$(r \to \alpha)[\alpha \backslash (r' \to \omega)] = (r[\alpha \backslash (r' \to \omega)] \cdot r') \to \omega\,.$$

This means that our domain of types is radically different of Herbrand, where unification is usually described. Knowing that, the existence of a type inference algorithm may be surprising. It is due to some good properties of our system, particularly that type normalization (the flattening of $r \to r' \to \omega$ in $r \cdot r' \to \omega$) does not change sizes of types.

Since in a well-typed term all type variables get fixed once for all, we have strong normalization.

Theorem 4.2 *Polymorphic selective λ-calculus is strongly normalizing.*

PROOF We find an upper bound of the longest evaluation of M by that of $\tilde{M}$, which is M where all occurrences of let are suppressed by transforming let $x = P$ in N into $K\ \{1 \Rightarrow N[x \backslash P], 2 \Rightarrow P\}$, where $K = \lambda\{1 \Rightarrow x, 2 \Rightarrow y\}.x$. We need K for the case x does not appear in N. Since the result is monomorphic everywhere, the argument for the simply typed calculus holds. □

4.2.3 Type unification

The base of a type synthesis algorithm is unification. We give here a unification algorithm for monotypes defined above.

The reason we have to design a new algorithm is that we work modulo type normalization. That is, we have a good form of E-unification [GS89b], where the equivalence relation on terms can be expressed by an oriented rewriting system.

Theorem 4.3 *There is an algorithm which gives the most generic unifier of a set of equations on monotypes or reports failure if there is none.*

PROOF We can write unification as a rewriting algorithm which normalizes a conjunction of equalities. The rules are given in figure 4.3. ϕ represents a conjunction of equalities, conjunction and equality are commutative and associative. Notations are α to match variables, τ or θ to match any type, ω to match return types. In return types we have abbreviated the $\{\} \to$ part.

This rewriting system has a strongly normalizing strategy, up to equalized or new variables substitution, and the resulting normal form is the most general unifier of the original set of equations.

We first prove the correctness of this rewriting system. That is, for each rewriting rule, any unifier of the result unifies the origin, and reciprocally any unifier of the origin can be extended in an unifier of the result, by defining it on variables absent in the origin.

Base type $\dfrac{\phi \wedge u = v}{\bot}(u \neq v;\ u, v \text{ base types})$ Redundancy $\dfrac{\phi \wedge \theta = \theta}{\phi}$

Non-recurrent $\dfrac{\phi \wedge \alpha = \tau}{\bot}(\tau \neq \alpha, \alpha \in Var(\tau))$

Type structure $\dfrac{\phi \wedge u = \{l_1 \Rightarrow \tau_1, ...\} \rightarrow \omega}{\bot}(u \text{ base type})$

Elimination $\dfrac{\phi \wedge \alpha = \tau}{\phi[\alpha \backslash \tau] \wedge \alpha = \tau}(\alpha \in Var(\phi) \setminus Var(\tau), \text{if } \tau \text{ variable then } \tau \in Var(\phi))$

Decomposition $\dfrac{\phi \wedge \{l \Rightarrow \theta\} \cdot r \rightarrow \omega = \{l \Rightarrow \theta'\} \cdot r' \rightarrow \omega'}{\phi \wedge \theta = \theta' \wedge r \rightarrow \omega = r' \rightarrow \omega'}$

Completion $\dfrac{\phi \wedge \{l \Rightarrow \theta\} \cdot r \rightarrow \omega = r' \rightarrow \omega'}{\phi \wedge \{l \Rightarrow \theta\} \cdot r \rightarrow \omega = \{l \Rightarrow \theta\} \uplus r' \rightarrow \alpha \wedge \omega' = \psi_{r'}\{l \Rightarrow \theta\} \rightarrow \alpha}$ $r' \neq \{\}$, $l \notin \mathcal{D}_{r'}$, α fresh

Figure 4.3: Rewriting rules for type unification

Base type, *Non-recurrent*, and *Type structure* detect inconsistencies in the equations. That is, equation between two different base types, between a type variable and a type containing it, or between a base type and a functional type. When one of these rules fires, the system has no unifier.

Redundancy suppresses meaningless equations. It leaves unchanged the set of unifiers.

Elimination substitutes variables (using type normalization), while keeping their referent. If σ unifies the upper side, then $\sigma(\alpha) = \sigma(\tau)$, and it unifies the lower side. And reciprocally.

Decomposition takes a label already present on the two sides of an equation, and equates the types. Correctness is clear.

Completion is used in case a label appears only on one side of an equation. We use here the second concatenation equality, to introduce it in the other side. Any unifier of the result unifies the origin, since $\{l \Rightarrow \theta\} \uplus r' \rightarrow \alpha = r' \cdot \psi_{r'}\{l \Rightarrow \theta\} \rightarrow \alpha$, which is by unification equal to $r' \rightarrow \omega'$. Reciprocally, if σ is a unifier of the upper side, then it maps ω' to a functional type of the form $\psi_{r'}\{l \Rightarrow \sigma(\theta)\} \cdot r'' \rightarrow \omega''$ and we extend it for the lower side by adding $\sigma(\alpha) = r'' \rightarrow \omega''$.

Next we prove that there is a terminating strategy for this system. A variable is *solved* when it appears only once, and as a side of an equation. We prove that a strategy reduces the lexicographical measure *(unsolved variables,sum of sizes)*, where the size of a type is the total number of labels, variable occurrences and base types it contains.

Failure rules terminate, Redundancy and Decomposition reduce the sum of sizes, and Elimination the number of unsolved variables.

Completion by itself does not reduce the measure. But if we use it in combination with Decomposition on the same equation, eliminating or failing as soon as possible, we finally reduce the number of unsolved variables. If ω' was not a variable, we fail immediately. Otherwise, it is solved, but we create a new variable α. We repeat this until we can solve a "successor" of α with the left hand side (which may suppose creating a lineage to ω too, if completion is mutual). This sequence terminates, since there is only a finite number of labels on each side.

$$
\begin{array}{rrl}
Tp(\Gamma) = & x \mapsto & \text{match } \Gamma(x) \text{ with} \\
 & & \forall(fv).\tau \rightarrow (\top, NV(fv,\tau)) \\
| & \lambda\{l_1 \Rightarrow x_1, \ldots\}.M \mapsto & \text{let } (\phi,\tau) = Tp(\Gamma[x_1 \mapsto \alpha_1, \ldots], M) \\
 & & \text{in } (\phi, (\{l_1 \Rightarrow \alpha_1, \ldots\} \rightarrow \beta)[\beta \backslash \tau]) \\
| & M\ \{l_1 \Rightarrow N_1, \ldots\} \mapsto & \text{let } (\phi,\tau) = Tp(\Gamma, M) \text{ in} \\
 & & \text{let } (\phi_i,\theta_i) = Tp(\Gamma, N_i) \text{ in} \\
 & & (\phi \wedge \phi_1 \wedge \ldots \wedge (\tau = \{l_1 \Rightarrow \theta_1, \ldots\} \rightarrow \alpha), \alpha) \\
| & \text{let } x = M \text{ in } M' \mapsto & \text{let } (\phi,\theta) = Tp(\Gamma, M) \text{ in} \\
 & & \text{let } \theta' = mgu_\phi(\theta) \text{ in} \\
 & & \text{let } fv = FV(\theta') \setminus FV(mgu_\phi(\Gamma)) \text{ in} \\
 & & \text{let } (\phi',\tau) = Tp(\Gamma[x \mapsto \forall(fv).\theta'], M') \\
 & & \text{in } (\phi \wedge \phi', \tau)
\end{array}
$$

Figure 4.4: Type inference algorithm

Last, we must show that our result can be interpreted as a substitution. First, in every equation, at least one side is a solved variable. If the two sides are functional types, then either Decomposition or Completion applies. If one side is a base type, then the other side is a solved variable, otherwise Elimination, Redundancy or some failure applies. If the two sides are variables, then at least one is solved.

We construct the substitution σ by taking for each equation $\alpha = \tau$, α solved, $\sigma(\alpha) = \tau$. σ is a most general unifier of the final system, and, as a consequence, if we suppress definitions for all variables introduced by completion, σ' is a most general unifier of the original system. □

4.2.4 Type inference

Once we have type unification, type inference becomes a very simple thing. We just add new equations to the list while moving through the term, as shown in figure 4.4.

The principal function of this algorithm, Tp, takes a typing environment Γ (bindings from variable names to types) and a selective λ-term, to give back a couple (equation list,type of the term).

Γ is an association list, and $\Gamma(x)$ is the polytype associated to x. We have flattened the structure by writing $\forall(\alpha,\beta,...).\tau$ for $\forall\alpha.\forall\beta.....\tau$. NV is a function that renames variables listed in fv with fresh names in τ. $FV(\tau)$ lists free variables in τ, and by extension we write $FV(\Gamma)$ for free variables in associated types in Γ. mgu_ϕ is the most general unifier of ϕ; by extension we apply it to environments too. "\" is set subtraction.

This algorithm constructs a derivation tree whose root is $\Gamma \vdash M : \tau$, where Γ and M are given. Since there is only one way to construct this tree, by induction on the structure of M, except for generalization and instantiation which are handled in the most general way in the variable and let cases, and that we add only necessary equations, this algorithm is complete and correct. To get the real derivation tree we apply finally mgu_ϕ to the scheme obtained.

4.3 Application to programming

We give here examples of the use of such extension in an ML-like language[1], together with inferred types.

4.3.1 Keywords: an enhancement for clarity

We start by giving some examples of how the use of keywords, and their appearance in types may help the programmer. Our view is already partially proved by the use of records as data structures: while theoretically everything could be done with tuples, one will often prefer to use a record, which explicits what is represented.

First, here are some examples of functions written in an ML-like syntax, with their inferred types.

```
#let cons car=>a cdr=>b = a::b;;
 cons : {car=>'a,cdr=>'a list} -> 'a list

#cons cdr=>[1];;
 it : {car=>int} -> int list

#let rec map f=>f = fun [] -> []
#      | [h|t] -> (f h)::map f=>f t;;
 map : {1=>'a list,f=>{1=>'a} -> 'b} -> 'b list

#map f=>(add 1);;
 it : {1=>int list} -> int list

#map [1;2;3];;
 {f=>{1=>int} -> 'a} -> 'a list
```

The advantage of this system is double. First, it is more expressive, and second, partial application can be done on any label.

One could argue that in the functions above, order is clear enough so that, even without labels, there is no possibility of error. However this becomes less systematic for functions of three arguments or more, and is not so natural in some two-argument functions. For instance, think about `mem` (membership) or `assoc` (association list), whose respective types are:

```
value mem : 'a -> 'a list -> bool
value assoc : 'a -> ('a * 'b) list -> 'b
```

There is no special reason for them to respect such an order. The opposite could even be more natural, since we will more often map them on the first argument than on the second. Here a quick glance at the type suppresses the ambiguity. But this is not always true, and even if we can suppose the programmer to be able to do that, the following types would certainly be more practical.

```
value mem : {1=>'a,in=>'a list} -> bool
value assoc : {1=>'a,in=>('a * 'b) list} -> 'b
```

[1] We use a notation close to CAML [W+90]. "`let`" denotes a definition, "`::`" the list constructor. Since "`=>`" is left unused (abstraction uses "`->`"), we use it for labeling.

This is not only more readable. If one knows that every time we fetch something in a list we use the label "in", there is no longer any ambiguity.

With two arguments, there were only two possibilities of order. If we have three, we jump to six. Since the number of combinations is $n!$, remembering arguments order for functions of more than three arguments, and there are lots of them in the functional programming paradigm, is more than uneasy. We give a little more examples. Take it_list and list_it (fold left and right),

```
value it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
value list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

A natural labeling as

```
value it_list : {1=>'a list,f=>{1=>'b,2=>'a} -> 'b,start=>'b} -> 'b
value list_it : {1=>'a list,f=>{1=>'a,2=>'b} -> 'b,start=>'b} -> 'b
```

would be expressive enough, and avoid my trying to understand the type every time I use one of them.

We have voluntarily limited ourselves here to generic functions, for which currying is useful. If we think of functions interfacing a window manager for instance, the number of arguments per function is such that the use of labels seems a necessity, but one could do with records, since currying is not so important. Nonetheless, the trend in functional languages is towards a systematic use of currying. Standard ML is a notable exception, preferring uncurried functions, but CAML is an example of ML dialect preferring currying.

4.3.2 Relative positions versus combinators

If the advantage of symbolic labels was in expressiveness, that of relative positions is in conciseness.

```
#let sub x y = x-y;;
 sub : {1=>int,2=>int} -> int

#let minus15 = sub 2=>15;;
 minus15 : {1=>int} -> int

#let cons a b = a::b;;
 cons : {1=>'a,2=>'a list} -> 'a list

#map f=>(cons 2=>[1;2]);;
 it : {1=>int list} -> int list list

#map f=>(sub 2=>10) [11;12;13];;
 it : int list = [1;2;3]
```

We can obtain the same result with the combinator C (back into CAML).

```
#let C f x y = f y x;;
 C : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c

#map (C sub 10) [11;12;13];;
 it : int list = [1;2;3]
```

But the result is hardly readable, and we need a combinator for each position.

Chapter 5

Semantics

We gave already two accounts of the label-selective λ-calculus: a purely syntactic one (*cf.* Chap. 2 and 3) and a type theoretic one (*cf.* Chap. 4). This gives us a good knowledge of how it works as a system. The next question is: What does it really represent? The trivial answer, *selective functions* —functions getting their arguments through labels—, gives only a partial account of what happens. We will start by formalizing it in a simply typed framework, but will go on studying what are really doing entities, first in a domain theoretical way, and then in a category theoretical way.

5.1 Model of the selective λ-calculus

We give here a definition, very similar to that of λ-model.

Definition 5.1 (selective λ-model) *A model of selective λ-calculus is a triple $\langle \mathcal{A}, (\widehat{p})_{p\in\mathcal{L}}, [\![\cdot]\!] \rangle$ where each $\widehat{p}$ is a binary operator of $\mathcal{A}$ and the translation*

$$(M,\rho) \mapsto [\![M]\!]_\rho : \Lambda \times P_{\mathcal{V}\to\mathcal{A}} \to \mathcal{A}$$

satisfies the following properties,

$$[\![x]\!]_\rho = \rho(x) \tag{i}$$

$$[\![M\ \{p \Rightarrow N\}]\!]_\rho = [\![M]\!]_\rho \,\widehat{p}\, [\![N]\!]_\rho \tag{ii}$$

$$[\![\lambda\{p \Rightarrow x\}.M]\!]\,\widehat{p}\, a = [\![M]\!]_{\rho[x\mapsto a]} \qquad (\forall a \in \mathcal{A}) \tag{iii}$$

$$[\![M]\!]_\rho = [\![M]\!]_{\rho|_{FV(M)}} \tag{iv}$$

$$(M \equiv N) \Rightarrow [\![M]\!]_\rho = [\![N]\!]_\rho \tag{v}$$

$$(\forall a \in \mathcal{A} [\![M]\!]_{\rho[x\mapsto a}} = [\![N]\!]_{\rho[x\mapsto a]}) \Rightarrow [\![\lambda\{p \Rightarrow x\}.M]\!]_\rho = [\![\lambda\{p \Rightarrow x\}.N]\!]_\rho \tag{vi}$$

It is extensional if

$$\forall a,b \in \mathcal{A}, \forall p \in \mathcal{L},\ (\forall x \in \mathcal{A}, a\,\widehat{p}\,x = b\,\widehat{p}\,x) \Rightarrow a = b. \tag{vii}$$

We will not give a model of this type. It is certainly feasible, and Scott models, for instance, should be easily adaptable. Nonetheless, the complexity introduced by labels suggests other ways, where label's meaning would be more evident.

Proposition 5.1 *In any non trivial model of the selective λ-calculus, $[\![_]\!]$ restricted to equivalence classes of normalizable closed terms by $=_{\beta\eta}$ is injective.*

PROOF By the second separation theorem we have an environment of applications P such that $(\{p \Rightarrow x, q \Rightarrow y\}.P{:}M) \downarrow = x$ and $(\{p \Rightarrow x, q \Rightarrow y\}.P{:}N) \downarrow = y$.

Since our model is not trivial we have a and b such that $a \neq b$. $[P](d)$ being a notation for the translation of a sequence of applications on d as operations in the model, we have

$$\begin{array}{lcl} ([P]([\![M]\!]))\,\widehat{p}\,a\,\widehat{q}\,b & = & a, \\ ([P]([\![N]\!]))\,\widehat{p}\,a\,\widehat{q}\,b & = & b. \end{array}$$

5.2 Typed model

These are models for the simply typed selective λ-calculus, presented in Chapter 4.

5.2.1 Definition

Definition 5.2 (typed model) *A typed model of selective λ-calculus on $\mathcal{L}$ is a quadruple $\langle \mathcal{A}, T, (\widehat{p})_{p\in\mathcal{L}}, [\![_]\!]_ \rangle$ where $T : \mathcal{A} \to \mathcal{T}$, $\mathcal{T}$ the set of types, is a total function, each $\widehat{p}$ is an internal binary operator of $\mathcal{A}$ which respects abstract application rules, and the translation*

$$(M, \rho) \mapsto [\![M]\!]_\rho : \Lambda_T \times P_{\mathcal{V}\to\mathcal{A}} \to \mathcal{A}.$$

has the following properties, for all M, N in Λ_T, p in $\mathcal{L}$, x in $\mathcal{V}$, a in $\mathcal{A}$,

$$(T \circ \rho \vdash M : \tau) \Rightarrow T([\![M]\!]_\rho) = \tau, \tag{i}$$

$$[\![x]\!]_{\rho[x\mapsto a]} = a, \tag{ii}$$

$$(M \equiv N \wedge T \circ \rho \vdash M : \tau) \Rightarrow [\![M]\!]_\rho = [\![N]\!]_\rho, \tag{iii}$$

$$(T \circ \rho \vdash M : \{l \Rightarrow \theta\} \cdot r \to u \wedge T \circ \rho \vdash N : \theta) \Rightarrow [\![M\ \{l \Rightarrow N\}]\!]_\rho = [\![M]\!]_\rho\, \widehat{\imath}\, [\![N]\!]_\rho, \tag{iv}$$

$$(T \circ \rho \vdash M : \tau) \Rightarrow [\![M]\!]_{\rho,\tau} = [\![M]\!]_{\rho|_{FV(M)}}, \tag{v}$$

$$(\forall a \in \mathcal{A}^\theta [\![M]\!]_{\rho[x\mapsto a]} = [\![N]\!]_{\rho[x\mapsto a]}) \Rightarrow [\![\lambda\{l \Rightarrow x\}.M]\!]_\rho = [\![\lambda\{l \Rightarrow x\}.N]\!]_\rho \tag{vi}$$

This model is moreover extensional if

$$\forall a, b \in \mathcal{A}^{\{l\Rightarrow\theta\}\cdot r\to u}, \forall p \in \mathcal{L},\ (\forall x \in \mathcal{A}^\theta, a\,\widehat{p}\,x = b\,\widehat{p}\,x) \Rightarrow a = b. \tag{vii}$$

5.2.2 Construction of a model

For any u in $\mathcal{T}_0$ (the set of base types), let $\mathcal{A}^u$ be the set of all objects of type u, ($T(\mathcal{A}^u) = \{u\}$). They are represented by a pair $(b, ())$, where b is the actual value, and $()$ expresses that they are basic objects. $\mathcal{A}_0 = \bigcup_{u\in\mathcal{T}_0} \mathcal{A}^u$ is the set of constant symbols of our domain. We demand that each object have a definite type: $u \neq u' \Rightarrow \mathcal{A}^u \cap \mathcal{A}^{u'} = \emptyset$, and $T(\mathcal{A}_0) \subset \mathcal{T}_0$.

We define the domain of labeled functions constructed on $\mathcal{A}_i$ as all the functions for which we can give a simple typing. That is the set

$$F_\omega(\mathcal{A}_i) = \bigcup_{u\in\mathcal{T}_0} \bigcup_{(\tau_i)\in T_i^*} \mathcal{A}^{\tau_1} \times \ldots \times \mathcal{A}^{\tau_n} \to \mathcal{A}^u$$

To obtain a typing as defined above, we must affect labels to each argument. This is done by ordered sets of $\mathcal{L}$. $\mathcal{P}_n(\mathcal{L})$ is the set of n-uples of strictly growing labels.

We can then define $\mathcal{A}_{i+1}$ as a product of $F_\omega(\mathcal{A}_i)$ by $\mathcal{P}(\mathcal{L})$. For this we slice the two sets according to the number of arguments and get

$$\mathcal{A}_{i+1} = \bigcup_{n\in\mathbb{N}} F_n(\mathcal{A}_i) \times cP_n(\mathcal{L}).$$

Our domain is finally:

$$\mathcal{A} = \bigcup_{n \in \mathbb{N}} \mathcal{A}_n$$

For each label l in $\mathcal{L}$ we define an operator $\widehat{l}$

$$\begin{aligned}\forall (f,(l_i)) \in \mathcal{A}^{\{l \Rightarrow \theta\} \cdot r \to u}, \forall a \in \mathcal{A}^{\theta},\ (\exists j, l_j = l) \Rightarrow f \widehat{l}\, a = \\ ((x_1, \cdots, x_{j-1}, x_{j+1}, \ldots, x_n) \mapsto f(x_1, \cdots, x_{j-1}, a, x_{j+1}, \ldots, x_n), \\ (l_1, \cdots, l_{j-1}, \psi_{\{l_j\}}(l_{j+1}), \ldots, \psi_{\{l_j\}}(l_n))) \in \mathcal{A}^{r \to u}\end{aligned}$$

5.2.3 Correctness of this model

We define a translation from selective λ-calculus expressions to our domain by

$$(M, \rho) \mapsto [\![M]\!]_{\rho} :\ \Lambda_T \times P_{\mathcal{V} \to \mathcal{A}} \to \mathcal{A}.$$

which gives for variables, ($x \in \mathcal{V}$),

$$\forall a \in \mathcal{A}\ [\![x]\!]_{\rho[x \mapsto a]} = a$$

and for abstractions, when $T \circ \rho \vdash \lambda\{l \Rightarrow x{:}\theta\}.M : \{l_i \Rightarrow \tau_i\}_{i=1}^{n} \to u$, l_i's are growing, and $l = l_j$

$$\begin{aligned}[\![\lambda\{l \Rightarrow x{:}\theta\}.M]\!]_{\rho} = \\ ((x_1, \ldots, x_n) \mapsto (fst([\![M]\!]_{\rho[x \mapsto x_j]}))(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n), (l_1, \ldots, l_n))\end{aligned}$$

and finally for applications, when $T \circ \rho \vdash M\ \{l \Rightarrow N\} : \tau$,

$$[\![M\ \{l \Rightarrow N\}]\!]_{\rho} = [\![M]\!]_{\rho}\, \widehat{l}\, [\![N]\!]_{\rho}.$$

Proposition 5.2 *$\langle \mathcal{A}, T, (\widehat{l})_{l \in \mathcal{L}}, [\![_]\!]_{_} \rangle$ is an extensive typed model of selective λ-calculus.*

PROOF Properties of $[\![\ ]\!]$ are verified.

i. We translate only typable expressions, and respect types.

ii. Idem rule for variables.

iii. Induction on the structure of terms. The point is ordering rules.

iv. The translation rule itself.

v. We don't use variables not free in M during the translation.

vi. Equality on functions is extensional.

vii. Idem.

□

5.3 Multi-application model

We have obtained, in the previous section, quite easily a model for the simply-typed selective λ-calculus. It gives a good abstraction for terms with same normal form, but it leaves the notion of commutativity of the calculus quite unclear.

To express clearly this property we need to define a monoid of parameters. This is exactly that of streams[1]: we take $\mathcal{P} = \mathcal{S}(\mathcal{L}, \mathcal{A})$.

With this we can define the multi-application $M : \mathcal{A} \times \mathcal{P} \to \mathcal{A}$ to be:

$$\forall f \in \mathcal{A}, \forall \{l_i \Rightarrow a_i\}_{i=1}^n \in \mathcal{P},\ (\forall i, l_i < l_{i+1}) \Rightarrow M(f,a) = f\ \widehat{l_n}\ a_n \dots \widehat{l_1}\ a_1$$

This multi-application is used by the new translation rule:

$$[\![M\ \{l_i \Rightarrow N_i\}_{i=1}^n]\!]_\rho = M([\![M]\!]_\rho, \{l_i \Rightarrow [\![N_i]\!]_\rho\}_{i=1}^n)$$

Definition 5.3 (multi-application model) *A multi-application model of the selective λ-calculus as is a quadruple $\langle \mathcal{A}, \mathcal{P}, M, [\![\ ]\!] \rangle$ such that, for some $\Rightarrow : \mathcal{L} \times \mathcal{A} \to \mathcal{P}$, when we define $\widehat{p}$ as $(x,y) \mapsto M(x, p \Rightarrow y)$, $\langle \mathcal{A}, \{\widehat{p}\}, [\![\ ]\!] \rangle$ is a model of the selective λ-calculus, $\mathcal{P}$ is a monoid, and $r \mapsto M(_, r)$ is an injective anti-homomorphism from $\langle \mathcal{P}, \cdot_{\mathcal{P}} \rangle$ to $\langle \mathcal{A} \to \mathcal{A}, \circ \rangle$.*

We similarly define a typed model.

This form of model may even appear more natural, since we use now the stream notation, but they depart a little from classical λ-models.

Proposition 5.3 *$\langle \mathcal{A}, T, \mathcal{P}, M, [\![\ ]\!] \rangle$ is a typed multi-application model of selective λ-calculus.*

PROOF We added the required M and $\Rightarrow$. □

5.4 Abstraction models

After multi-application models, it is natural to extend modeling to abstraction, seen as similar to application, and put the accent on a larger argument monoid $\mathcal{P}$. For simplicity we give here only untyped definitions. Typing may be added in the classical way.

5.4.1 Definition

Definition 5.4 (abstraction model) *An abstraction model of selective λ-calculus is a quintuple $\langle \mathcal{A}, \mathcal{P}, H, [\![\ ]\!]^\Lambda, [\![\ ]\!]^\Gamma \rangle$[2] such that*

- *$\mathcal{P}$ is a monoid,*
- *$[\![\ ]\!]^\Lambda$ and $[\![\ ]\!]^\Gamma$ are applications respectively $\Lambda \to \mathcal{A}$ and $\Gamma^* \to \mathcal{P}$,*
- *when M is convertible to N in selective λ-calculus, images of M and N are equal:*

$$M \to N \vee M \equiv N \Rightarrow [\![M]\!]^\Lambda = [\![N]\!]^\Lambda,$$

- *H is an injective anti-homomorphism from $\langle \mathcal{P}, \cdot_{\mathcal{P}} \rangle$ to $\langle \mathcal{A} \to \mathcal{A}, \circ \rangle$, (we may note $h{:}a$ for $H(h)(a)$,)*

[1] In fact the original idea of streams comes from this need of a parameter monoid in the model.

[2] Λ and Γ^* are respectively the sets of selective λ-terms and entity sequences

- *we have canonical injections* $/ : \mathcal{S}(\mathcal{A}) \to \mathcal{P}$ *(antimorphism) and* $\lambda : \mathcal{S}(\mathcal{V}) \to \mathcal{P}$ *(morphism), both into* $\mathcal{P}$*, called application and abstraction, which have the properties:*

$$\begin{array}{lcl} /\{l \Rightarrow [\![M]\!]^\Lambda\} & = & [\![\{l \Rightarrow M\}]\!]^\Gamma \\ \lambda\{l \Rightarrow x\} & = & [\![\lambda\{l \Rightarrow x\}]\!]^\Gamma \\ /\{l \Rightarrow [\![N]\!]^\Lambda\} : [\![M]\!]^\Lambda & = & [\![M\ \{l \Rightarrow N\}]\!]^\Lambda \\ \lambda\{l \Rightarrow x\} : [\![M]\!]^\Lambda & = & [\![\lambda\{l \Rightarrow x\}.M]\!]^\Lambda \end{array}$$

5.4.2 Properties

We have introduced variables in the model, and as a consequence $\mathcal{A}$ will "contain" at least all $\beta\eta$-normal forms in Λ.

Proposition 5.4 $[\![\,]\!]^\Lambda$ *restricted to* $\beta\eta$*-normal forms is injective for any non trivial abstraction model.*

PROOF Since our model is not trivial, we have two distinct terms a and b in $\mathcal{A}$.

For two different $\beta\eta$-normal terms M and N, we use the separation theorem to get a context P such that $(P{:}M) \downarrow = x$ and $(P{:}N) \downarrow = y$. We suppose we have to different labels p and q. We abstract the variables by $\lambda\{p \Rightarrow x, q \Rightarrow y\}$. We then have:

$$\begin{array}{lcl} /\{p \Rightarrow a, q \Rightarrow b\}.\lambda\{p \Rightarrow x, q \Rightarrow y\}.[\![P]\!]^\Gamma{:}[\![M]\!]^\Lambda & = & a, \\ /\{p \Rightarrow a, q \Rightarrow b\}.\lambda\{p \Rightarrow x, q \Rightarrow y\}.[\![P]\!]^\Gamma{:}[\![N]\!]^\Lambda & = & b, \end{array}$$

which proves the difference. □

Now, we can see what we can do with entity concatenation. In fact we will read everything from left to right and see the meaning of some canonical combinations. Here we interpret labels as channels and variables as broadcasts.

- $\{p \Rightarrow M\}$: put M on channel p. We send the result of the calculation of M to the first user of p
- $\{p \Rightarrow x\}$: save x on channel p. By moving a variable to a channel we protect it from hazardous redefinitions.
- $\lambda\{p \Rightarrow x\}$: get x from channel p.
- $\{p \Rightarrow M\}.\lambda\{p \Rightarrow x\}$: assign M to x.
- c : substitute x with $M(x)$ on p. Side-effect on x. If $x \notin FV(M)$, this is just forgetful assignment on labels.
- $\lambda\{p \Rightarrow x\}.\{q \Rightarrow M(x)\}$: write the result of $M(p)$ on channel q. Side-effect on x.

We see here that we can translate any imperative program into an environment, and then we need not care about order of execution.

Of course there are these disturbing side-effects, that make composition difficult to verify. We have to restrict this model.

5.4.3 Restricted abstraction model

A restricted abstraction model is just an abstraction model restricted to closed terms and environments.

Definition 5.5 (restricted abstraction model) *A* restricted abstraction model *of selective λ-calculus is a quintuple $\langle \mathcal{A}, \mathcal{P}, H, [\![\,]\!]^{\Lambda}_{\cdot}, [\![\,]\!]^{\Gamma}_{\cdot} \rangle$ such that*

- *$\mathcal{P}$ is a monoid,*
- *$[\![\,]\!]^{\Lambda}$ and $[\![\,]\!]^{\Gamma}$ are applications respectively $\Lambda \times P_{\mathcal{V}\to\mathcal{A}} \to \mathcal{A}$ and $\Gamma^* \times P_{\mathcal{V}\to\mathcal{A}} \to \mathcal{P}$,*
- *when M is convertible to N in selective λ-calculus, images of M and N are equal:*

$$M \to N \vee M \equiv N \Rightarrow [\![M]\!]^{\Lambda}_{\rho} = [\![N]\!]^{\Lambda}_{\rho},$$

- *H is an injective anti-homomorphism from $\langle \mathcal{P}, \cdot_{\mathcal{P}} \rangle$ to $\langle \mathcal{A} \to \mathcal{A}, \circ \rangle$, (we may note $h{:}a$ for $H(h)(a)$,)*
- *we have a family of mappings $\{[(p_m) \to (q_n)] : \mathcal{A}^n \to \mathcal{P} \mid (p_m) \in \mathcal{L}^m, (q_n) \in \mathcal{L}^n\}$, called transformations, which has the properties:*

$$\begin{array}{l}
[(p_m) \to (q_n)]([\![M_1]\!]^{\Lambda}_{\rho}, ..., [\![M_n]\!]^{\Lambda}_{\rho}) = \\
\qquad [\![\lambda \{p_i \Rightarrow x_i\}_{i=1}^{m}.\{q_1 \Rightarrow M_1\ \{p_i \Rightarrow x_i\}_{i=1}^{m}, \ldots, q_n \Rightarrow M_n\ \{p_i \Rightarrow x_i\}_{i=1}^{m}\}]\!]^{\Gamma} \\
[(p_m) \to (q_n)]([\![M_1]\!]^{\Lambda}_{\rho}, ..., [\![M_n]\!]^{\Lambda}_{\rho}){:}[\![M_0]\!]^{\Lambda}_{\rho} = \\
\qquad [\![\lambda \{p_i \Rightarrow x_i\}_{i=1}^{m}.\{q_1 \Rightarrow M_1\ \{p_i \Rightarrow x_i\}_{i=1}^{m}, \ldots, q_n \Rightarrow M_n\ \{p_i \Rightarrow x_i\}_{i=1}^{m}\}.M_0]\!]^{\Gamma}
\end{array}$$

when $\{x_i\} \cap \bigcup FV(M_i) = \emptyset$.

Proposition 5.5 *If in an abstraction model we take only* closed elements *(that do not depend on variables), we have a restricted abstraction model.*

What do represent restricted abstraction models? The selective λ-calculus extended with composition, but where we still distinguish fonctions in $\mathcal{A}$ and transformations in $\mathcal{P}$. When we lose this distinction, we get the transformation calculus of Chapter 6, and the model we give for it (*cf.* 7.2) is a typed restricted abstraction model, where $\mathcal{P}$ is included in $\mathcal{A}$.

5.5 Categorical models

5.5.1 Label Category

Definition 5.6 (Label monoid) *A label monoid $\mathcal{M} = \{r \subset \mathcal{L} \mid |r| < \infty\}$ is defined by a set $\mathcal{L}$ of* labels, *together a shifting function $\phi : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ such that:*

- *for any r in $\mathcal{M}$, $\phi(r, _)$ is a bijection from $\mathcal{M}$ to $\{s \in \mathcal{M} \mid r \cap s = \emptyset\}$,*
- *$r \cdot s = r \cup \phi_r(s)$ defines an associative operation,*
- *$\emptyset$ is the neutral element of $\mathcal{M}$.*

Properties The following are consequences of the definition:

1. For any r in $\mathcal{M}$, $\emptyset$ is a fixpoint of ϕ_r. (bijection + definition of $\cdot$)
2. $\phi_r(s) = s$ implies $\phi_s(r) = r$. ($r \cdot s = r \cup s = s \cup \phi_s(r)$ and bijection domain)
3. $r \cdot s = r \cup \phi_r(s) = \phi_r(s) \cdot \phi^{-1}_{\phi_r(s)}(r)$.

Definition 5.7 (label category) *A* label category L *on a label monoid $\mathcal{M}$ has*

- *as objects,* Ob_{L}
- *as arrows, for all* a, b *in* Ob_{L}, $\mathrm{L}[a, b]$
- *an associative endo-bifunctor* $_ \cdot _ : \mathrm{L} \times \mathrm{L} \to \mathrm{L}$
- *and a function* $L : \mathrm{L} \to \mathcal{M}$ *such that*
  - *i.* $L \circ (_ \cdot _) = (_ \cdot _) \circ \langle L \times L \rangle : \mathrm{L} \times \mathrm{L} \to \mathcal{M}$,
  - *ii. associativity:* $\forall a, b, c \in \mathrm{L},\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$
  - *iii. neutral element:* $\forall a, b \in \mathrm{L},\ (L(a) = \emptyset) \Rightarrow a \cdot b = b$

A very simple label category is the discrete category with objects in $\mathcal{M}$, $\mathcal{L}$ and ϕ being those of streams, and L the identity.

Example 5.1 (Sel) A little more interesting one is the category of labeled finite sets (or streams) and selections Sel which is defined as

- for objects $Ob_{\mathrm{Sel}} = \mathcal{M}$
- for morphisms, the selections. That is, for all a, b in Ob_{Sel}, $\mathrm{Sel}[a, b] = a^b$
- L is the identity.
- reverse composition : $\forall f \in \mathrm{Sel}[a, b], g \in \mathrm{Sel}[b, c], l \in L(c), (g \circ f)(l) = f(g(l))$
- $_ \cdot _$ is defined as in $\mathcal{M}$ for objects, and for morphisms we extend with identity. More precisely, for any f in $\mathrm{Sel}[a, b]$,

$$(r \cdot f)(l) = \begin{cases} l & \text{if } l \in r \\ \phi_r(f(\psi_r(l))) & \text{otherwise} \end{cases}$$

We have similarly

$$(f \cdot r)(l) = \begin{cases} f(l) & \text{if } l \in L(b) \\ \phi_{L(a)}(\psi_{L(b)}(l)) & \text{otherwise} \end{cases}$$

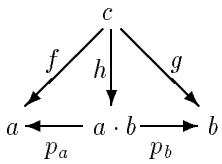
Why consider this category and not the more natural dual one? The λ-calculus with no constants does nothing else than select variables and combine, so that a great part of it can be seen through selection.

To see that this is a label category we just have to verify that $_ \cdot _$ is really a functor. For any f in $\mathrm{Sel}[a, b]$ and g in $\mathrm{Sel}[b, c]$,

$$((r \cdot g) \circ (r \cdot f))(l) = \begin{cases} l = (r \cdot (g \circ f))(l) & \text{if } l \in r \\ \phi_r(f(\psi_r(\phi_r(g(\psi_r(l)))))) = \phi_r((g \circ f)(\psi_r(l))) & \text{otherwise} \end{cases}$$

and similarly for the left side.

Definition 5.8 (Cartesian label category) *A* Cartesian label category L *on $\mathcal{L}$ is a label category on $\mathcal{L}$ in which the label product $_\cdot_$ is a Cartesian product. That is, we have chosen $p_a : a \cdot b \to a$ and $p_b : a \cdot b \to b$ such that, for any $f \in \mathrm{L}[c,a]$ and $g \in \mathrm{L}[c,b]$, there exists exactly one $h \in \mathrm{L}[c, a \cdot b]$ such that the following diagram commutes:*



Then for any $f \in \mathrm{L}[a,c]$ and $g \in \mathrm{L}[b,d]$, we can define $f \cdot g =< f \circ p_a, g \circ p_b >: a \cdot b \to c \cdot d$.

Example 5.2 1. The stream monoid (*cf.* Chapter 3) with appropriate arrows is a Cartesian label category.

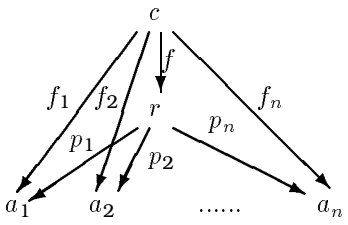
2. The monoid part of the restricted abstraction model forms a category where streams are objects and transformations are morphisms.

5.5.2 Record category

If label categories introduce some notion of product through the bifunctor, this is very far from a Cartesian product in two ways. First we have no real conditions on the properties of the result, and second, all is on the same level so that product does not abstract.

In label categories we can replace the notion of Cartesian product by that of record[3]. Intuitively, a record is a finite product in which we access to information through labels (nothing new.)

Definition 5.9 (record) *A record of the n-uple (a_n) of objects of* L *with label (l_n) ($l_i < l_{i+1}$) is an object $r = \{l_1 \Rightarrow a_1, \ldots, l_n \Rightarrow a_n\}$ together with a tuple of morphisms (p_n) in $(\mathrm{L}[r,a_1], \ldots, \mathrm{L}[r,a_n])$, such that $L(r) = (u_n)$ and, for any c in Ob_{L}, and n morphisms (f_n) in $\mathrm{L}[c,a_i]$ there exists exactly one morphism f in $\mathrm{L}[c,r]$ verifying $\forall i \in [\![1,n]\!]$, $f_i = p_i \circ f$, or equivalently the following diagram commutes.*



We could equivalently define a record with label (l_n) as a limit for a diagram indexed on a discrete graph with n nodes. In fact, a record is only a labeled indexed product.

Definition 5.10 (record category) *A* record category R *on $\mathcal{L}$ is a Cartesian label category on $\mathcal{L}$ with a family of endofunctors indexed on $\mathcal{L}$, $\{l \Rightarrow _\} : \mathrm{R} \to \mathrm{R}$, such that a is isomorphic to $\{l \Rightarrow a\}$ and $\{l \Rightarrow a\} \cdot \{l' \Rightarrow b\} = \{\phi_{\{l\}}(l') \Rightarrow b\} \cdot \{\phi^{-1}_{\phi_{\{l\}}\{l'\}}(l) \Rightarrow a\}$, and an object t such that $L(t) = \emptyset$.*

Proposition 5.6 R *has the following properties:*

1. t is terminal.

[3]These records correspond to streams in the syntax, but we were less directive in the definition of the label monoid.

2. for any label (l_n), and any tuple (a_n) of r there is a record.

PROOF Since R is a label category, for any a in R, $a \cdot t = a$. Since it is Cartesian, we use the diagram with $c = a$, $b = t$, and $f = id$. We deduce from it that $h = p_a^{-1}$, so that g has to be $p_b \circ p_a^{-1}$. But the choice of g does not depend on the rest of the diagram, so it means that there is only one arrow from a to t.

With t, labeling, and product, we can produce any record. □

Example 5.3 (TSel) The category of selections on trees is a record category. Its fundamental objects are atomic labels in $\mathcal{L}$, and all others are given by closure of product and labeling.

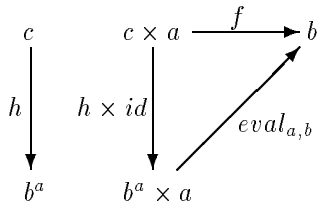
Selections between two objects a and b of Ob_{TSel} are selections between their sets of leaves (we see records as trees.) Extension through $_ \cdot _$ is done as before, but according to the number of leaves.

Since labeling doesn't add leaves, all works as before.

5.5.3 Record closed category

It is what one expects, a model of simply typed selective λ-calculus.

Definition 5.11 (exponent) *Let* C *be a Cartesian category and a, b in Ob_{C}. The* exponent *of a and b is an object b^a together with a morphism $eval_{a,b} : b^a \times a \to b$, such that for all morphisms $f : c \times a \to b$, there exists one and only one $h : c \to b^a$ such that the following diagram commutes :*



For record categories, replace $_ \times _$ by $_ \cdot _$, the exponent should labeled as b.

Definition 5.12 (record closed category) *A* record closed category R *on $\mathcal{L}$ is a record category on $\mathcal{L}$ which has an exponent for every pair of objects, and it satisfies the conditions:*

i. for all a in $\mathcal{A}$, $a^t = a$, and $eval_{t,a}$ is the identity on a.

ii. for all a, b in $\mathcal{P}$, c in $\mathcal{A}$, $(c^b)^a = c^{a \cdot b}$, and $eval_{b,c} \circ (eval_{a,c^b} \cdot id_b) = eval_{a \cdot b, c}$.

Proposition 5.7 *Any record closed category is a Cartesian closed category.*

PROOF by definition. □

Definition 5.13 (generator) *Let* C *be a category. $t \in Ob_{\text{C}}$ is a* generator *iff for all $a, b \in Ob_{\text{C}}$ and all $f, g \in \text{C}[a, b]$, $f \neq g \Rightarrow \exists h \in \text{C}[t, a] f \circ h \neq g \circ h$.*

Proposition 5.8 *A RCC* R *where there is an object for each base type and a terminal object that is a generator provides a model of simply typed selective λ-calculus.*

PROOF We first remark that a model of simply type selective λ-calculus is a model of simply typed model, since the last introduces types. We start from the model of the beginning of this chapter. Let t, $L(t) = o$ be a generator terminal object of R. For any υ in $\mathcal{T}_0$, $\mathcal{A}^\upsilon$ is represented by R$[t, a_\upsilon]$. For any τ in $\mathcal{T}$, a_τ is obtained by the corresponding record constructions and exponentiations.

For $a, b \in Ob_{\mathrm{R}}$, $f : t \to b^{(p \Rightarrow a)}$, $x : t \to a$, we define

$$f \mathbin{\widehat{p}} b \stackrel{\text{def}}{=} eval_{(p \Rightarrow a),b} \circ < f, p_1^{-1} \circ x >: t \to b.$$

where $p_1 : (p \Rightarrow a) \to a$ is the isomorphism of record construction. □

5.5.4 Multi-categorical model

It is true that RCCs provide models of selective λ-calculus, but they express much more than necessary. In fact they provide a selective λ-calculus on a datatype of labeled trees, which is not our initial goal. This superfluous structure came from the fact we expressed everything in the same category. The idea here is to separate clearly terms and environments into two different categories.

Definition 5.14 *The category $\mathcal{P}$ of records on a category $\mathcal{A}$, with labels in $\mathcal{L}$, is a $\mathcal{M}$-Cartesian label category such that*

a. *we have a family of full and faithful covariant functors $\{\mathbin{\widehat{p}} : \mathcal{A} \to \mathcal{P} \mid p \in \mathcal{L}\}$, called labeling functors ($L(\mathbin{\widehat{p}}\, a) = \{p\}$ and $\{l \Rightarrow a\} \cdot \{l' \Rightarrow b\} = \{\phi_{\{}l\}(l') \Rightarrow b\} \cdot \{\phi^{-1}_{\phi_{\{l\}}\{l'\}}(l)\}$),*

b. *and $Ob_{\mathcal{P}}$ is the closure of the union of images of $\{\mathbin{\widehat{p}} \mid p \in \mathcal{L}\}$ and a terminal object t generator of $\mathcal{P}$, $L(t) = o$, by $_ \cdot _$.*

Definition 5.15 *A category $\mathcal{A}$ of 1-morphisms over a $\mathcal{L}$-Cartesian label category $\mathcal{P}$ with a terminal object t generator of $\mathcal{P}$ has*

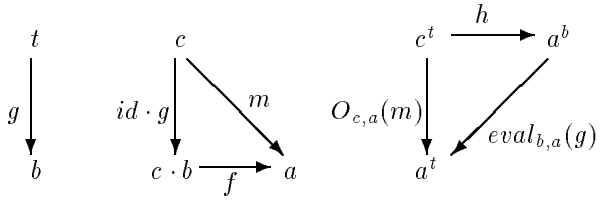
a. *contravariant functors $A_a : \mathcal{P} \to \mathcal{A}$, for each a in $\mathcal{P}$ such that $L(a) = 1$; for objects we write $a^b = A_a(b)$*

b. *bijective functions $O_{a,b} : \mathcal{P}[a, b] \to \mathcal{A}[a^t, b^t]$, for $a, b \in Ob_{\mathcal{P}}$, L(a)=L(b)=1,*

c. *for each $a, L(a) = 1$ functions $eval_{b,a} \in \mathcal{P}[t, b] \to \mathcal{A}[a^b, a^t]$ such that for any $c \in Ob_{\mathcal{P}}$, $L(c) = 1$ and any $f : c \cdot b \to a$ there exists one and only one $h : c^t \to a^b$ such that*

$$\forall g \in \mathcal{P}[t, b],\ eval_{b,a}(g) \circ h = O_{c,a}(f \circ (id \cdot g))$$

That is, for any f there is a unique h such that the following diagram commutes for any g.



Definition 5.16 (label-exponential pair) *A* label-exponential pair *of categories $\langle \mathcal{A}, \mathcal{P} \rangle$ is such that*

a. $\mathcal{P}$ *is the category of records on* $\mathcal{A}$*,*

b. $\mathcal{A}$ *is a category of 1-morphisms over* $\mathcal{P}$*,*

c. we have the equalities $O_{a,b} \circ \widehat{1}_{a^t,b^t} = Id_{a^t,b^t}$ *and* $\widehat{1}_{a^t,b^t} \circ O_{a,b} = Id_{a,b}$*,*

d. we have the associativity of arguments

$$eval_{b,c}(f) \circ eval_{a,\{1 \Rightarrow c^b\}}(g) = eval_{a \cdot b,c}((a \cdot f) \circ g).$$

Arrows in $\mathcal{P}$ are meant to express transformations (functions between two records), while in $\mathcal{A}$ they may express the presence of a preprocessing transformation: if $r \in \mathcal{P}[b,c]$, $f \in \mathcal{A}[d,a^c]$, $L(a) = 1$, then $f \circ A_a(r) \in \mathcal{A}[d,a^b]$.

Proposition 5.9 *Any label-exponential pair* $\langle \mathcal{A}, \mathcal{P} \rangle$ *where* A_a*'s are faithful is a restricted abstraction model of simply typed selective* λ*-calculus.*

PROOF Chose objects in $\mathcal{A}$ to represent your base types. The extensive representation of a base type u is $\mathcal{P}[t, \widehat{1}\, a^u]$. We obtain an object for each type inductively:

$$\{1 \Rightarrow a^u\}^{\{l_1 \Rightarrow a^{\tau_1}, \ldots, l_n \Rightarrow a^{\tau_n}\}} = a^{\{l_1 \Rightarrow \tau_1, \ldots, l_n \Rightarrow \tau_n\} \to u}$$

The construction of such a record is insured by the commutation equality.

For each type τ, abstractions of functions towards this type is obtained by $A_{\{1 \Rightarrow a^\tau\}}$ from objects of $\mathcal{P}$.

Multi-application is obtained as, with non normalized types,

$$\forall f : (r_1 \cdot r_2 \to u) \in \mathcal{P}[t, \{1 \Rightarrow \{1 \Rightarrow a^{r_2 \to u}\}^{r_1}], \ \forall p : r_1 \in \mathcal{P}[t, b^{r_1}],$$
$$[f \cdot p] = \{1 \Rightarrow eval_{b^{r_1}, a^{r_2 \to u}}(p) \circ f\} : r_2 \to u \in \mathcal{P}[t, \{1 \Rightarrow a^{r_2 \to u}\}].$$

Transformations are morphisms in $\mathcal{P}$. □

Remark We have to require faithfulness of A_a's to ensure all transformations are obtained. Without this restriction we can construct label-exponential bicategories in which, for instance, we have only constant functions.

Example 5.4 The category of set representations of types and maps, coupled with the category of records constructed on it, is a label exponential pair. The construction is just the same.

Proposition 5.10 *A label-exponential pair* $\langle \mathcal{A}, \mathcal{P} \rangle$ *where* A_a*'s are faithful and there is an object* U *in* $\mathcal{A}$ *such that for any label* p*,* $(\{1 \Rightarrow U\})^{\{p \Rightarrow U\}} < U$ *by* (ϕ, ψ) *provides a restricted abstraction model of selective* λ*-calculus.*

5.5.5 Mono-categorical model

The multi-categorical model seems a success, in that it correctly models the selective λ-calculus, without introducing more values than necessary.

However, the separation in two categories introduces lots of complexity. The question is then: Do we really need to restrict that much our model? Wouldn't it be possible to put a little more in, and get a simpler structure? This is a classical way of proceeding: for instance the usual CCC model for the λ-calculus introduces pairs in the model, which didn't exist in the original λ-calculus.

Since records roughly correspond to pairs, one could think that we are already done with the RCC model. But there is an essential difference between records and products: records add a notion of labeling, which distinguishes $\{1 \Rightarrow a\}$ from a. We need more discernment.

The solution we propose here is to limit some properties to subcategories of a full category modeling our calculus. It appears to be quite successful, since the resulting category corresponds to an interesting extension of selective λ-calculus, the *transformation calculus*, whose syntax is studied in Chapter 6.

Definition 5.17 (function-transformation category) *A* function-transformation *category $\mathcal{A}$, containing* functions *has:*

a. a subcategory $\mathcal{P}$, which is itself a Cartesian label category, which contains the records,

b. a family of faithful labeling functors indexed by $\mathcal{L}$, $\{l \Rightarrow _\} : \mathcal{A} \to \mathcal{P}$, such that $L(\{l \Rightarrow a\}) = \{l\}$, $\{l \Rightarrow a\} \cdot \{l' \Rightarrow b\} = \{\phi_{\{l\}} l\}(l') \Rightarrow b\} \cdot \{\phi^{-1}_{\phi_{\{l\}}\{l'\}}(l)$, and a and $\{l \Rightarrow a\}$ are isomorphic.

c. an object t generator in $\mathcal{P}$ such that $L(t) = \emptyset$,

d. an exponential bifunctor (contravariant in its first argument), $A : \mathcal{P} \times \mathcal{A} \to \mathcal{A}$. We note b^a for $A(a,b) \in \mathcal{A}$, and we have morphisms $eval_{a,b} : \{1 \Rightarrow b^a\} \cdot a \to b$ such that $(b^a, eval_{a,b})$ is an exponent of a and b (cf. Def. 5.11, replacing $_ \times _$ by $\{1 \Rightarrow _\} \cdot _$)

e. a subcategory $\mathcal{T}$, formed by the exponents of objects of $\mathcal{P}$, which contains the transformations,

f. an extension bifunctors $E : \mathcal{T} \times \mathcal{P} \to \mathcal{T}$, such that $E(b^a, c) = (b \cdot c)^{a \cdot c}$ and $a < E(a,b)$,

g. and satisfies the following conditions,

(i) for all a in $\mathcal{A}$, $a^t = a$, and $eval_{t,a}$ is the canonical isomorphism between $\{1 \Rightarrow a\}$ and a.

(ii) for all a, b in $\mathcal{P}$, c in $\mathcal{A}$, $(c^b)^a = c^{a \cdot b}$, and $eval_{b,c} \circ (eval_{a,\{1 \Rightarrow c^b\}} \cdot id_b) = eval_{a \cdot b, c}$.

Properties The following are properties of function-transformation categories:

1. $\mathcal{P}$ is a record category. (by definition)
2. t is terminal in $\mathcal{A}$. (follows 1 + isormorphism between a and $\{l \Rightarrow a\}$)
3. $\mathcal{P}$ is a subcategory of $\mathcal{T}$. ($a^t = a$)
4. If $b \in \mathcal{T}$, then $b^a \in \mathcal{T}$.

Proposition 5.11 *A function-transformation category $\mathcal{A}$ is a model of the typed transformation calculus, where $\mathcal{P}$ models streams, $\mathcal{T}$ models transformations, and $\mathcal{A}$ models functions.*

5.6 Final remarks on semantics

This chapter gave a lot of definitions, and very few theorems and proofs. This is related to our approach, in which models are rather supposed to give an intuition of what is behind the syntax, than to be studied for themselves.

The most important result of this approach starts with the next chapter, in which we define the transformation calculus. This calculus is the syntactical interpretation of all the semantic constructions of this chapter. As such, the relation between transformation calculus and function-transformation categories is close to that between the categorical combinatory logic and Cartesian closed categories [Cur93], eventhough our approach is less formal.

Chapter 6

The transformation calculus

This chapter presents the transformation calculus, starting with an operational view based on stack machines. Then the calculus is formally defined, and the notion of scope-free variable is introduced. It closes with the definition of transformational combinators, ie we had in lambda calculus.

We only deal here with the basic untyped transformation calculus, typing and extensions being the objet the next chapter.

6.1 Introduction

Currying is as old as lambda calculus. For the simple reason that, in raw lambda calculus —without pairing or similar built-in constructs—, this is the only way to represent multi-argument functions. This just means that we will write

$$\lambda x.\lambda y.M[x,y]$$

in place of

$$(x,y) \mapsto M[x,y].$$

At this stage appears a first asymmetry: while in the pair (x,y) the two variables play symmetrical roles, in $\lambda x.\lambda y.M$ they don't. An implicit order was introduced. Materially this means that we can partially apply our function directly on x but not on y.

We now look at types. There, currying can be seen as isomorphism of types [BCL90]:

$$(A \times B) \to C \simeq A \to B \to C.$$

Here comes another asymmetry: why don't we get any similar isomorphism for $A \to (B \times C)$.

The calculus we will present here generalizes currying to these two kinds of symmetries: between arguments, and between input and output. For the first one, we are just taking over the mechanism of *label-selective currying* developped in Chapter 3.

For the second one we develop a new notion of *composition*, which, contrary to the usual one, is compatible with currying.

The resulting system, *transformation calculus*, is a conservative extension of lambda calculus. Why such a name? Because this essentially syntactic extension —semantics stay very similar (*cf.* Section 7.2— provides us with a new way of representing state transformations, *i.e.* state being represented by labeled input parameters, that may get returned by our term. Handling state as a supplementary parameter that gets returned with the result is not new. But by extending currying we get more flexibility, in two ways.

First, since a part of the state is no more than a labeled parameter, we can dynamically extend it by simply adding a new parameter at some point in our term. Second, extended currying lets a transformation ignore parts of the state it doesn't need. They will just be left unmodified.

To demonstrate our point, we introduce *scope-free variables*, which are trivially encoded in the transformation calculus, and can be used in place of usual scoped mutable variables, in the Algol tradition. Since they have no syntactic scope, scope-free variables respect dynamic binding rather than static binding; but they are more flexible than Algol variables, while simulating blocks and stack discipline.

6.2 Composition and streams

We first introduce informally and progressively the features of our calculus. We start from the classical pure lambda-calculus, that is

$$M ::= x \mid \lambda x.M \mid M\,M$$

with β-reduction

$$(\lambda x.M)N \rightarrow_\beta [N/x]M$$

and where terms are considered modulo α-conversion (renaming of bound variables).

6.2.1 Implicit currying

Currying is the fundamental transformation by which multi-argument functions are encoded in the lambda-calculus. It can appear in abstractions as well as applications. For instance $f(a,b)$ will be encoded as $(f(a))(b)$ ($f\,a\,b$ without parentheses), and $\lambda(x,y).M$ becomes $\lambda x.\lambda y.M$.

This operation does not modify the nature of calculations, since clearly $(\lambda(x,y).M)(a,b)$ and $(\lambda x.\lambda y.M)\,a\,b$ reduce to the same $[a/x,b/y]M$ (provided x and y are distinct variables). Currying can be extended to an arbitrary number of arguments, *i.e.* $\lambda(x_1,\ldots,x_n).M$ is encoded as $\lambda x_1.\cdots.\lambda x_n.M$. As long as we encode similarly applications and abstractions, no problem should appear.

By implicit currying, we mean that we will write curried and uncurried versions of terms indifferently, always supposing that we reduce curried ones. Of course we work in the pure lambda-calculus without pairing, so that no confusion is possible. The new syntax becomes

$$M ::= x \mid \lambda(x,\ldots).M \mid M(M,\ldots)$$

where abstracted variables under the same λ should be distinct. Implicit currying is expressed by the two structural equivalences:

If all x_i's are distinct then

$$\begin{array}{rcl} \lambda(x_1,\ldots,x_n).M & \equiv_\cdot & \lambda(x_1,\ldots,x_k).\lambda(x_{k+1},\ldots,x_n).M \\ M(N_1,\ldots,N_n) & \equiv_\lambda & M(N_1,\ldots,N_k)(N_{k+1},\ldots,N_n) \end{array}$$

$\equiv$ is defined as the relflexive, symmetric and transitive closure of $\equiv_\varsigma$'s.

β-reduction applying on the implicitly curried form, a reduction step only binds one variable.

$$(\lambda(x,y).M)(a,b) \rightarrow_\beta (\lambda(y).[a/x]M)(b) \rightarrow_\beta [a/x,b/y]M$$

In fact, if we remember the habit many have of writing $(\lambda xy.M)\,a\,b$ for the above term, we have done absolutely nothing new. However this syntax lets us emphasize some natural groupings of values. For instance the encoding of pairs in lambda-calculus can be written as $\lambda(x,y).\lambda f.f(x,y)$.

6.2.2 Composition

The next step is to introduce a binary *composition* operator (";") and a *transformation* constructor ("$\downarrow$"). A transformation is what should appear on the left side of a composition, for a reduction to progress correctly.

$$M ::= \ldots \mid \downarrow \mid M;M$$

It will be more intuitive to use an alternative syntax for application,

$$M ::= \ldots \mid (M,\ldots).M$$

with $(N_1,\ldots).M \equiv M(N_1,\ldots)$, and both the dots of abstraction and application binding tighter than composition.

Together we add a new reduction rule, and a new structural equivalence, to eliminate compositions. Some other equivalences are introduced in the actual calculus, to enable earlier flattening of terms, but we leave them for later.

$$\begin{array}{rcl} \downarrow;M & \rightarrow_{\downarrow} & M \\ (N_1,\ldots,N_k).(M_1;M_2) & \equiv_{.;} & (N_1,\ldots,N_k).M_1;M_2 \end{array}$$

We can see the sequencing role of composed pairs as follows: when we apply $(M_1;M_2)$ to a sufficient input tuple of arguments, we first apply M_1 to this tuple, get (hopefully) a *tuple-term* (term of form $(N_1,\ldots,N_k).\downarrow$) as result of its reduction, and apply M_2 to this result tuple.

It just looks like we added a stack machine into the lambda-calculus. For instance, we can write the transformation that switches two terms on top of a stack as $sw = \lambda(x,y).(y,x).\downarrow$, and can apply it to an input tuple of any size:

$$\begin{array}{rrl} & (a,b,c) & .\lambda(x,y).(y,x).\downarrow \\ \rightarrow_{\beta} & (b,c) & .\lambda(y).(y,a).\downarrow \\ \rightarrow_{\beta} & (c) & .(b,a).\downarrow \\ \equiv_{.} & (b,a,c) & .\downarrow \end{array}$$

Composed with another term, it plays the same role as the $C = \lambda f.\lambda(x,y).f(y,x)$ combinator; but in a postfix way.

$$\begin{array}{rrl} & (c) & .(a,b).(sw;K) \\ \equiv_{.} & (a,b,c) & .(sw;K) \\ \equiv_{.;} & (a,b,c) & .sw;K \\ \stackrel{*}{\rightarrow} & (b,a,c) & .\downarrow;K \\ \equiv_{.;} & (b,a,c) & .(\downarrow;K) \\ \rightarrow_{\downarrow} & (b,a,c) & .\lambda(x,y).x \\ \stackrel{*}{\rightarrow} & (c) & .b \end{array}$$

Since we are in the lambda-calculus, we can define the fix-point operator Y. We just define then loops in terms of this operator. The functional for a while-do loop can be defined as

$$\begin{array}{rcl} \text{while} & = & Y(\lambda whl. \\ & & \quad \lambda(end,do).(end; \\ & & \quad\quad \lambda b.\text{if } b \text{ then } do; whl(end,do) \text{ else } \downarrow) \\ & &) \end{array}$$

The end-condition is a transformation that adds to its input a boolean b, false to end, true to go on, leaving the rest in position. do may change the values from the input, but not their number. Such a functional works on a state of any size.

An imperative version of Euclid's algorithm for the greatest common divisor can then be written

$$\begin{array}{l}\text{while}(\lambda(x).(x \neq 0, x).\downarrow,\\ \qquad \lambda(x,y).(y \bmod x, x).\downarrow);\\ \lambda(x,y).y\end{array}$$

We notice here an important difference between this "while" functional and something equivalent written using pairing. Here our end-condition only uses x, whereas a functional using pairing would have required it to receive the whole state eventhough y is not needed. This remark will become even more important when we will add to our calculus the power of *selective currying*.

6.2.3 Selective currying

Combining lambda-calculus and a stack machine should be enough to express algorithms both in their functional and imperative form. However, in chosing a reduction system rather than an equational theory to express our calculus, we are interested in giving some meaning to the reductions themselves. We can think of various meanings, like complexity, sequentiality constraints, *etc...* If we do such an analysis, then we see that, with simple implicit currying, we need much more reduction steps than should be necessary. That is, when we want to access the 5^{th} element of a tuple we have to extract successively all the elements before it, and put them back:

$$\lambda(x_1,x_2,x_3,x_4,x_5).(x_5,x_1,x_2,x_3,x_4).\downarrow$$

Even more than the number of reductions, we should be preoccupied by the fact we have accessed four unrelated values to move only one. And this in an asymmetrical way, values after the fifth element being left untouched.

6.2.3.1 Indexed streams

The answer to this problem comes from the canonical injection of tuples into records, as it can be found functional languages like Standard ML or LIFE. That is

$$(x_1,\ldots,x_n) \equiv \{1 \Rightarrow x_1,\ldots,n \Rightarrow x_n\}$$

We will just consider tuples as a particular case of *streams* (the name we give to these numerically indexed records). We use streams to access directly the arguments we are interested in. For instance we write the previous transformation

$$\lambda\{5 \Rightarrow x\}.\{1 \Rightarrow x\}.\downarrow$$

We extract the 5^{th} element from the input, and put it in first position.

By symmetry we can use such incomplete streams in applications as well as abstractions. The transformation $\{5 \Rightarrow a\}.\downarrow$ inserts a before the fifth argument of its input, pushing up all its followers by one.

$$(b_1,b_2,b_3,b_4,b_5,b_6).\{5 \Rightarrow a\}.\downarrow \rightarrow_\beta (b_1,b_2,b_3,b_4,a,b_5,b_6).\downarrow$$

The stream we apply our transformation to can be incomplete too, like in the following case.

$$\{2 \Rightarrow a, 5 \Rightarrow b, 7 \Rightarrow c\}.\{5 \Rightarrow d\}.\downarrow \rightarrow_\beta \{2 \Rightarrow a, 5 \Rightarrow d, 6 \Rightarrow b, 8 \Rightarrow c\}.\downarrow$$

Generally, we must define a concatenation operation on streams, compatible with the partial isomorphism between streams and tuple. This is done by shifting indexes in the inserted stream according to those present in the original one. These mechanisms are described in Section 3.1. The important point is that we have a reciprocal operation, sub-stream extraction, which we can use to separate a stream into two parts, which form it back by concatenation.

$$\begin{array}{ll} & \{1\Rightarrow a, 2\Rightarrow b, 3\Rightarrow c, 5\Rightarrow d, 7\Rightarrow e\}.\lambda\{2\Rightarrow x, 3\Rightarrow y\}.M \\ \equiv & \{1\Rightarrow a, 3\Rightarrow d, 5\Rightarrow e\}.\{2\Rightarrow b, 3\Rightarrow c\}.\lambda\{2\Rightarrow x, 3\Rightarrow y\}.M \\ \xrightarrow{*} & \{1\Rightarrow a, 3\Rightarrow d, 5\Rightarrow e\}.[b/x, c/y]M \end{array}$$

This operation can be applied to the abstraction part too:

$$\begin{array}{ll} & \{2\Rightarrow a, 4\Rightarrow b\}.\lambda\{2\Rightarrow x, 3\Rightarrow y\}.M \\ \equiv & \{3\Rightarrow b\}.\{2\Rightarrow a\}.\lambda\{2\Rightarrow x\}.\lambda\{2\Rightarrow y\}.M \\ \rightarrow_\beta & \{3\Rightarrow b\}.\lambda\{2\Rightarrow y\}.[a/x]M \\ \equiv_\lambda & \lambda\{2\Rightarrow y\}.\{2\Rightarrow b\}.[a/x]M \end{array}$$

In the second line we decompose both abstraction and application to permit β-reduction in the third one. The $\equiv_\lambda$ equivalence in the last line is there to switch unrelated abstractions and applications, and let the reduction progress smoothly. We will argue later its coherence with the rest of the system.

Thanks to these streams we can now write algorithms using the power of something quite close to a direct-access stack machine. It is slightly different since reading a position in the stream destroys this position, but writing resulting in a symmetrical insertion this is not a problem. Here is a transformation incrementing the fifth position in a stream.

$$\begin{array}{lll} & (a,b,c,d,e) & .\lambda\{5\Rightarrow x\}.\{5\Rightarrow x+1\}.\downarrow \\ \equiv & (a,b,c,d) & .\{5\Rightarrow e\}.\lambda\{5\Rightarrow x\}.\{5\Rightarrow x+1\}.\downarrow \\ \rightarrow_\beta & (a,b,c,d) & .\{5\Rightarrow e+1\}.\downarrow \\ \equiv & (a,b,c,d,e+1) & .\downarrow \end{array}$$

6.2.3.2 Naming positions

The problem of such a system is that since the indexes in the stream may change with each transformation we apply to it, we have no uniform way to address a defined position in it. This increments the fifth position by the first (which is destroyed):

$$\lambda\{1\Rightarrow x, 5\Rightarrow y\}.\{4\Rightarrow x+y\}.\downarrow$$

The position we addressed as 5^{th} before the transformation must become the 4^{th} after it, since we do not distinguish arguments (x) from mutable variables (y).

This possibility of mixing is good, since it means that we can see everything with a functional insight. However we would like to have a more uniform way to handle a position. Going on with our analogy between streams and records, we will accept to have named fields in our streams. So that we can write the previous incrementer as

$$\lambda\{1\Rightarrow x, i\Rightarrow y\}.\{i\Rightarrow x+y\}.\downarrow$$

Since i is a named position its index is not modified by the extraction of the first one.

A problem may appear when we concatenate two streams containing the same named position. On records this operation has two definitions. Either we just refuse to do it (symmetrical concatenation), either we accept, take for this field the value in one

operand, and just forget the value in the other (asymmetrical concatenation). Since concatenation is already asymmetrical on numerical indexes, there is no point in refusing to do such a thing. However we cannot erase a value since we would lose confluence: $\{i \Rightarrow b\}.\{i \Rightarrow a\}.\lambda\{i \Rightarrow x\}.\lambda\{i \Rightarrow y\}.M$ can reduce to $[a/x, b/y]M$ but $\{i \Rightarrow b\}.\lambda\{i \Rightarrow x\}.\lambda\{i \Rightarrow y\}.M$ cannot. So we just add a numerical index to our position name. That is, we view both numerical and name positions as their injections into their product, the set of labels. We have a default name ϵ such that the index n is in fact ϵn, and we add 1 to names so that p is $p1$. With that we define easily

$$\{p \Rightarrow b\}.\{p \Rightarrow a\}.M \equiv \{p1 \Rightarrow a, p2 \Rightarrow b\}.M$$

This works right with the above example, which becomes $\{i1 \Rightarrow a, i2 \Rightarrow b\}.\lambda\{i1 \Rightarrow x, i2 \Rightarrow y\}.M$.

Going on with our comparison with stacks, we have now as many stacks as we have names, each of them handled through indexed extraction and insertion.

We see here a new version of Euclid's algorithm, using labels for both the function and the while functional.

$$\begin{array}{l}\text{while}\{end \Rightarrow end, do \Rightarrow do\} = \\ \qquad end; \lambda\{ok \Rightarrow ok\}. \\ \qquad\ \text{if } ok \text{ then } do; \text{while}\{end \Rightarrow end, do \Rightarrow do\} \text{ else } \downarrow \end{array}$$

$$\begin{array}{ll}\text{gcd} = & \lambda(x, y).\{m \Rightarrow x, n \Rightarrow y\}.\downarrow; \\ & \text{while}\{end \Rightarrow \lambda\{m \Rightarrow m\}.\{ok \Rightarrow m \neq 0, m \Rightarrow m\}.\downarrow, \\ & \quad do \Rightarrow \lambda\{m \Rightarrow m, n \Rightarrow n\}.\{m \Rightarrow n \bmod m, n \Rightarrow m\}.\downarrow\}; \\ & \lambda\{m \Rightarrow m, n \Rightarrow n\}.n \end{array}$$

On such an example the addition of labels may look as pure verbosity, but what we obtain here is very close to what we would write as an imperative algorithm. We have only to add trivial abstractions of the form $\lambda\{m \Rightarrow m\}$ in order to transform an assignment-like syntax into functions.

6.2.4 Stream behaviour

The examples we presented above worked all right, but what happens with "incorrect" terms, that are not well-behaved?

We had already such terms in classical lambda-calculus. For instance, if we encode an if-then-else by a pair $\lambda s.(s\ t\ e)$, where s is expected to be an encoded boolean, and t and e the two cases, we expect in most cases t and e to be well-behaved, that is if t encodes a pair, then e should also encode a pair. Otherwise, we will have an unexpected behaviour when trying to apply a projection on it.

This problem of behaviour is even more pernicious with the transformation calculus. Again in an if-then-else we expect the two branches to have similar behaviour. But even if the second one gives back a stream with more labels than the first, it may well not appear, as long as we only use transformations that only access labels present in the first stream. This is still an incoherence.

So, by *well-behaved*, we will mean here that for any acceptable input with same stream structure, a transformation should give back a stream with same labels. That is, its *stream-behaviour*, the stream structure of the result with respect to the stream structure of the input, should not be dependent on encoded values in the input.

For instance,

$$\lambda b.\text{if } b \text{ then } \{l \Rightarrow M\}.\downarrow \text{ else } \downarrow$$

is not well behaved since it returns either a stream with label l or an empty stream, depending on the value of b.

This is difficult to give a precise definition of *well-behaved* terms in an untyped framework, since it depends on what we are encoding. In a typed framework that amounts to subject reduction, and we give in the next chapter, Section 7.1, a simply typed transformation calculus that satisfies it (*i.e.* all typable terms are well-behaved).

6.2.5 Scope-free variables

Up to this point we have progressively enriched the lambda-calculus with new constructs. The transformation calculus is approximately the result of this process. We have insisted on how this calculus was a potential basis for an integration of imperative and functional styles in the design of algorithms. Here we introduce a general method to directly map the imperative notion of variable into the transformation calculus.

In fact, what we mean by *scope-free variable* is slightly stronger than a mutable variable. We call it scope-free, since it is not syntactically scoped like in structured programming, neither is it global. We can say that it is local to a sequence of transformations, composed together.

A scope free variable is essentially a name v whose use in labels is exclusively reserved in the concerned sequence of transformations. This sequence is delimited by the creation of the variable with value a, encoded $\{v \Rightarrow a\}.\downarrow$, and its destruction by an abstraction, $\lambda\{v \Rightarrow x\}.\downarrow$. Between these, all transformations using or modifying this variable should once take it (through abstraction) and then put it back (by application), identical or modified. Typically a modification can be written $\lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}.\downarrow$. That is, the sequence has form:

$$\{v \Rightarrow a\}.\downarrow; \ldots; \lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}.\downarrow; \ldots; \lambda\{v \Rightarrow x\}.M$$

Since some transformations may be functionals, the recognition of such a structure is not immediate, but for instance m and n in the last version of Euclid's algorithm are scope-free variables.

The most interesting property of scope-free variables is that, like scoped variables, they have no effect outside of the sequence they are used in. That is, we can use the same label v outside of the sequence our scope-free variable is local to, without interference. A scope-free variable may even be used in a subsequence of another scope-free variable using the same label:

$$\{v \Rightarrow a\}.\downarrow; \ldots; \underline{\{v \Rightarrow b\}.\downarrow; \ldots; \lambda\{v \Rightarrow x\}.\downarrow}; \ldots; \lambda\{v \Rightarrow x\}.\downarrow$$

In the underlined subsequence the external scope-free variable is identifiable by the label $v + 1$ but comes back to v after.

Still, we must be careful that scope-free variables are not variables in the meaning of lambda-calculus: they appear on a completely different level, that of labels. Nor are they pervasive like would be references. We do not add side-effects to functions, but just provide some implicit way to manipulate a "stream" of arguments. That means that a function that is not called directly on this stream (through composition) will not access the scope-free variables it contains, and as such cannot have any imperative behaviour with respect to this stream. This is this limitation which permits us to assimilate scope-free variables with arguments, and still be a conservative extension of lambda-calculus.

We give two examples of the use of scope-free variables. The first one is a simple encoding of an imperative programming language *a la* Algol. The second one shows how scope-free variable are stronger than scoped ones.

We translate the following program:

```
begin
  var x=5, y=10;
  x := x+y;
  begin
    var x=3;
    y := x+y
  end
  x := x-y;
  return(x)
end
```

We expect this program to evaluate to $5+10-(3+10)=2$. The translation is:

$$
\begin{array}{l}
\{x \Rightarrow 5, y \Rightarrow 10\}.\downarrow; \\
\lambda\{x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x+y, y \Rightarrow y\}.\downarrow; \\
\quad \{x \Rightarrow 3\}.\downarrow; \\
\quad \lambda\{x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x, y \Rightarrow x+y\}.\downarrow; \\
\quad \lambda\{x \Rightarrow x\}.\downarrow; \\
\lambda\{x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x-y, y \Rightarrow y\}.\downarrow; \\
\lambda\{x \Rightarrow x, y \Rightarrow y\}.x
\end{array}
$$

It evaluates as follows, and gives the expected result.

$$
\begin{array}{rl}
 & \{x \Rightarrow 5, y \Rightarrow 10\}.\downarrow; \ldots \\
\{x \Rightarrow 5, y \Rightarrow 10\} & .\lambda \cdots .\{x \Rightarrow x+y, y \Rightarrow y\}.\downarrow; \ldots \\
\{x \Rightarrow 15, y \Rightarrow 10\} & .\{x \Rightarrow 3\}.\downarrow; \ldots \\
\{x1 \Rightarrow 3, x2 \Rightarrow 15, y \Rightarrow 10\} & .\lambda \cdots .\{x \Rightarrow x, y \Rightarrow x+y\}.\downarrow; \ldots \\
\{x1 \Rightarrow 3, x2 \Rightarrow 15, y \Rightarrow 13\} & .\lambda\{x \Rightarrow x\}\ \downarrow; \ldots \\
\{x \Rightarrow 15, y \Rightarrow 13\} & .\lambda \cdots .\{x \Rightarrow x-y, y \Rightarrow y\}.\downarrow; \ldots \\
\{x \Rightarrow 2, y \Rightarrow 13\} & .\lambda\{x \Rightarrow x, y \Rightarrow y\}.x \\
 & 2
\end{array}
$$

Note here that since we encode dynamic binding[1] for scope-free variables, we would get the same result even if the central part was defined as a subprogram: with scope-free variable, even Basic's subprograms would behave correctly, since we can create a scope-free variable before the call to pass a parameter, and destroy it after.

The above example still respects a scoping discipline: variables are created and destroyed in opposite order. To show the specificity of scope-free variables, we must disobey it.

Not respecting a scoping discipline seems quite dangerous for variables, and of little use in purely computing programs. However, if we think of IO's, then the situation is different. Imagine a program with structure

$$\overline{A;\underline{B};C}$$

in which we want the console to be redirected in part $A;B$, and the screen to be changed in $B;C$. We suppose that we have mutable variables con and scr to indicate respectively which console and which screen should be used. Moreover we do not know which were the console and screen before entering A.

[1] Dynamic binding is generally considered as bad. However, while dynamic binding would destroy referential transparency for static variables, defined only once — but we have usual λ-variables for that use — the case of mutable variables is not so clear. Since they are not referentially transparent w.r.t. their values, in a way dynamic binding is more natural and easier to model for them.

$$
\begin{array}{rcll}
l & ::= & pn & p \in \mathcal{L}_s, n \in \mathcal{N} \\
M & ::= & x & \text{variable} \\
 & | & \downarrow & \text{transformation constructor} \\
 & | & \lambda\{l \Rightarrow x, \ldots\}.M & \text{abstraction} \\
 & | & \{l \Rightarrow M, \ldots\}.M & \text{application} \\
 & | & M; M & \text{composition}
\end{array}
$$

Figure 6.1: Syntax of the transformation calculus

$$
\begin{array}{rcll}
S.R.M & \equiv_{.} & (R \cdot S).M & \\
\lambda R.\lambda S.M & \equiv_{\lambda} & \lambda(S \cdot R).M & V(R) \not\sharp V(S) \\
R.\lambda S.M & \equiv_{.\lambda} & \lambda\psi_R(S).\psi_S(R).M & FV(R) \not\sharp V(S), \mathcal{D}_R \not\sharp \mathcal{D}_S \\
 & & & \\
(R.M_1); M_2 & \equiv_{.;} & R.(M_1; M_2) & \\
(\lambda R.M_1); M_2 & \equiv_{\lambda;} & \lambda R.(M_1; M_2) & V(R) \not\sharp FV(M_2) \\
(M_1; M_2); M_3 & \equiv_{;} & M_1; (M_2; M_3) &
\end{array}
$$

Figure 6.2: Structural equivalences

A dirty method is to use temporary variables c and s, to store the old values:

```
c:=con; con:=newc; A; s:=scr;
scr:=news; B; con:=c; C; scr:=s
```

The problem is that these temporary variables may be modified by error in A, B or C. So a better solution is to use static variables, only set once:

```
let c = !con in
  con:=newc; A ;
  let s = !scr in
    scr:=newc; B ; con:=c; C ; scr:=s
end end
```

However, because of the scope discipline, c is still defined in C, whereas we do not need it anymore. We can see here an inconsistency between the scope of c, which is $A; B; C$, and its expected area of use, $A; B$.

We think that the scope-free variable way to do it is cleaner:

$$\{con \Rightarrow newc\}.\downarrow; A; \{scr \Rightarrow news\}.\downarrow; B;$$
$$\lambda\{con \Rightarrow c\}.\downarrow; C; \lambda\{scr \Rightarrow s\}.\downarrow$$

We didn't define any new variable, but did just temporarily hide the original value by the redirected one. And this would not be possible with dynamic binding alone, because of the scoping discipline.

6.3 Syntax of transformation calculus

In this section we define the untyped transformation calculus, and the selective λ-calculus as a subsystem of it.

The definition is done in two steps. 1) We give a syntactic definition of terms in the transformation calculus, and add a structural equivalence on these terms[2]. 2) Then we define reduction rules for these equivalence classes.

[2]We could use all equivalences as directed reduction rules. This would result in a slightly more complicated system (*cf.* Chap. 2 for selective λ-calculus)

Notations In the following definitions we will use the abbreviations $A \not\frown B$ for $A \cap B = \emptyset$, $FV(M)$ for the free variables of M, and $V(R)$ for the values contained in the stream R.

Definition 6.1 (terms) *Terms of the transformation calculus, or Λ_T, are those generated by M in the grammar of figure 6.1, where variables should be distinct in abstractions, and labels distinct in streams. Composition has lower priority than dots.*

They are considered modulo $\equiv$, the minimal equivalence relation defined by the closure of the equalities in figure 6.2.

Equalities $\equiv_.$ and $\equiv_\lambda$ are derived from the monoidal structure. $\equiv_{.;}$, $\equiv_{\lambda;}$ and $\equiv_;$ are intuitive.

Equality $\equiv_{.\lambda}$ is the "symmetrical" of β-reduction. It comes from the need to close the equality

$$(R' \uplus S').\lambda(R \uplus S).N \equiv_. \psi_R(S').R'.\lambda S.\lambda\psi_S(R).N,$$

with $\mathcal{D}_{R'} = \mathcal{D}_R, \mathcal{D}_{S'} = \mathcal{D}_S$ and $V(S) \not\frown V(R)$. If we take $M = \lambda\phi_S^{-1}(R).N$, and apply $R'.\lambda S.M$ to $\psi_R(S')$, then $\equiv_{.\lambda}$ preserves confluence: it gives

$$(R' \uplus S').\lambda(R \uplus S).N \equiv \psi_R(S').\lambda\psi_R(S).\psi_S(R').\lambda\psi_S(R).N.$$

Substitutions are done in the same way as for lambda-calculus, composition not interacting with variable binding. Terms will always be considered modulo α-conversion. That is $\lambda\{l \Rightarrow x\}.M \equiv \lambda\{l \Rightarrow y\}.[y/x]M$ when $y \notin FV(M)$.

Definition 6.2 (reductions) *"$\rightarrow$" is defined on transformation calculus terms by β-reduction and $\downarrow$-elimination*[3]*.*

$$\begin{array}{lll} \{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M & \rightarrow_\beta & [N/x]M \\ \downarrow; M & \rightarrow_\downarrow & M \end{array}$$

$\stackrel{*}{\rightarrow}$ is the reflexive and transitive closure of $\rightarrow$.

Theorem 6.1 *Transformation calculus is confluent.*

The proof is given in the next section. Confluence of transformation calculus is obtained from selective λ-calculus through a translation into it.

Proposition 6.1 *Normal terms are generated by N in the following grammar, where streams and anti-streams may be empty.*

$$\begin{array}{lll} H & ::= & x \mid \downarrow \\ F & ::= & \{l \Rightarrow N, \ldots\}.x \mid F; \lambda\{l \Rightarrow x, \ldots\}.F \\ N & ::= & \lambda\{l \Rightarrow x, \ldots\}.\{l \Rightarrow N, \ldots\}.H \mid \lambda\{l \Rightarrow x, \ldots\}.(F; N) \end{array}$$

This formalizes our intuition that, in a normal form, a composition only subsists when its left side cannot be reduced to a transformation.

[3]We chose to make $\downarrow$-elimination a reduction rule rather than a structural equality because it reduces the size of terms, while the structural equalities of Definition 6.1 do not change it.

6.4 Applicative translation and confluence

The idea of applicative continuation semantics was developed in [JD88] for a framework of stores and partial continuations. A partial continuation is a function from a state to a new state, but we can translate it into a function prefixing a continuation, that is a function from a continuation to a new continuation doing intended operations before calling the old continuation. In fact transformations look very much like partial continuations on streams, and this idea provides us with a translation from transformation calculus to selective λ-calculus.

Definition 6.3 (applicative translation) *Tr is the applicative translation from $\Lambda_T(\mathcal{L}_s)$ (without the associativity of composition $(M_1;M_2);M_3 \equiv M_1;(M_2;M_3)$) to $\Lambda_S(\mathcal{L}_s \cup \{cont\})$ (selective λ-calculus with names in $\mathcal{L}_s \cup \{cont\}$).*

$$\begin{array}{rcl} Tr(\downarrow) & = & \lambda\{cont \Rightarrow x\}.x \\ Tr(M;N) & = & \{cont \Rightarrow Tr(N)\}.M \\ Tr(x) & = & x \\ Tr(\lambda R.M) & = & \lambda R.Tr(M) \\ Tr(R.M) & = & Tr(R).Tr(M) \end{array}$$

The well-definedness of this translation is proved in Lemma 6.1.
The image of Λ_T is $Tr(\Lambda_T) = \Lambda_S^*$.

"*cont*" in the above definition stands for continuation. We translate composition into an application of its right-hand side to its left-hand side, seen as a continuation. Since *cont* is a new name, the continuation is received by the $\lambda\{cont \Rightarrow x\}.x$ head of the right-hand side, and operational semantics of the calculus are preserved.

We will use this translation to prove the confluence of transformation calculus, based on that of selective λ-calculus.

Lemma 6.1 *Tr is coherent w.r.t. $\equiv$ (without associativity of composition), and reduction paths coincide. Tr is a bijection from Λ_T to ${\Lambda_S}^*$.*

PROOF For coherence, we just verify that all equivalences in Λ_T are mapped to equivalences in Λ_S. For $\equiv_.$, $\equiv_\lambda$ and $\equiv_{.\lambda}$ this is immediate, since the same equivalence applies in the translation. $\equiv_{.;}$ and $\equiv_{\lambda;}$ are mapped respectively to $\equiv_.$ and $\equiv_{.\lambda}$:

$$\begin{array}{rcl} Tr((R.M_1);M_2) & = & \{cont \Rightarrow Tr(M_2)\}.Tr(R).Tr(M_1) \\ & \equiv & Tr(R).\{cont \Rightarrow Tr(M_2)\}.Tr(M_1) \\ & = & Tr(R.(M_1;M_2)) \end{array}$$

$$\begin{array}{rcl} Tr((\lambda R.M_1);M_2) & = & \{cont \Rightarrow Tr(M_2)\}.\lambda R.Tr(M_1) \\ & \equiv & \lambda R.\{cont \Rightarrow Tr(M_2)\}.Tr(M_1) \\ & = & Tr(\lambda R.(M_1;M_2)) \end{array}$$

For reduction steps, both $\rightarrow_\beta$ and $\rightarrow_\downarrow$ are mapped to β-reduction:

$$Tr(\downarrow;M) = \{cont \Rightarrow Tr(M)\}.\lambda\{cont \Rightarrow x\}.x$$

We define Tr^{-1} by reversing each case in the definition of T_r. It is well-defined on raw terms of ${\Lambda_S}^*$, and coherent by reversing cases above. This makes Tr a bijection. □

The above lemma lets us translate reduction of the transformation calculus into selective λ-calculus ones. We need another lemma to get reductions starting in ${\Lambda_S}^*$ back into Λ_T.

Lemma 6.2 *If* $M \in {\Lambda_S}^*$ *and* $M \to N$ *then* $N \in {\Lambda_S}^*$.

PROOF Terms of ${\Lambda_S}^*$ are characterized by the fact all abstractions using $cont$ only appear in the form $\lambda\{cont \Rightarrow x\}.x$.

If the reduction step is not on $cont1$, then it do not create any abstraction on $cont$, nor modify $\lambda\{cont \Rightarrow x\}.x$'s.

If this is on $cont1$, then the $\lambda\{cont \Rightarrow x\}.x$ concerned disappears.

In the two cases, the resulting term is still in ${\Lambda_S}^*$. □

Theorem 6.1 *Transformation calculus is confluent.*

$$\begin{aligned}\forall M,P,Q \in \Lambda_T \quad & (M \xrightarrow{*} P \wedge M \xrightarrow{*} Q) \\ & \Rightarrow (\exists T \in \Lambda_T \; P \xrightarrow{*} T \wedge Q \xrightarrow{*} T)\end{aligned}$$

PROOF By Lemma 6.1, we get $Tr(M) \xrightarrow{*} Tr(P)$ and $Tr(M) \xrightarrow{*} Tr(Q)$. By confluence of selective λ-calculus, we have T' such that $Tr(P) \xrightarrow{*} T'$ and $Tr(Q) \xrightarrow{*} T'$. But by Lemma 6.2 these reductions are in ${\Lambda_S}^*$, so that $T = Tr^{-1}(T')$ is a solution. □

6.5 Scope-free variable encoding

We cannot expect to give a precise definition of *scope-free variable* in the transformation calculus, where it is only encoded. It appears as an intuitive notion of a variable whose locality is not syntactical but operational. We will define it outside of the calculus.

For this we use a framework in which a program is a sequence of operations. Operations can themselve contain programs, but these are independent, and may not have side-effects on the external sequence.

Definition 6.4 (scope-free variable) *A* scope-free variable *is some way to create, modify and destroy a value such that:*

1. *these operations may appear in different syntactic entities, which may be used independently.*
2. *a* closed use *of this variable is obtained when a creator, some modifiers, and a destructor result in a* modification sequence.
3. *its closed use in a modification sequence has no side-effect outside it.*

A consequence of this definition is the hiding property we insisted on. The same variable may have several independent closed uses included in the same whose modification sequences intersect, and there is no problem as long as we do not try to modify the value from one use inside another's modification sequence.

As we introduced in Section 6.2, elementary creators, modifiers and destructors in transformation calculus are respectively $\{v \Rightarrow M\}.\downarrow$, $\lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}.\downarrow$ and $\lambda\{v \Rightarrow x\}.\downarrow$. But we can think of more complex ones, acting simultaneously on multiple variables, taking arguments, or returning results. For instance, in

$$\begin{aligned}&\lambda\{1 \Rightarrow x\}.\{a \Rightarrow 2, b \Rightarrow x\}.\downarrow; \\ &\lambda\{a \Rightarrow x, b \Rightarrow y\}.\{a \Rightarrow x \times y, b \Rightarrow x - y\}.\downarrow; \\ &\lambda\{a \Rightarrow x, b \Rightarrow y\}.(x + y)\end{aligned}$$

a and b are two scope-free variables, but their creator, modifier and destructor are joint.

To ensure that we have a correct scope-free variable encoding here, we must verify the third point of the definition, which says that it has no effect outside the sequence it is used in.

Proposition 6.2 *The scope-free variable encoding into the transformation calculus ensures locality to the modification sequence.*

PROOF If our variable is encoded on a label whose name is unique, there is no problem since it does not modify indexes of other arguments and variables, and its label appears only between its creation and destruction.

Otherwise, labels with the same name may appear with a different index in the sequence. But since each time they meet an abstraction or an application on our variable their indexes go alternately up and down, and the number of abstraction is equal to the number of applications, their index is the same on each side of the sequence. Moreover modifiers do not modify the value they support, since index changes avoid overloading of the same label. □

As we have seen, thanks to this property, scope-free variables are not only more flexible than classical scoped variables, but can replace them in most of their uses. Particularly, in functional language they can replace "disciplined" references (which do not go out of their scope), without the need of a specific evaluation strategy. Their only limitation is that —in the transformation calculus— one cannot export them like references, since they are linked to an explicit name. However, this is a limitation of the label system we use, and not of scope-free variable in themselves: one can add a syntactical scope to scope-free variables —we actually do that in the next chapter, Section 7.5.1. The real point about them is that the use of a (now scoped) scope-free variable is not restricted by that syntactical scope[4] (which is only a problem of naming), like with the stack discipline, but by its *life area*, or modification sequence (its real semantic scope).

Another notion easily encodable in the transformation calculus is that of communicating pairs. In fact, it is identical to scope-free variables, except that we only use creators and destructors on it, and no modifiers. What we obtain then is highly reminiscent of communication calculi. The creator is a sender, and the destructor a receiver. However the communication is unidirectional: a transformation can only send a message to a transformation following it, and their order is fixed by this potential communication (since their labels conflict they cannot commute by $(\cdot\cdot)$). This can give a good abstraction for certain types of problems. For instance the following transformations are an example of analysis-solving-synthesis structure.

$$
\begin{array}{l}
(\lambda\{1 \Rightarrow x\}.\{p \Rightarrow extract_p(x), q \Rightarrow extract_q(x), r \Rightarrow extract_r(x)\}.\downarrow; \\
\lambda.\{p \Rightarrow x\}.\{p \Rightarrow solve_p(x)\}; \\
\lambda.\{q \Rightarrow x\}.\{q \Rightarrow solve_q(x)\}; \\
\lambda.\{r \Rightarrow x\}.\{r \Rightarrow solve_r(x)\}; \\
\lambda\{p \Rightarrow x, q \Rightarrow y, r \Rightarrow z\}.synthesis(x, y, z))
\end{array}
$$

We can remark here that as long as two transformations are not directly dependent by such a communicating pair (or equivalently, a shared scope-free variable) they can commute. It is evident in this example, but this would be true too with independent uses of the same symbol, with appropriate reindexing:

$$
\begin{array}{l}
\lambda\{p1 \Rightarrow x\}.\{q1 \Rightarrow M\}; \lambda\{p1 \Rightarrow x\}.\{p1 \Rightarrow N\} \\
= \lambda\{p2 \Rightarrow x\}.\{p2 \Rightarrow N\}; \lambda\{p1 \Rightarrow x\}.\{q1 \Rightarrow M\}
\end{array}
$$

While intrinsically transformation calculus is sequential, we may interpret these possibilities of commutation as potential concurrency.

[4]This is true with references too, but their semantic scope is only defined by garbage collection

6.6 Transformational combinators

Similarly to lambda-calculus, transformation calculus can be generated from a small set of combinators. They are used for the compilation of a language based on the transformation calculus, FIML (*cf.* Appendix B).

Since we have no abstraction in this system, we will use a slightly different notation: terms are elements of the free monoid $\mathcal{T}^*$ of streams and transformational combinators. Streams are defined by induction as $\mathcal{S} ::= \{l \Rightarrow \mathcal{T}^*, \ldots\}$; combinator are of form C^n, where n is the arity of the combinator, that is the number of labels parameterizing it (this number is independent of the way it reacts with streams it is juxtaposed with).

Transformation logic forms an equational theory, which includes stream concatenation: $\forall R, S \in \mathcal{S},\ P\ Q = (P \cdot Q)$, and the equation proper to each combinator.

6.6.1 First formulation : $A^1U^2C^2D^2K^1$

A first set uses five combinators.

$A_p\{p \Rightarrow M\}$	$=$	M	projection
$U_{pq}\{p \Rightarrow M\}$	$=$	$\{p \Rightarrow \{q \Rightarrow M\}\}$	up
$C_{pq}\{p \Rightarrow M, q \Rightarrow N\}$	$=$	$\{p \Rightarrow MN\}$	composition
$D_{pq}\{p \Rightarrow M\}$	$=$	$\{p \Rightarrow M, q \Rightarrow M\}$	duplication
$K_p\{p \Rightarrow M\}$	$=$	$\{\}$	killer

From these we can deduce the two following combinators:

L_{pq}	$=$	A_pU_{pq}	relabeling	$\{p \Rightarrow M\} \mapsto \{q \Rightarrow M\}$
Λ_{px}	$=$	$C_{p\lambda}U_{\lambda x}D_{x\lambda}$	broadcast	$\{p \Rightarrow M, x \Rightarrow N\} \mapsto \{p \Rightarrow M\{x \Rightarrow N\}, x \Rightarrow N\}$

In Λ, we suppose that p, x and $\lambda$are different symbolic labels.

This is enough to encode the full transformation calculus. We define our translation T as $T(M) = T[FV(M)](M)$.

$T[x](x)$	$=$	A_x
$T[x : E](M)$	$=$	$T[E](M)\ K_x$ if $x \notin FV(M)$
$T[E](\lambda\{p \Rightarrow x\}.M)$	$=$	$T[x : E](M)\ L_{px}$
$T[E](\{p \Rightarrow N\}.M)$	$=$	$T[E](M)\ \Lambda_{px_1} \ldots \Lambda_{px_n}\ \{p \Rightarrow T(N)\}$
		$FV(N) = \{x_1, \ldots, x_n\} \subset E$
$T[FV(MN)](N; M)$	$=$	$T(M)\ T(N)\ D_{x_1x_1+1} \ldots D_{x_nx_n+1}$
		$FV(M) \cap FV(N) = \{x_1, \ldots, x_n\} \subset E$

6.6.2 Second formulation : $A^1K^1U^2S^2$

In fact, it is possible to reduce the number of combinators to four.

This time we define our system as a grammar (no deep reason here, just another presentation).

$$M ::= MM \mid A_l \mid K_l \mid U_{ll} \mid S_{ll} \mid \{l \Rightarrow M, \ldots\}$$

And we have the following equations.

$M(NP)$	$=$	$(MN)P$
$A_p\{p \Rightarrow M\}$	$=$	M
$K_p\{p \Rightarrow M\}$	$=$	$\{\}$
$U_{pq}\{p \Rightarrow M\}$	$=$	$\{p \Rightarrow \{q \Rightarrow M\}\}$
$S_{pq}\{p \Rightarrow M, q \Rightarrow N\}$	$=$	$\{p \Rightarrow MN, q \Rightarrow M\}$

The two combinators C^2 and D^2 become deduced combinators.

$$\begin{aligned} C_{pq} &= K_q S_{pq} \\ D_{pq} &= S_{pq}\{q \Rightarrow \{\}\} \\ L_{pq} &= A_p U_{pq} \\ \Lambda_{px} &= C_{p\lambda} U_{\lambda x} D_{x\lambda} \end{aligned}$$

These definition suppose $p \neq q$.
We can translate similarly.

Chapter 7

Typed transformation caculus and extensions

In this this chapter we first study the typing of transformation calculus, giving two typing systems, a simple one and a polymorphic one. For the simple one, we complete its description by constructing a mathematical model of the simply typed transformation calculus. For the polymorphic one, we give a type reconstruction algorithm, incomplete since principal types do not always exist, but efficient enough to limit type annotations to a minimum.

We then extend transformation calculus with name scoping, extending simple typings together, and use it to demonstrate how an object-oriented style fits well in the framework of a typed transformation calculus.

7.1 Simply typed transformation calculus

To obtain a simply typed form of transformation calculus, we annotate variables with some type in abstractions, just the same way it is done in lambda calculus. But first we must define what are these types.

The two most important novelties are that, first, stream types are introduced, and second, that function type are not from any type to any other, but only from stream types to stream or base types. This last particularity "flattens" types, but still contains as a subset all simple types of lambda-calculus.

Definition 7.1 (simple type) *Simple types in the transformation calculus are generated by t in the following grammar.*

u	::=	$u_1 \mid \ldots$	*base types*
r	::=	$\{l \Rightarrow t, \ldots\}$	*stream types*
w	::=	$u \mid r$	*return types*
t	::=	$r \rightarrow w$	*types*

The same label may not appear more than once in the same stream type; stream types are equal up to different orders, and $(\{\} \rightarrow w) = w$, to shorten.

These last restrictions make a stream type a stream of types as defined in Chapter 3. This means that we can use stream composition on these types, as we will do for typing rules.

Definition 7.2 (simply typed term) *A term in the simply typed transformation calculus is constructed according to the following syntax.*

$$M \ ::= \ x \mid \downarrow \mid \lambda\{l \Rightarrow x{:}t, \ldots\}.M \mid \{l \Rightarrow M, \ldots\}.M \mid M; M$$

with the same constraints on labels and variables as before.

$$\Gamma[x \mapsto \tau] \vdash x : \tau \quad \text{(I)}$$

$$\frac{\Gamma[x \mapsto \theta] \vdash M : r \to w}{\Gamma \vdash \lambda\{l \Rightarrow x{:}\theta\}.M : (\{l \Rightarrow \theta\} \cdot r) \to w} \quad \text{(II)}$$

$$\frac{\Gamma \vdash M : (\{l \Rightarrow \theta\} \cdot r) \to w \qquad \Gamma \vdash N : \theta}{\Gamma \vdash \{l \Rightarrow N\}.M : r \to w} \quad \text{(III)}$$

$$\Gamma \vdash \downarrow : \{\} \to \{\} \quad \text{(IV)}$$

$$\frac{\Gamma \vdash M : r_1 \to r_2 \qquad \Gamma \vdash N : r_2 \to w}{\Gamma \vdash M; N : r_1 \to w} \quad \text{(V)}$$

$$\frac{\Gamma \vdash M : r_1 \to r_2}{\Gamma \vdash M : (r_1 \cdot r) \to (r_2 \cdot r)} \quad \text{(VI)}$$

Figure 7.1: Typing rules for simply typed transformation calculus

Finally the relation between terms and types is given in the following definition.

Definition 7.3 (type judgement) *A* type judgement, *written* $\Gamma \vdash M : \tau$, *expresses that the term* M *has type* τ *in the context* Γ. *Induction rules for type judgements are given in figure 7.1.*

Rules (I,II,III) are the traditional ones for typed lambda calculus, simply extended to streams. We can go back to it by limiting labels in streams to sequences of integers starting from 1 (that is, in the above rules, having only $l = \epsilon 1$).

Rule (IV) types the constant $\downarrow$. However it will most often need the cooperation of rule (VI), transformation subtyping, which expresses that any transformation may be applied to labels it is not concerned with: they will simply be rejected to the result. For instance, it gives to $\downarrow$ any symmetrical type ($r \to r$). Rule (V) types composition: M is applied to the result stream of N, and re-abstracted by its abstraction part. Here again, we need the collaboration of rule (VI) to extend the types of either M or N.

Proposition 7.1 (subject reduction) *If* $\Gamma \vdash M : \tau$, *and* $M \to N$ *or* $M \equiv N$, *then* $\Gamma \vdash N : \tau$.

PROOF We start from a proof $\Gamma \vdash M : \tau$, and construct a proof $\Gamma \vdash N : \tau$. We can suppose that the redex concerned is external in M (the proof for the context does not change). We limit the use of rule (VI) in the original proof to variables and $\downarrow$ — this only amounts to pushing its applications up in the proof tree, and avoids the case where this rule is used between two others.

If $M = \{l \Rightarrow M_1\}.\lambda\{l \Rightarrow x{:}\theta\}.M_0$, Π_0 a proof of $\Gamma[x \mapsto \theta] \vdash M_0 : \tau$ and Π_1 a proof of $\Gamma \vdash M_1 : \theta$, then Π_0 where $\Gamma[x \mapsto \theta] \vdash x : \theta$ is replaced by Π_1 is a proof of $\Gamma \vdash [M_1/x]M_0 = N : \tau$.

If $M = \downarrow; N$, then we have already the right proof for N.

We proceed similarly for cases where $M \equiv N$. □

Proposition 7.2 (strong normalization) *If* $\Gamma \vdash M : \tau$ *in the simply typed transformation calculus, then there is no infinite reduction sequence starting from* M.

PROOF We base ourselves on a translation into a simply typed λ-calculus with streams (we use matching rather than projections). This calculus, clearly equivalent to simply typed λ-calculus with pairing, satifies both suject reduction and strong normalization.

We translate a proof Π of $\Gamma \vdash M : \tau$ into a proof Π' of $\Gamma \vdash M' : \tau$ in simply typed λ-calculus with streams, where types are identical to those of the simply typed transformation calculus. This translation is very similar to that used for the simply typed model presented next.

Rules get translated into (I does not change):

$$\frac{\Gamma[x \mapsto \theta] \vdash M' : r \to w}{\Gamma \vdash \lambda^*\{k \Rightarrow y_k\}_{k \in \{l\} \cdot \mathcal{D}_r} : \{l \Rightarrow \theta\} \cdot r . (\lambda x{:}\theta . M' \ \{k \Rightarrow y_{\phi_{\{l\}}(k)}\}_{k \in \mathcal{D}_r}) \ y_l : \{l \Rightarrow \theta\} \cdot r \to w} \quad \text{(II')}$$

$$\frac{\Gamma \vdash M' : \{l \Rightarrow \theta\} \cdot r \to w \qquad \Gamma \vdash N' : \theta}{\Gamma \vdash \lambda^* y{:}r . M(\{l \Rightarrow N\} \cdot y) : r \to w} \quad \text{(III')}$$

$$\Gamma \vdash \lambda y{:}\{\} . y : \{\} \to \{\} \quad \text{(IV')}$$

$$\frac{\Gamma \vdash M' : r_1 \to r_2 \qquad \Gamma \vdash N : r_2 \to w}{\Gamma \vdash \lambda^* y{:}r_1 . N'(M' \ y) : r_1 \to w} \quad \text{(V')}$$

$$\frac{\Gamma \vdash M' : r_1 \to r_2}{\Gamma \vdash \lambda^*\{l \Rightarrow y_l\}_{l \in \mathcal{D}_{r_1 \cdot r}} : r_1 \cdot r . ((M'\{l \Rightarrow y_l\}_{l \in \mathcal{D}_{r_1}}) \cdot \{l \Rightarrow y_{\phi_{r_1}(l)}\}_{l \in \mathcal{D}_r}) : r_1 \cdot r \to r_2 \cdot r} \quad \text{(VI')}$$

We sketch the rest of the proof.

We have marked some abstractions with * in this translation. This means that they have only a structural role. We will just ignore them, and write $\overline{M'}$ for M' where all $*$-marked redexes were reduced (thanks to strong normalization). All other redexes are kept, since $*$-marked abstractions are linear, and their reduction does not change inclusion relations between redexes. This gives us unicity of $\overline{M'}$. Moreover, in $\overline{M'}$ all potential redexes of M (using $\equiv$) appear. Particularly, associativity of transformation composition maps to associativity of λ-function composition.

One can verify that if we have $\Gamma \vdash M \to N : \tau$ in the simply typed transformation calculus, then, for M' and N' translations of M and N (the proof of $\Gamma \vdash N : \tau$ chosen to be similar to that of $\Gamma \vdash M : \tau$), we have $\Gamma \vdash \overline{M'} \to^* \overline{N'} : \tau$ in the simply typed λ-calculus with streams (we may need more steps than in the orginal, because of marked redexes).

As a result, strong normalization extends to simply typed transformation calculus. □

This last property is interesting, since it is general belief that introducing mutables suppresses strong normalization: we keep it here, because all values used by a term appear in its type.

Such a type system is not polymorphic, but it is more generic than what we would have obtained by the translation towards selective λ-calculus. In this translation, composition $T; M$ is interpreted as the passing of a continuation to the transformation, $\{cont \Rightarrow M\}.T$, which means that when typing T we fix the type. Thanks to the rule (VI), this is not the case here: the continuation M must only be able to accept all the output of T, but its result has no link with T's type.

For instance, suppose we have a language modelling the transformation calculus, with basic arithmetic operation. We define a transformation on a pair of values add_sub:

$$\lambda\{x{:}int, y{:}int\}.\{x + y, x - y\}.\downarrow : \{1 \Rightarrow int, 2 \Rightarrow int\} \to \{1 \Rightarrow int, 2 \Rightarrow int\}$$

Now we can compose it with $mult = \cdot\lambda\{x{:}int, y{:}int\}.(x * y)$ and obtain

$$\lambda\{x{:}int, y{:}int\}.((x + y) * (x - y)) : \{1 \Rightarrow int, 2 \Rightarrow int\} \to int$$

or with itself for

$$\lambda\{x{:}int, y{:}int\}.\{x + x, y + y\}.\downarrow : \{1 \Rightarrow int, 2 \Rightarrow int\} \to \{1 \Rightarrow int, 2 \Rightarrow int\}$$

Application can be done on related labels (with the good type), or unrelated ones, the type being free then:

$$\{2 \Rightarrow 5, ok \Rightarrow true\}.add_sub = \lambda\{1 \Rightarrow x{:}int\}.\{1 \Rightarrow x+5, 2 \Rightarrow x-5, ok \Rightarrow true\}.\downarrow \\ :\{1 \Rightarrow int\} \rightarrow \{1 \Rightarrow int, 2 \Rightarrow int, ok\,1 \Rightarrow bool\}$$

7.2 Denotational semantics

In this section we give a model of the transformation calculus. However, to avoid technical problems specific to untyped models, and to get simple semantics, we base ourselves on the simply typed version of the calculus.

Definition 7.4 (model) *A model of the transformation calculus is a pair $(\mathcal{A}, [\![_]\!]_)$ with $\mathcal{A}$ a set of values, and $(M,\rho) \mapsto [\![M]\!]_\rho : \Lambda_T \times \mathcal{A}^{\mathcal{V}} \rightarrow \mathcal{A}$ a translation from a simply typed term M and an environment ρ ($FV(M) \subset \mathcal{D}_\rho$) into our model satisfying the axioms*

$$\begin{array}{rcl} M \equiv N & \Rightarrow & [\![M]\!]_\rho = [\![N]\!]_\rho, \\ M \rightarrow N & \Rightarrow & [\![M]\!]_\rho = [\![N]\!]_\rho, \\ \rho(x) = a & \Rightarrow & [\![x]\!]_\rho = a. \end{array}$$

We define our model $\mathcal{A} = \bigcup_{\tau \in T} \mathcal{A}^\tau$ by closure of the following procedure.

1. For $u \in T_0$ (base type), $\mathcal{A}^u$ is given. $\mathcal{A}_0 = \bigcup_{u \in T_0} \mathcal{A}^u$.
2. Streams values of level n are in $\mathcal{S}_n = \bigcup_{r \in \mathcal{S}(T_n)} \mathcal{A}^r$, where
$$\mathcal{A}^{\{l_i \Rightarrow \tau_i\}_{i=1}^m} = \bigcup_{a_1 \in \mathcal{A}^{\tau_1}} \cdots \bigcup_{a_m \in \mathcal{A}^{\tau_m}} \{l_1 \Rightarrow a_1, \ldots, l_m \Rightarrow a_m\}$$
3. Types of level $n+1$ are defined by
$$T_{n+1} = T_n \cup \{r \rightarrow w \mid r \in \mathcal{S}(T_n), w \in \mathcal{S}(T_n) \cup T_0\}$$
4. Values of level $n+1$ are defined by $\mathcal{A}_{n+1} = \bigcup_{\tau \in T_{n+1}} \mathcal{A}^\tau$ where
$$\mathcal{A}^{r \rightarrow w} = \mathcal{A}^r \rightarrow \mathcal{A}^w$$
5. $\mathcal{A} = \lim_{n \rightarrow \infty} \mathcal{A}_n$

$\mathcal{A}$ is well-defined, since for any τ there exists n such that $\tau \in T_n$ and then $\mathcal{A}^\tau \subset \mathcal{A}_n$.

Note that, like we did with types, we are identifying the streams of $\mathcal{A}^r$ with the functions of $\mathcal{A}^{\{\} \rightarrow r} = \mathcal{A}^{\{\}} \rightarrow \mathcal{A}^r$.

Once we have defined the values of the model, we must define operations on them. Out of concatenation, already defined on streams, we have two: *extension* and *composition*.

Extension is the operation by which a value in $\mathcal{A}^{r_1 \rightarrow r_2}$ gets canonically extended into a value of $\mathcal{A}^{(r_1 \cdot r) \rightarrow (r_2 \cdot r)}$. If f is in $\mathcal{A}^{r_1 \rightarrow r_2}$ then $f * r$, the r-extension of f is defined as:

$$\begin{array}{rlcl} f * r : & \mathcal{A}^{r_1 \cdot r} & \rightarrow & \mathcal{A}^{r_2 \cdot r} \\ & \{l \Rightarrow x_l\}_{l \in \mathcal{D}_{r_1 \cdot r}} & \mapsto & (f\{l \Rightarrow x_l\}_{l \in \mathcal{D}_{r_1}}) \cdot (\{l \Rightarrow x_{\phi_{r_1}(l)}\}_{l \in \mathcal{D}_r}) \end{array}$$

Composition is just the mathematical one; if f is in $\mathcal{A}^{r_1 \rightarrow r_2}$ and g in $\mathcal{A}^{(r_2 \cdot r) \rightarrow w}$, then $f;g$ is defined as:

$$\begin{array}{rlcl} f;g : & \mathcal{A}^{r_1} & \rightarrow & \mathcal{A}^w \\ & s & \mapsto & g(f(s)) \end{array}$$

$$
\begin{array}{lcl}
[\![x]\!]_\rho & = & \rho(x) \\
[\![\downarrow]\!]_\rho & = & \{\} \\
[\![\lambda\{l_0 \Rightarrow x{:}\tau\}.M]\!]_\rho & = & (\text{when } [\![M]\!]_{\rho[a^\tau/x]} \in \mathcal{A}^{r \to w}) \\
\quad \mathcal{A}^{\{l_0 \Rightarrow \tau\} \cdot r} & \to & w \\
\quad \{l \Rightarrow x_l\}_{l \in \{l_0\} \cup \mathcal{D}_r} & \mapsto & [\![M]\!]_{\rho[x_{l_0}/x]}\{l \Rightarrow x_l\}_{l \in \mathcal{D}_r} \\
[\![\{l \Rightarrow N\}.M]\!]_\rho & = & \{l \Rightarrow [\![N]\!]_\rho\}; [\![M]\!]_\rho \\
[\![M; N]\!]_\rho & = & [\![M]\!]_\rho; [\![N]\!]_\rho
\end{array}
$$

Figure 7.2: Semantic function of the simply typed transformation calculus

However the above definitions will not work as model: with the subtyping introduced on transformations in the calculus, we expect that the same value may actually be contained in several $\mathcal{A}^\tau$ sets. That is why we will consider the above definition modulo extension.

More precisely, we introduce the following equivalence $f =_* f * r$, closed by symmetry and transitivity (it is already reflexive). Since extension only applies on members of $\mathcal{A}^{r_1 \to r_2}$, equivalent terms are transformations. Moreover we remark that, modulo this equivalence, $\mathcal{A}^{s_1 \to s_2}$ now includes all $\mathcal{A}^{r_1 \to r_2}$ such that for some r, $r_1 \cdot r = s_1$ and $r_2 \cdot r = s_2$.

We define $\mathcal{A}_*$ as $\mathcal{A}_{/=_*}$, and $\mathcal{A}^\tau_*$ as the sets of all classes containing an element of $\mathcal{A}^\tau$.

We show easily that composition is coherent with this equivalence: if $f \in \mathcal{A}^{r_1 \to r_2}$ and $g \in \mathcal{A}^{r_2 \to r_3}$ (for extension to be possible, both f and g must be transformations), then $f * r \in \mathcal{A}^{(r_1 \cdot r) \to (r_2 \cdot r)}$, $g * r \in \mathcal{A}^{(r_2 \cdot r) \to (r_3 \cdot r)}$, and

$$
\begin{array}{l}
(f * r; g * r)\{l \Rightarrow x_l\}_{l \in \mathcal{D}_{r_1 \cdot r}} \\
= (g * r)((f\{l \Rightarrow x_l\}_{l \in \mathcal{D}_{r_1}}) \cdot (\{l \Rightarrow x_{\phi_{r_1}(l)}\}_{l \in \mathcal{D}_r})) \\
= (g(f\{l \Rightarrow x_l\}_{l \in \mathcal{D}_{r_1}})) \cdot (\{l \Rightarrow x_{\phi_{r_1}(l)}\}_{l \in \mathcal{D}_r})) \\
= ((f; g) * r)\{l \Rightarrow x_l\}_{l \in \mathcal{D}_{r_1 \cdot r}}
\end{array}
$$

Finally we define the translation $(M, \rho) \mapsto [\![M]\!]_\rho : \Lambda_T \times \mathcal{A}_*^{\mathcal{V}} \to \mathcal{A}_*$, from a simply typed term M and an environment ρ ($FV(M) \subset \mathcal{D}_\rho$) into our model[1] in figure 7.2.

Proposition 7.3 *$(\mathcal{A}_*, [\![_]\!]__)$ is a model of the simply typed transformation calculus.*

PROOF We must prove that our three axioms are verified.

- $\equiv$: the proof is direct for each basic equivalence.
- $\to_\downarrow$: just notice that all extensions of $[\![\downarrow]\!]$ are the identity.
- $\to_\beta$: $[\![\{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M]\!]_\rho = [\![M]\!]_{\rho[[\![N]\!]_\rho/x]} = [\![[N/x]M]\!]_\rho$.

□

7.3 Polymorphic types

Like for the label-selective λ-calculus, we define here a polymorphically typed version of the transformation calculus. Together with it, we give a type reconstruction algorithm.

[1] Since terms are statically typed, we could construct a model based on tuples and type information, which would avoid the extra structure for streams, and compile away labels. Only extension would have to be modified. We based ourselves on streams here for the sake of simplicity of the equivalence relation.

However, unlike selective λ-calculus, this algorithm is very incomplete. For a very simple reason: there are cases where no principal type exists. Still, type reconstruction lets one reduce the amount of typing information necessary in a term, and opens the way to transformation calculus based strongly typed programming languages.

7.3.1 Syntax and types

The polymorphism by subtyping we had in the simply typed calculus becomes a problem when we try to polymorphically type transformation calculus expressions. Suppose for instance that we want to define composition as a function $comp\ \{f \Rightarrow f, g \Rightarrow g\} = f;g$. If we have only generic type variables, we cannot express the relation which is between the output stream of g and the input stream of f.

The intuitive solution to this problem is the introduction of stream variables, for stream types. It is indeed quite expressive. Suppose $comp$ defined above to be typed $\{f \Rightarrow (\rho \rightarrow \rho'), g \Rightarrow (\rho' \rightarrow \alpha)\} \cdot \rho \rightarrow \alpha$. It correctly expresses the constraint, even permitting to receive arguments on labels other than $f1$ and $g1$, linking their types with types in f's input.

However it will appear not to be enough. The second problem comes from rule (VI) of typing, which expresses than any transformation has an infinity of types. But, when we write $\rho \rightarrow \rho'$ we are interested in only one of these types, whose ρ' is the same as in $\rho' \rightarrow \alpha$, which may be a simple function. In this case the maximal possible ρ' is the total input of g. If we let α be any type, it may contain part of the input and reduce the genericity of g. This is why we need a last sort of type variables, we will call them return type variables, and they are restricted to represent return types.

Definition 7.5 (polymorphic type) *We have a set u of base types, and another set α of variable names.*

$*$	$::=$	$\bullet \mid \diamond$	*return/stream sort*
$\tau^\diamond$	$::=$	$\{l \Rightarrow \tau, \ldots\}$	*closed stream*
	$\mid$	$\{l \Rightarrow \tau, \ldots\} \cdot \alpha^\diamond$	*open stream*
$\tau^\bullet$	$::=$	$\alpha^\bullet \mid u \mid \tau^\diamond$	*return types*
τ	$::=$	$\tau^\diamond \rightarrow \tau^\bullet$	*monotypes*
σ	$::=$	$\tau \mid \forall\alpha.\sigma$	*polytypes*

There is a subtype relation between sort, that is $\diamond < \bullet$.

By $\diamond < \bullet$, we mean that during type instantiation an unsorted variable may be substituted with any type, but a return ($\bullet$) variable is restricted to return types, and a stream ($\diamond$) one to stream types.

Notations To simplify the writing, and help the intuition, we will use the following abbreviations for variables:

$$\alpha = \alpha^\diamond \rightarrow \alpha^\bullet \qquad \forall\alpha.\sigma = \forall\alpha^\diamond.\forall\alpha^\bullet.\sigma \qquad \sigma[\tau/\alpha] = \sigma[r/\alpha^\diamond, w/\alpha^\bullet]$$

Moreover we identify $\alpha^\diamond$ and $\{\} \cdot \alpha^\diamond$.

Patterns in the rules use the corresponding non-terminal, but we use both θ and τ for τ. Streams are normalized after substitution.

We give a type inference algorithm in subsection 7.3.4. The inference is complex, and incomplete in the general case, since there are cases where there is no principal type. This is why we allow adding some type information to terms, which permits to guide the inference towards a solution type.

Definition 7.6 (constrained term) *Terms are those of transformation calculus, extended to allow type specification.*

$$M ::= x \mid \{\} \mid \lambda\{l \Rightarrow x : \tau, \ldots\}.M \mid M\{l \Rightarrow M, \ldots\} \mid M; M$$

We use $\{l \Rightarrow x\}$ as an abbreviation for $\{l \Rightarrow x : \alpha\}$, which introduces no constraint on the type of x.

In this system composition will be typed:

$$comp : \{f \Rightarrow (\alpha^\diamond \rightarrow \beta^\bullet), g \Rightarrow (\gamma^\diamond \rightarrow \alpha^\diamond)\} \cdot \gamma^\diamond \rightarrow \beta^\bullet.$$

Most functions can be typed in this formalism. There are still exceptions. The simplest one is auto-composition: $\lambda(x).(x;x)$. The only polymorphic type we can give it is $\{1 \Rightarrow (\alpha^\diamond \rightarrow \alpha^\diamond)\} \cdot \alpha^\diamond \rightarrow \alpha^\diamond$. If our intention was to use it as stream duplication ($\{p \Rightarrow a, q \Rightarrow b\} \mapsto \{p1 \Rightarrow a, p2 \Rightarrow a, q1 \Rightarrow b, q2 \Rightarrow b\}$), this will not work.

7.3.2 Typing rules

To preserve polymorphism in streams, we use here a special form of polymorphism, *applicative polymorphism*. Its application to classical λ-calculus is described in appendix A.1.

The essential difference is in the structure of the assumption. Together with the typing environment Γ, we have an applicative context A, which contains the type of the stream our type is applied to.

$$\Gamma, \mathrm{A} \vdash M : \sigma$$

means that in an environment Γ, when M is applied to a streams whose elements have types in A, the result of the application has type σ.

Inference rules are in Figure 7.3. In Abs', $\theta \sqsubseteq \tau$ means that there is a substitution ς (respecting $\diamond < \bullet$), such that $\varsigma(\theta) = \tau$.

Proposition 7.4 (subject reduction) *If $\Gamma, A \vdash M : \sigma$ in the polymorphically typed transformation calculus, and $M \rightarrow N$ or $M \equiv N$, then $\Gamma, A \vdash N : \sigma$*

PROOF Like for the simply typed version, we can suppose the concerned redex to be the most external in M.

We distinguish three categories of rules: axioms (Var, Empty), structural rules (App, Abs, Abs', Comp), and auxiliary rules (all others).

Axioms form always the top of proofs, but auxiliary rules may appear in their middle. To avoid problems when an auxiliary rule appears between two structural ones, we either push them up (Open, Charge, Inst, Inst') or down (Discharge, Gen).

If M is a β-redex, then the proof has form,

$$\dfrac{\dfrac{\Pi'}{\Gamma, \{\} \vdash N : \sigma} \quad \dfrac{\dfrac{\Pi}{\Gamma[x \mapsto \sigma], \mathrm{A} \vdash M : \sigma'}}{\Gamma, \{l \Rightarrow \sigma\} \cdot \mathrm{A} \vdash \lambda\{l \Rightarrow x\}.M : \sigma'}}{\Gamma, \mathrm{A} \vdash \{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M : \sigma'}$$

After β-reduction, Π where $\Gamma[x \mapsto \sigma], \{\} \vdash x : \sigma$ is replaced by Π' is a proof of

$$\Gamma, A \vdash [N/x]M : \sigma'$$

Similarly if we had to use Abs', since $\theta \sqsubseteq \tau$ is only a side condition.

Var $\Gamma[x:\sigma],\{\} \vdash x:\sigma$

Empty $\Gamma, \mathrm{A} \vdash \downarrow : \mathrm{A}$ A only monotypes

App $$\frac{\Gamma,\{\} \vdash N:\sigma \quad \Gamma,\{l \Rightarrow \sigma\}\cdot \mathrm{A} \vdash M:\sigma'}{\Gamma,\mathrm{A} \vdash \{l \Rightarrow N\}.M:\sigma'}$$

Abs $$\frac{\Gamma[x \mapsto \sigma],\mathrm{A} \vdash M:\sigma'}{\Gamma,\{l \Rightarrow \sigma\}\cdot \mathrm{A} \vdash \lambda\{l \Rightarrow x\}.M:\sigma'}$$

Abs' $$\frac{\Gamma[x \mapsto \tau],\mathrm{A} \vdash M:\sigma}{\Gamma,\{l \Rightarrow \tau\}\cdot \mathrm{A} \vdash \lambda\{l \Rightarrow x:\theta\}.M:\sigma}\ \theta \sqsubseteq \tau$$

Comp $$\frac{\Gamma,\mathrm{A} \vdash M:\tau^\diamond \rightarrow \{l_i \Rightarrow \theta_i\}_{i=1}^n \cdot \theta^\diamond \quad \Gamma,\{l_i \Rightarrow \forall \mathrm{B}_i.\theta_i\}_{i=1}^n \vdash N:\theta^\diamond \rightarrow \tau^\bullet}{\Gamma,\mathrm{A} \vdash M;N:\tau^\diamond \rightarrow \tau^\bullet}\ \mathrm{B}_i \not\in (Var(\Gamma) \cup Var(A) \cup Var(\tau^\diamond))$$

Open $$\frac{\Gamma,\mathrm{A} \vdash M:\{l_i \Rightarrow \theta_i\} \rightarrow \{l'_j \Rightarrow \theta'_j\}}{\Gamma,\mathrm{A} \vdash M:\{l_i \Rightarrow \theta_i\}\cdot\alpha^\diamond \rightarrow \{l'_j \Rightarrow \theta'_j\}\cdot\alpha^\diamond}$$

Discharge $$\frac{\Gamma,\mathrm{A}\cdot\{l \Rightarrow \theta\} \vdash M:\tau^\diamond \rightarrow \tau^\bullet}{\Gamma,\mathrm{A} \vdash M:\{l \Rightarrow \theta\}\cdot\tau^\diamond \rightarrow \tau^\bullet}$$

Charge $$\frac{\Gamma,\mathrm{A} \vdash M:\{l \Rightarrow \theta\}\cdot\tau^\diamond \rightarrow \tau^\bullet}{\Gamma,\mathrm{A}\cdot\{l \Rightarrow \theta\} \vdash M:\tau^\diamond \rightarrow \tau^\bullet}$$

Gen $$\frac{\Gamma,\mathrm{A} \vdash M:\sigma}{\Gamma,\mathrm{A} \vdash M:\forall\alpha^*.\sigma}\quad \alpha^* \notin Var(\Gamma) \cup Var(\mathrm{A})$$

Inst $$\frac{\Gamma,\mathrm{A} \vdash M:\forall\alpha^*.\sigma}{\Gamma,\mathrm{A} \vdash M:\sigma[\tau^*/\alpha^*]}$$

Inst' $$\frac{\Gamma,\{l \Rightarrow \sigma[\tau^*/\alpha^*]\}\cdot\mathrm{A} \vdash M:\sigma}{\Gamma,\{l \Rightarrow \forall\alpha^*.\sigma\}\cdot\mathrm{A} \vdash M:\sigma}$$

Figure 7.3: Inference rules for polymorphically typed transformation calculus

Base type	$\dfrac{\varphi, u = v \quad u \neq v}{\bot \qquad u, v \text{ base types}}$
Stream type	$\dfrac{\varphi, \{\} = \{l \Rightarrow \theta\} \cdot \tau^\diamond}{\bot}$
Sort mismatch	$\dfrac{\varphi, u = \tau^\diamond}{\bot}$
Variable recurrence	$\dfrac{\varphi, \alpha^* = \tau^* \quad \tau^* \neq \alpha^*}{\bot \qquad \alpha^* \in Var(\tau^*)}$
Variable elimination	$\dfrac{\varphi, \alpha^* = \tau^* \quad \alpha^* \in Var(\varphi) - Var(\tau^*)}{[\tau^*/\alpha^*]\varphi, \alpha^* = \tau^* \quad \text{if } \tau^* = \beta^* \text{ then } \beta^* \in Var(\varphi)}$
Redundancy	$\dfrac{\varphi, \theta = \theta}{\varphi}$
Splitting	$\dfrac{\varphi, r \to w = r' \to w'}{\varphi, r = r', w = w'} \quad r \neq \{\} \text{ or } r' \neq \{\}$
Decomposition	$\dfrac{\varphi, \{l \Rightarrow \theta\} \cdot r = \{l \Rightarrow \theta'\} \cdot r'}{\varphi, \theta = \theta', r = r'}$

Figure 7.4: Monotype unification algorithm

For $\downarrow$-elimination, the proof has form,

$$\dfrac{\dfrac{\dfrac{\Gamma, \{l_i \Rightarrow \theta_i\}_{i=1}^n \vdash \downarrow : \{l_i \Rightarrow \theta_i\}_{i=1}^n}{\Gamma, \{l_i \Rightarrow \forall A_i.\theta_i\}_{i=1}^n \vdash \downarrow : \{l_i \Rightarrow \theta_i\}_{i=1}^n}}{\Gamma, \{l_i \Rightarrow \forall A_i.\theta_i\}_{i=1}^n \vdash \downarrow : \tau^\diamond \to \{l_i \Rightarrow \theta_i\}_{i=1}^n \cdot \tau^\diamond} \qquad \dfrac{\text{II}}{\Gamma, \{l_i \Rightarrow \forall \mathrm{B}_i.\theta_i\}_{i=1}^n \vdash M : \tau^\diamond \to \tau^\bullet}}{\Gamma, \{l_i \Rightarrow \forall A_i.\theta_i\}_{i=1}^n \vdash \downarrow; M : \tau^\diamond \to \tau^\bullet}$$

with $B_i \subset A_i$ (thanks to the side condition).

Then the proof after elimination is,

$$\dfrac{\dfrac{\text{II}}{\Gamma, \{l_i \Rightarrow \forall \mathrm{B}_i.\theta_i\}_{i=1}^n \vdash N : \tau^\diamond \to \tau^\bullet}}{\Gamma, \{l_i \Rightarrow \forall A_i.\theta_i\}_{i=1}^n \vdash M : \tau^\diamond \to \tau^\bullet}$$

We do not detail cases for $\equiv$: there are 9, as many as combinations of application, abstraction, and composition (we do not distinguish Abs and Abs'). All proofs are easy. □

7.3.3 Monotype unification

We solve a system of equations on monotypes, return types and stream types. Here we make no distinction between $\{\} \to \tau^\bullet$ and $\tau^\bullet$, $\{\} \cdot \alpha^\diamond$ and $\alpha^\diamond$. Equations, in Figure 7.4, are considered modulo symmetry.

Termination A measure based on *(unsolved variables,labels+non-trivial →)*, where a trivial rightarrow appears in $\{\} \to \tau^\bullet$.

Correctness We prove that, for each rewrite rule, any solution of the denominator is a solution for the numerator, and conversely.

This is clear for the failure rules.

Variable elimination substitutes variables (using flattening type substitution), while keeping their reference. Let σ be a solution of the numerator. Then, by construction, $\sigma(\alpha) = \sigma(\tau)$, thus t is also a solution of the denominator.

Redundancy only suppresses useless equations (and harmless) equations, which makes it clearly correct.

Splitting and *Decomposition* do the correct structural transformations on terms.

As a conclusion, all these rules are safe tranformations of our equation system.

Validity Last we must verify that the final state of our equation system expresses it solution. That is, all equations must be of the form $\alpha^* = \tau^*$, where α^* appears in no other equation.

First, if our equation is of the form $\alpha^* = \tau^*$, $\alpha^* \notin Var(\tau^*)$ and $\tau^* \notin v^*$, $\alpha^* \in Var(\varphi)$ would mean that *Variable elimination* applies. If $\alpha^* \in Var(\tau^*)$ then *Variable reccurence* or *Redundancy* applies. If $\tau^* = \beta^*$, and $\beta^* \notin Var(\varphi)$, then it applies the second condition applies for the β^* side.

Second, we show that all equations must be of the form $\alpha^* = \tau^*$: if the two sides have compatible structures, they are submitted to one of *Redundancy*, *Splitting* and *Decomposition*, otherwise they cause a failure.

Our form constraint is enough to guarantee the existence of a most general unifier, precisely σ^n, where σ is the substitution mapping α^* to τ^* for each equation, and n the number of variables (we may have to follow links, but may not find the same variable two times in a path).

Theorem 7.1 (most-general unifier) *The above proves the existence of an algorithm giving either the mgu of a set of equations on monotypes, or reporting failure.*

7.3.4 Type inference

The type inference algorithm for the polymorphic transformation calculus is in Figure 7.5.

Remarks In the absence of principal typing, the above algorithm cannot be "most general" in any way. In fact, it is not even confluent, since *stripping* is not monotonous: a closed transformation may be generalized, but not a semi-open one.

Example 7.1 *We go on with our point example, and give types for our different transformations.*

- $defpoint : \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\}\}$.
- $lookpoint : \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\diamond\} \rightarrow \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\diamond, x \Rightarrow int, y \Rightarrow int\}$.
- $move : \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\diamond, x \Rightarrow int, y \Rightarrow int\} \rightarrow \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\diamond\}$.

We see here that polymorphism gives us "inheritance" for streams. For instance, if we apply $move$ on a $mypoint$ with more fields than x and y, it will work and leave other fields unchanged.

- $Tp(\Gamma, A, x) = \langle \{\mathsf{strip}(\Gamma(x)) = \mathsf{strip}(A) \cdot \alpha^{\diamond} \rightarrow \alpha^{\bullet}\}, \alpha \rangle$ where α fresh
- $Tp(\Gamma, \{l \Rightarrow \sigma\} \cdot A, \lambda\{l \Rightarrow x\}.M) = Tp(\Gamma[x \mapsto \sigma], A, M)$
- $Tp(\Gamma, A, \lambda\{l \Rightarrow x : \theta\}.M) = \langle \varphi, \phi_A^{-1}\{l \Rightarrow \theta'\} \cdot \tau^{\diamond} \rightarrow \tau^{\bullet} \rangle$

$$\text{where} \begin{cases} \langle \varphi, \tau \rangle = Tp(\Gamma[x \mapsto \theta'], \phi^{-1}_{\{l \Rightarrow x\}}(A), M) \\ \theta' = [\vec{\alpha}/Var(\theta)]\theta, \vec{\alpha} \text{ fresh variables} \end{cases}$$

- $Tp(\Gamma, A, M\{l \Rightarrow N\}) = \langle \mathsf{sol}(\varphi \cup \varphi'), \tau \rangle$

$$\text{where} \begin{cases} \langle \varphi, \theta \rangle = Tp(\Gamma, \{\}, N) \\ \langle \varphi', \tau \rangle = Tp(\Gamma', \{l \Rightarrow \sigma\} \cdot A', M) \\ \Gamma' = \varphi(\Gamma), A' = \varphi(A) \\ \sigma = \mathsf{wrap}(\varphi(\theta), Var(\Gamma') \cup Var(A')) \end{cases}$$

- $Tp(\Gamma, A, M; N) = \langle \mathsf{sol}(\varphi' \cup \varphi''), \tau^{\diamond} \rightarrow \theta^{\bullet} \rangle$

$$\text{where} \begin{cases} \langle \varphi, \tau \rangle = Tp(\Gamma, A, M) \\ \langle \varphi'', \theta \rangle = Tp(\Gamma', B', N) \\ \langle B, r \rangle = \mathsf{bone}(\varphi(\tau^{\bullet})) \\ \varphi' = \mathsf{sol}(\varphi \cup \{r = \theta^{\diamond}\}), \Gamma' = \varphi'(\Gamma) \\ B' = \mathsf{wraps}(\varphi'(B), \varphi'(Var(\Gamma) \cup Var(A) \cup Var(\tau^{\diamond}))) \end{cases}$$

- bone : $\begin{array}{lcl} \{l_i \Rightarrow \tau_i\}_1^n & \mapsto & \langle \{l_i \Rightarrow \tau_i\}_1^n, \{\} \rangle \\ \{l_i \Rightarrow \tau_i\}_1^n \cdot \alpha^{\diamond} & \mapsto & \langle \{l_i \Rightarrow \tau_i\}_1^n, \alpha^{\diamond} \rangle \end{array}$
- open = *opens the transformation types.*
- sol(φ) = *most general unifier of* φ.
- $\mathsf{strip}(\forall \vec{\alpha}.\tau) = \mathsf{open}([\vec{\nu}/\vec{\alpha}]\tau)$
- $\mathsf{wrap}(\tau, BV) = \forall (Var(\tau) - BV).\tau$
- $\mathsf{wraps}(A, BV) = \mathsf{map}\ (\lambda\tau.\mathsf{wrap}(\tau, BV))\ A$

Figure 7.5: Type reconstruction algorithm

7.3.5 Polymorphic records

An extension suggested by this polymorphism on streams is the use of an extended pattern matching on them. Extracting a field from a stream can then be done polymorphically, without extending the typing system. This is all we need to get a complete polymorphic record calculus, since we already had modification. Here is the example of a field selector for label l. $\{l \Rightarrow x\} \cdot y$ is an extended pattern, matching the value on l with x and the rest of the stream with y, and it can come in place of a variable.

$$\#l = \lambda\{1 \Rightarrow \{l \Rightarrow x\} \cdot y\}.x : \{1 \Rightarrow \{l \Rightarrow \alpha\} \cdot \beta^{\diamond}\} \rightarrow \alpha$$

Equivalently we could directly add $\#l$ as an operator, with the following reduction rule:

$$(\{l \Rightarrow M, \ldots\}.\downarrow).\#l \rightarrow M$$

This is enough to define all extended patterns.

It works nicely —since this extension is orthogonal, confluence is kept—, however such types are fragile, in that the same polymorphic stream should not be used as a transformation, since constraints would interfere. This is not surprising: transformation composition gives us record concatenation, and such an operation has no most generic type (see [Wan91] in a slightly different system: generally label repetition is handled by retaining only one of the two occurrences, and we keep the two by our label-shifting). If this rule against mixing is respected, we can enjoy record operations as an "extra".

7.4 FIML: a typed language based on the transformation calculus

We define in Appendix B a language based on the polymorphically typed transformation calculus

7.5 Transformations and stateful objects

The idea here is to use the state handling mechanisms defined in Chapter 6 to implement stateful objects, by associating an object with a label. That is, for each object we should have a label to which its state would be attached. By making its use exclusive, we can assimilate modifications of the object's state to modifications of the value associated to the label, through a transformation.

This is a neat way to encode objects, but we must remember that what we encode with scope-free variables is *dynamic binding*: a name is in no way exclusive. We have form of dynamic locality, limiting the life area of a variable, but this is not the kind of locality we need for object-oriented programming, where we want on the contrary to have the life area of a private variable (the use scope) larger than its syntactical scope (the object itself).

To solve this problem, we first define a transformation calculus with scoped labels, and then demonstrate its use.

7.5.1 The scoped transformation calculus

We define this calculus as an extension of the untyped transformation calculus.

7.5.1.1 Syntax

Terms are extended with a label-scoping construct,

$$M ::= x \mid \lambda\{l \Rightarrow x, \ldots\}.M \mid \{l \Rightarrow M\}.M \mid \downarrow \mid M; M \mid \nu p.M$$

where p is a name in $\mathcal{L}_s$.

$FN(M)$, the set of free names in M, is defined in the usual way, $\nu p.N$ hiding occurences of p in N.

The interactions of this construct with others is defined by the following new structural equivalences.

$$\begin{array}{rcll} \nu p.M & \equiv & M & p \notin FN(M) \\ \nu p.M & \equiv & \nu q.[q/p]M & q \notin FN(M) \\ \nu p.\nu q.M & \equiv & \nu q.\nu p.M & \end{array}$$

$$\begin{array}{rcll} \lambda\{pm \Rightarrow x\}.\nu q.M & \equiv & \nu q.\lambda\{pm \Rightarrow x\}.M & p \neq q \\ \{pm \Rightarrow N\}.\nu q.M & \equiv & \nu q.\{pm \Rightarrow N\}.M & p \neq q, q \notin FN(N) \\ M; \nu p.N & \equiv & \nu p.(M; N) & p \notin FN(M) \\ \nu p.M; N & \equiv & \nu p.(M; N) & p \notin FN(N) \end{array}$$

Finally we extend variable substition with

$$[N/x](\nu p.M) = \nu p.[N/x]M \qquad p \notin FN(N)$$

Reduction rules themselves are not modified.

The key point in these rules is that we don't equate $\{pm \Rightarrow \nu q.N\}.M$ and $\nu q.\{pm \Rightarrow N\}.M$. This would, in this form, make the calculus incoherent, scopes depending of the form on which we apply β-reduction —if the νq is out all residuates of N are in the same scope, otherwise they are in different ones—, and even if we restrict β reductions to terms containing no ν, this would lose the name generating power: once all nu pushed out we can just forget about them for all internal reductions.

By keeping the ν in the applications, we create different scopes for each residual of N. This amounts to creating news names for each of them, and gives us static binding.

Remark however that this does not mean that transformation calculus suddenly changed from dynamic to static binding. That only means that they are different problems: scope-free variables are about where a value is available in a program, whereas name scopes are about how one can access it (or not). There is no point in scoping a variable if we know that it will not interfer with other scope-free variables in its use area.

Since the only thing we have done here is to add scopes to names, this calculus is still confluent.

Theorem 7.2 *The scoped transformation calculus is Church-Rosser.*

PROOF We first prove that all structural equivalences and reduction rules keep unchanged scopes of names.

For the above equivalences, this is exactly the role of the side conditions. For the original ones, they do not change scopes.

$\downarrow$-elimination is immediate again.

Last, β-reduction keeps scopes, thanks to our new definition of substitutions.

We now can consider our terms as classical transformation calculus terms, plus some scoping. For that we remark that in all transformational "spines", we can push all the ν's outwards, and have terms of the form:

$$\begin{array}{rcl} N & ::= & x \mid \lambda\{l \Rightarrow x\}.N \mid \{l \Rightarrow M\}.N \mid \downarrow \mid N; N \\ M & ::= & N \mid \nu p.M \end{array}$$

External ν's do not interact with structural equivalences: $\nu\vec{p}.M \equiv \nu\vec{p}.N$ (M and N not name-scoped) implies $M \equiv N$. That means that our scoping do not equate originally different terms.

Thanks to all that we can (sketch):

1. translate a reduction in the scoped calculus into one in a variant of the classical one (β-reduction of $\{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M$ may change labels in the various residuals of N, accordind to scopes).
2. use the confluence of the transformation calculus to get closing reductions in the scoped calculus.

This gets us confluence. □

7.5.1.2 Simple types

We only present here simple types, but extension to polymorphism is of course possible.

Types are extended in

$$\begin{array}{lcl} r & ::= & \{l \Rightarrow t, \ldots\} \\ w & ::= & u \mid r \\ t & ::= & r \rightarrow w \mid \nu p.t \end{array}$$

with the three scoping equivalences we had already on terms.

We must slightly modify scoping/abstraction structural equivalence to cope with the presence of (possibly scoped) types in abstractions.

$$\lambda\{pm \Rightarrow x{:}\tau\}.\nu q.M \equiv \nu q.\lambda\{pm \Rightarrow x{:}\tau\}.M \qquad p \neq q, q \notin FN(\tau)$$

Typing rules have to be modified:

$$\frac{\Gamma[x \mapsto \theta] \vdash M : \nu\vec{p}.(r \rightarrow w)}{\Gamma \vdash \lambda\{qn \Rightarrow x{:}\theta\}.M : \nu\vec{p}.(\{qn \Rightarrow \theta\} \cdot r \rightarrow w)}\ q \notin \vec{p} \qquad \text{(II)}$$

$$\frac{\Gamma \vdash M : \nu\vec{p}.(\{qn \Rightarrow \theta\} \cdot r \rightarrow w) \qquad \Gamma \vdash N : \theta}{\Gamma \vdash \{qn \Rightarrow N\}.M : \nu\vec{p}.(r \rightarrow w)}\ q \notin \vec{p} \qquad \text{(III)}$$

$$\frac{\Gamma \vdash M : \nu\vec{p}.(r_1 \rightarrow r_2) \qquad \Gamma \vdash N : \nu\vec{q}.(r_2 \rightarrow r_3)}{\Gamma \vdash M; N : \nu\vec{p}.\nu\vec{q}.(r_1 \rightarrow r_3)}\ \vec{p} \not\bowtie \vec{q} \qquad \text{(V)}$$

$$\frac{\Gamma \vdash M : \nu\vec{p}.(r_1 \rightarrow r_2)}{\Gamma \vdash M : \nu\vec{p}.\nu\vec{q}.(r_1 \cdot r \rightarrow r_2 \cdot r)}\ \begin{array}{l} \vec{q} \not\bowtie FN(r_1 \rightarrow r_2) \\ \vec{p} \not\bowtie FN(r) \end{array} \qquad \text{(VI)}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \nu p.M : \nu p.\tau} \qquad \text{(VII)}$$

Proposition 7.5 (subject reduction) *If $\Gamma \vdash M : \tau$ in the scoped transformation calculus, and $M \equiv N$ or $M \rightarrow N$, then $\Gamma \vdash N : \tau$.*

7.5.2 Stateful objects

In this new calculus, we can create a unique relation between an object an its state, by scoping its internal variables. This gives us a simple definition of the methods attached to an object. They are transformations accessing or modifying the values on scoped labels. This means that at the transformation calculus level an object is separated into a state and a set of methods, the only relation between the two being these scoped labels.

$$\text{screen} = \nu s.\left\{\left\{\begin{array}{l} plot \Rightarrow \lambda\{s \Rightarrow n\}.(\{nb \Rightarrow n\}.\text{plot_screen}; \{s \Rightarrow n\}.\downarrow) \\ init \Rightarrow \lambda(x).\{s \Rightarrow x\}.\downarrow \\ del \Rightarrow \lambda\{s \Rightarrow x\}.\downarrow \end{array}\right\}\right\}$$

$$: \quad \nu s.\left\{\left\{\begin{array}{l} plot \Rightarrow \{1 \Rightarrow int, 2 \Rightarrow int, s \Rightarrow int\} \rightarrow \{s \Rightarrow int\} \\ init \Rightarrow \{1 \Rightarrow int\} \rightarrow \{s \Rightarrow int\} \\ del \Rightarrow \{s \Rightarrow int\} \rightarrow \{\} \end{array}\right\}\right\}$$

$$\text{point} = \nu x.\nu y.\left\{\left\{\begin{array}{l} init \Rightarrow \lambda(x,y).\{x \Rightarrow x, y \Rightarrow y\}.\downarrow \\ del \Rightarrow \lambda\{x \Rightarrow x, y \Rightarrow y\}..\downarrow \\ move \Rightarrow \lambda\{1 \Rightarrow x', 2 \Rightarrow y', x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x + x', y \Rightarrow y + y'\}.\downarrow \\ plot \Rightarrow \lambda\{scr \Rightarrow s, x \Rightarrow x, y \Rightarrow y\}. \\ \qquad\qquad ((x,y).(s).\#plot; \\ \qquad\qquad \{x \Rightarrow x, y \Rightarrow y\}.\downarrow) \end{array}\right\}\right\}$$

$$: \quad \nu x.\nu y.\left\{\left\{\begin{array}{l} init \Rightarrow \{1 \Rightarrow int, 2 \Rightarrow int\} \rightarrow \{x \Rightarrow int, y \Rightarrow int\} \\ del \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \rightarrow \{\} \\ move \Rightarrow \{scr \Rightarrow \{plot \Rightarrow \{1 \Rightarrow int, 2 \Rightarrow int\} \cdot \alpha^\diamond \rightarrow \beta^\diamond\} \cdot \gamma^\diamond, \\ \qquad\qquad x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\diamond \rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \beta^\diamond \end{array}\right\}\right\}$$

Figure 7.6: Examples of prototypes

This approach differs from a frequent one in the object oriented field, which sees methods as being invocated by messages sent to the object [AR88, Red88], which is considered as "active" and "atomic" (methods and state are together *in* the objet). In our approach an object is simply a set of transformations implementing methods, plus a label holding its state. In ignoring the notion messages we do like [Car88], but he suggested to include the state in the record representing the methods. The distinction we make here avoids self-references, which made necessary recursively quantified types.

This distinction results, we will see, in a two-step way of creating an object from its prototype. First we extract the methods from the prototype. Only then can we use the `init` method to create its state. Then, like with scope-free variables, we have composed transformations based on this object's methods, and finally we delete its state.

The definition of a prototype is done in the following way.

1. We define the methods in a stream, they must all be transformations.

   $$\{\text{meth}_1 \Rightarrow T_1, \ldots, \text{meth}_n \Rightarrow T_n\}.\downarrow$$

2. Then we pack these methods by scoping private names.

   $$\text{proto} = \nu p_1 \ldots . \nu p_m . \{1 \Rightarrow \{\text{meth}_1 \Rightarrow T_1, \ldots, \text{meth}_n \Rightarrow T_n\}.\downarrow\}.\downarrow$$

   The need for this "double packing" will appear clearly with object creation.

We give in Figure 7.6 an example illustrating prototype definition (we abbreviate $.\downarrow$ when no confusion is possible). The integer we use as screen state is in fact a dummy state. The only real modification is done by the low level operation *plot_screen* but by reading and writing the screen number we simulate here a modification.

The next operation is the creation of an object from a prototype. That is, if we try to extract methods directly from prototypes, their scoped labels will be quantified

independently, so that they will not match each other. Object creation is done by creating a stream containing the methods of the prototype — but not the quantifiers.

proto; $\lambda\{1 \Rightarrow obj\}.M$

Let us see what is the result of this composition.

$$\begin{array}{ll} & \nu p_1 \ldots . \nu p_m . \{1 \Rightarrow \{\text{meth}_1 \Rightarrow T_1, \ldots, \text{meth}_n \Rightarrow T_n\}\}.\downarrow; \lambda\{1 \Rightarrow obj\}.M \\ \rightarrow & \nu p_1 \ldots . \nu p_m . \{1 \Rightarrow \{\text{meth}_1 \Rightarrow T_1, \ldots, \text{meth}_n \Rightarrow T_n\}\}.\lambda\{1 \Rightarrow obj\}.M \\ \rightarrow & \nu p_1 \ldots . \nu p_m . [\{\text{meth}_1 \Rightarrow T_1, \ldots, \text{meth}_n \Rightarrow T_n\}/obj]M \end{array}$$

This does what we wanted, the scope of the labels being now external to obj.
We define a context C with one hole, where one screen and two points are created.

$C[] =$ screen; point; point; $\lambda\{p1, p2, s1\}.[]$

Once our objects created, we use them by composing methods from different objects. We first initialize our objects, then move the point, plot it on the screen, and suppress its object.

$e_1 = (0,0).(p1).\#init; (3,4).(p2).\#init; (1).(s1).\#init$

$C[e_1] \xrightarrow{*} \nu s x_1 y_1 x_2 y_2 . \{x_1 \Rightarrow 0, y_1 \Rightarrow 0, x_2 \Rightarrow 3, y_2 \Rightarrow 4, s \Rightarrow 1\}$

$e_2 = e_1; (10,15).(p1).\#move$

$C[e_2] \xrightarrow{*} \nu s x_1 y_1 x_2 y_2 . \{x_1 \Rightarrow 10, y_1 \Rightarrow 15, x_2 \Rightarrow 3, y_2 \rightarrow 4, s \Rightarrow 1\}$

$e_3 = e_2; (s1).(p1).\#plot$

$C[e_3] \xrightarrow{*} C'[\{x_1 \Rightarrow 10, y_1 \Rightarrow 15, x_2 \Rightarrow 3, y_2 \rightarrow 4, s \Rightarrow 1\}.(10,15).(s1).\#plot]$

where C' is the normal version of C.

$e_4 = e_3; (p1).\#del$

$C[e_4] \xrightarrow{*} \nu s x_1 y_1 x_2 y_2 . \{x_2 \Rightarrow 3, y_2 \Rightarrow 4, s \Rightarrow 1\}$

Security *vs.* **interactivity**

We have given here a practical approach, but it lacks security. Objects are only accessible from their methods, but since $init$ and del are accessible from anywhere in a calculation, we may well try to use a method on an object not yet initialized, or already destroyed, or even create more than one state for the same object (labels).

In a robust scheme, the definition of the labels should have the same locality as the object, that is all object should be defined locally to a calculation, being initialized at its beginning and destroyed at its end, with no initialization nor destruction method accessible during this calculation, nor explicit access to its label.

In this last scheme we create and initialize simultaneously an object, and a prototypes is transformations of the form:

$$\begin{array}{l} \text{proto} = \lambda\{l_1 \Rightarrow x_1, \ldots\}.\nu p_1 \ldots . \nu p_m . \{p_1 \Rightarrow M_1, \ldots, p_m \Rightarrow M_m, \\ \qquad\qquad met \Rightarrow \{\text{meth}_1 \Rightarrow T_1, \ldots, \text{meth}_n \Rightarrow T_n\}, del \Rightarrow \lambda\{p_1 \Rightarrow y_1, \ldots, p_m \Rightarrow y_m\}.\downarrow\}.\downarrow \end{array}$$

where M_i's are initialization functions.

We have a construct "`let object` *obj* = *proto init-values* `in` T", which is syntactic sugar for

$$\textit{init-values.proto}; \lambda\{met \Rightarrow obj, del \Rightarrow del\}.(T; del)$$

Now, *obj* is accessible only in the transformation part T, and gives access to methods using a scoped label for the object state. These methods should get part of the object state as input, and output the same part, eventually modified. This is only a constraint on the use of the scoped labels. Such a scheme gives perfect encapsulation and robustness, object states being accessible only through their methods, and having exactly the same locality.

Nonetheless, in an interactive environment for instance, one may want to define an object globally, or with a less rigid locality. In this case we must allow initialization and destruction methods, letting the user control the object's existence by himself, the methods being possibly by a "`let object` *objname* = *prototypename*" command at toplevel, like above. We lose in security, since we may try accessing to an object already destroyed, or still not created. More dangerous, we may create more than one instance of the same object, with same name and same state label. But at the same time it is a powerful way of interactive control: we may locally change the state accessed by an object, initializing it once more, and after that recover the original state, by destroying this temporary state. And this while keeping the same identity.

Chapter 8

Symmetric Transformations

Currying, by allowing the use of mutiple arguments whereas the result must be single, makes the λ-calculus asymmetrical. The symmetry of types, with respect to the arrow, is only superficial, since $\alpha \to \beta \to \gamma$ is in fact equivalent to $(\alpha \times \beta) \to \gamma$.

A first way to correct this has been categorical formulation of the λ-calculus, with, for instance, *categorical combinators* [Cur93]. By allowing products, we recover symmetry for types. We proposed another way to do that, allowing a stronger form of currying (of both input and output) with the *transformation calculus* [Gar94]. Again, types are symmetrical. On a different ground, the *symmetrical λ-calculus* [Fil89] is about semantical duality between values and continuations.

However, this does nothing about another asymmetry of λ-calculus, that of syntax. While application takes two terms, abstraction takes one variable and one term. Why not allow terms everywhere? For this we first need to distinguish the scoping and binding roles of λ-abstraction, and have a new, bidirectional view on the calculus.

8.1 Logical version

In this first version we work with a system of logical relations, where β-reduction acts as unification.

8.1.1 Grammar

Our terms are defined by the following grammar

l	::=	pn	label
M	::=	x	variable
	|	$\{l \Rightarrow M, \ldots\}$	stream
	|	$M; M$	composition
	|	$\overline{M}$	conjugation
	|	$M \mid M$	disjunction
	|	$\forall x.M$	scoping
	|	M/M	constraint
	|	$\bot$	failure

A number of things may look surprising in this definition. First, the absence of application. In fact we replace it by composition: application itself is intrinsically asymmetric. Similarly, in a logical context we can reduce abstraction to scoping.

Last, conjugation is the new operation which reverses the syntactical direction of a term. To have a β-reduction, we need to compose terms with different directions.

The constraint constructor is specific to the logical version. It is made necessary by the global effect of term reduction. That is, reducing a subterm may unify variables appearing out of it. If we delete this subterm, we may loose confluence. By this construct, we leave unevaluated subterms in the term. In M/N, the "real" term is M and the constraint one is N.

8.1.2 Structural rules

We have a number of structural equalities on our terms. They can be divided in four groups, according to which construct they deal with. *Composition*, *Conjugation* and *Scoping* are common to the functional version, but *Disjunction* and *Constraint* are only logical.

- Composition

$R = R_1 \cdot R_2 \Rightarrow R \equiv R_2; R_1$	concatenation
$M_1; (M_2; M_3) \equiv (M_1; M_2); M_3$	associativity
$M; \{\} \equiv \{\}; M \equiv M$	neutral
$\{l \Rightarrow M_1\}; \{\overline{l \Rightarrow M_2}\} \equiv \{l \Rightarrow M_2\}; \{\overline{l \Rightarrow M_1}\}$	symmetry
$\{l_1 \Rightarrow M_1\}; \{\overline{l_2 \Rightarrow M_2}\} \equiv \{\overline{\psi_{l_1}(l_2) \Rightarrow M_2}\}; \{\psi_{l_2}(l_1) \Rightarrow M_1\}$	transparency $(l_1 \neq l_2)$

- Conjugation

$\overline{\{\}} \equiv \{\}$	$\overline{M_1; M_2} \equiv \overline{M_2}; \overline{M_1}$	
$\overline{\overline{M}} \equiv M$	$\{l \Rightarrow M\}; \{\overline{l \Rightarrow N}\} \equiv \{l \Rightarrow \overline{M}\}; \{\overline{l \Rightarrow \overline{N}}\}$	duality

Concatenation introduces the monoidal structure of streams into the calculus. If we work with tuples, this becomes:

$$(M_1, \ldots, M_n) \equiv (M_{k+1}, \ldots, M_n); (M_1, \ldots, M_k)$$

The inversion comes from the fact we use the "flowing" composition ";", by opposition to the backwards one "$\circ$". Looking from the right, the right tuple comes before the left one.

Transparency extends this structure to differently oriented terms. A label is "transparent" to different labels coming in the opposite direction. ψ is defined on streams as the reverse shifting function, that is $\{l_1 \Rightarrow M_1\} \cdot \{\psi_{l_1}(l_2) \Rightarrow M_2\} = \{l_1 \Rightarrow M_1, l_2 \Rightarrow M_2\}$. As a result,

$$\begin{aligned} &\{l_1 \Rightarrow M_1, l_2 \Rightarrow M_2'\}; \{\overline{l_1 \Rightarrow M_1', l_2 \Rightarrow M_2}\} \\ &\quad\equiv\quad \{\psi_{l_1}(l_2) \Rightarrow M_2'\}; \{l_1 \Rightarrow M_1\}; \{\overline{l_2 \Rightarrow M_2}\}; \{\overline{\psi_{l_2}(l_1) \Rightarrow M_1'}\} \\ &\quad\equiv\quad \{\psi_{l_1}(l_2) \Rightarrow M_2'\}; \{\overline{\psi_{l_1}(l_2) \Rightarrow M_2}\}; \{\psi_{l_2}(l_1) \Rightarrow M_1\}; \{\overline{\psi_{l_2}(l_1) \Rightarrow M_1'}\}, \end{aligned}$$

which explains the equality, the same that appears in the transformation calculus.

Associativity, along with equalities for conjugation and quantification are straightforward (just notice the inversion in $\overline{M_1; M_2} \equiv \overline{M_2}; \overline{M_1}$).

Symmetry and *duality* are more related to the fact they apply on redexes than to composition or conjugation: unification is the same in both directions, and unifying conjugates is the same as unifying originals.

- Disjunction

$\overline{M_1 \mid M_2} \equiv \overline{M_1} \mid \overline{M_2}$	
$M_1; (M_2 \mid M_3) \equiv M_1; M_2 \mid M_1; M_3$	$(M_1 \mid M_2); M_3 \equiv M_1; M_3 \mid M_2; M_3$
$M_1 \mid M_2 \equiv M_2 \mid M_1$	$(M_1 \mid M_2) \mid M_3 \equiv M_1 \mid (M_2 \mid M_3)$

- Quantification

$$\begin{array}{ll}
\forall x.\forall y.M \equiv \forall y.\forall x.M & \\
\forall x.M \equiv M & x \notin FV(M) \\
(\forall x.M_1); M_2 \equiv \forall x.(M_1; M_2) & x \notin FV(M_2) \\
M_1; (\forall x.M_2) \equiv \forall x.(M_1; M_2) & x \notin FV(M_1) \\
(\forall x.M_1) \mid M_2 \equiv \forall x.(M_1 \mid M_2) & x \notin FV(M_2) \\
M_1 \mid (\forall x.M_2) \equiv \forall x.(M_1 M_2) & x \notin FV(M_1) \\
\overline{\forall x.M} \equiv \forall x.\overline{M} & \\
\forall x.M \equiv \forall y.[x/y]M & y \notin FV(M)
\end{array}$$

Disjunction creates two worlds, one with its left side, and one with its right side, while quantification simply scopes variables. They have in common that they cannot get out of the stream they are in: this is how we preserve their locality.

- Constraints

$$\begin{array}{ll}
(M_1/M_2)/M_3 \equiv M_1/(M_2/M_3) & M_1/M_2/M_3 \equiv M_1/M_3/M_2 \\
M/x \equiv M & M/\{\} \equiv M \\
M/(\{\overline{l_1 \Rightarrow M_1, \ldots, l_k \Rightarrow M_k}\}; \{l_{k+1} \Rightarrow M_{k+1}, \ldots, l_n \Rightarrow M_n\}) \equiv M/M_1/\ldots/M_n & \\
M_1/M_2; M_3 \equiv (M_1; M_3)/M_2 & M_1; M_2/M_3 \equiv (M_1; M_2)/M_3 \\
M_1/(M_2 \mid M_3) \equiv M_1/M_2 \mid M_1/M_3 & (M_1 \mid M_2)/M_3 \equiv M_1/M_3 \mid M_2/M_3 \\
M_1/\overline{M_2} \equiv M_1/M_2 & \{l \Rightarrow M_1/M_2\} \equiv \{l \Rightarrow M_1\}/M_2 \\
\forall x.M_1/M_2 \equiv (\forall x.M_1)/M_2 \ \ (x \notin M_2) & \forall x.M_1/M_2 \equiv M_1/(\forall x.M_2) \ \ (x \notin M_1)
\end{array}$$

For this last construct, the equations are numerous, but they only express that a constraint may go anywhere in the scope of the variables it contains, as long it does not cross a disjunction, and may loose its term structure out of potential redexes. Some of the equations are redundant; but it is clearer to have them.

The second row are the two base cases where a constraint may simply disappear. By these rules any constraint containing no potential redexes when not quantified may disappear. However we do not analyze cases like $M/\forall x.(x; x)$ (unreducible constraint) for the sake of simplicity.

Definition 8.1 (first-level stable) *A term is* first-level stable *if it can be put in the form*

$$\forall x_1 \ldots x_n.(\{\overline{l \Rightarrow M, \ldots}\}; \{l' \Rightarrow M', \ldots\}).$$

That is, no first-level redexes, nor variables, nor constraints, nor external quantifiers.

8.1.3 Reduction rules

We only present here a very weak reduction system, but it is sound.

$$\beta \qquad \frac{\forall x.C[\{l \Rightarrow x\}; \{\overline{l \Rightarrow M}\}]}{C[\{\}/M][x \backslash M]} \ (x \notin FV(M))$$

C should not contain a disjunction on the way to its hole.

$$\text{Match} \qquad \frac{\{l \Rightarrow \{\overline{l_i \Rightarrow M_i}\}_1^k; \{l_i \Rightarrow M_i\}_{k+1}^n\}; \{\overline{l \Rightarrow \{\overline{l_i \Rightarrow M_i}\}_1^k; \{l_i \Rightarrow M_i\}_{k+1}^n}\}}{\{\}/(\{l_i \Rightarrow M_i\}_1^n; \{\overline{l_i \Rightarrow N_i}\}_1^n}$$

$$\text{Id} \qquad \frac{\{l \Rightarrow M\}; \{\overline{l \Rightarrow M}\}}{\{\}/M}$$

$$\text{Mismatch}\,\frac{\{l \Rightarrow \forall x_1 \dots x_m.M\}; \{\overline{l \Rightarrow \forall x'_1 \dots x'_m.N}\}}{\bot}\ (*)$$

(*) Only when M and N are first-level stable without external quantifiers, and Match fails on $\{l \Rightarrow M\}; \{\overline{l \Rightarrow N}\}$.

Failure

$$\begin{array}{lll} \{l \Rightarrow \bot\} \rightarrow \bot & \bot; M \rightarrow \bot & M; \bot \rightarrow \bot \\ ov\bot \rightarrow \bot & \forall x.\bot \rightarrow \bot & M \mid \bot \rightarrow M \\ \bot/M \rightarrow \bot & M/\bot \rightarrow \bot & \end{array}$$

β gets its name from its role similar to β-reduction. We keep M as a constraint for the case x should not appear in the context C. *Match* decomposes terms for unification. *Id* suppresses redundant redexes, while keeping the original M for its possible internal redexes.

Last, *Mismatch* and *Failure* makes inconsistent terms end into error. The only way to recover from an error is through a disjunction.

These rules are very weak in that failure happens only on non-quantified terms: we do not try to unify functions. But the lambda-calculus is included, and we expect to have a confluent calculus, which is our main goal for this very simple version.

Conjecture 8.1 *The Church-Rosser Property holds in the logical symmetric transformation calculus.*

Everything was done for it to hold, but the numerous structural equivalences makes the problem complex.

Example 8.1 (reduction) We encode here a λ-calculus function, but in a bidirectional way.

$$\begin{array}{c} \{a \Rightarrow 1, f \Rightarrow \forall x.\{r \Rightarrow x+1, v \Rightarrow x\}\}; \forall x.(\{\overline{a \Rightarrow x}\}; \{v \Rightarrow x\}); \forall x.(\{\overline{f \Rightarrow x}\}; \overline{x}) \\ \downarrow \\ \{f \Rightarrow \forall x.\{r \Rightarrow x+1, v \Rightarrow x\}, v \Rightarrow 1\}; \forall x.(\{\overline{f \Rightarrow x}\}; \overline{x}) \\ \downarrow \\ \{v \Rightarrow 1\}; \forall x.\{\overline{r \Rightarrow x+1, v \Rightarrow x}\} \\ \downarrow \\ \{\overline{r \Rightarrow 2}\} \end{array}$$

8.1.4 Lambda calculus

The lambda-calculus is trivially encoded in this system by the following translation:

$$\begin{array}{rcl} T(x) & = & x \\ T(\lambda x.M) & = & \forall x.(\{\overline{l \Rightarrow x}\}; T(M)) \\ T(MN) & = & \{l \Rightarrow T(N)\}; T(M) \end{array}$$

In fact, this encoding of λ-calculus is very similar to that into the transformation calculus, which can itself be encoded into this calculus by just adding quantifiers were necessary.

We can easily show that this properly encodes the reductions of the call by value λ-calculus: since all substitutions are done with normal forms, we never get any constraints.

Since other strategies may introduce unnormalizable constraints, we must suppress constraints in our translation back, and ignore reduction on constraints, in order to identify reductions in our calculus with reductions in the lambda-calculus. This is not a problem, since constraints generated by λ-terms do not modify their free variables anyway.

8.1.5 Types

Here is a type system for this first version. Of course we have nothing like Curry-Howard isomorphism, since terms are not semantically directed. However types reproduce the syntactical orientation.

Grammar

Types are defined as follows. The bidirectional arrow recalls that we do not know in which direction data flows.

u	::=	$u_1 \mid u_2 \mid \ldots$	base types
r	::=	$\{l \Rightarrow t, \ldots\}$	stream types
w	::=	$u \mid r$	data types
t	::=	$w \leftrightarrow w$	types

We modify slightly the calculus for the use of types.

$$M ::= \ldots \mid \forall x{:}t.M \mid \ldots$$

Deduction rules

Typings defined here are very similar to those for the simply typed transformation calculus, but for the conjugation rule, which make the specificity of this calculus.

Variable $\quad \Gamma[x \mapsto \tau] \vdash x : \tau$

Stream
$$\frac{\Gamma \vdash M_i : \tau_i \qquad \forall i \in [1,n]}{\Gamma \vdash \{l_i \Rightarrow M_i\}_1^n : \{\} \leftrightarrow \{l_i \Rightarrow \tau_i\}_1^n}$$

Conjugation
$$\frac{\Gamma \vdash M : w \leftrightarrow w'}{\Gamma \vdash \overline{M} : w' \leftrightarrow w}$$

Composition
$$\frac{\Gamma \vdash M_1 : w_1 \leftrightarrow w \qquad \Gamma \vdash M_2 : w \leftrightarrow w_2}{\Gamma \vdash M_1; M_2 : w_1 \leftrightarrow w_2}$$

Disjunction
$$\frac{\Gamma \vdash M_1 : w \leftrightarrow w' \qquad \Gamma \vdash M_2 : w \leftrightarrow w'}{\Gamma \vdash M_1 \mid M_2 : w \leftrightarrow w'}$$

New
$$\frac{\Gamma[x \mapsto \theta] \vdash M : \tau}{\Gamma \vdash \forall x{:}\theta.M : \tau}$$

Subtype
$$\frac{\Gamma \vdash M : r_1 \leftrightarrow r_2}{\Gamma \vdash M : r_1 \cdot r \leftrightarrow r_2 \cdot r}$$

Constraint
$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \theta}{\Gamma \vdash M/N : \tau}$$

Failure $\quad \Gamma \vdash \bot : \tau$

Proposition 8.1 *The Subject Reduction Property holds.*

PROOF This is trivially true for reductions, once understood the behaviour of streams: only terms of same type are substituted, and reductions are always on null types ($\vdash \{\} : \{\} \leftrightarrow \{\}$).

We must then verify all structural rules are longer to verify, but they work. □

Proposition 8.2 *Well typed terms containing no $\bot$ do not reduce into failure.*

PROOF The only case of failure is structural inconsistency, and it is recognized by the type system. □

The last property comes from the fact we did not introduce typed constants into the calculus. With only variables, two identically typed terms are either unifiable or *blocking* (*i.e.* potential redex, but not immediately reducible).

8.1.6 Denotational semantics

The denotational semantics of the simply typed symmetric transformation calculus is very similar to that of the transformation calculus, once we replace functions by binary relations.

Definition 8.2 (model) *A model of the logical symmetric transformation calculus is a pair $(\mathcal{A}, [\![_]\!]_)$ with $\mathcal{A}$ a set of values, and $(M,\rho) \mapsto [\![M]\!]_\rho : \Lambda_L \times \mathcal{A}^{\mathcal{V}} \to \mathcal{A}$ (Λ_L the set of our terms) a translation from a simply typed term M and an environment ρ ($FV(M) \subset \mathcal{D}_\rho$) into our model satisfying the axioms*

$$\begin{array}{rcl} M \equiv N & \Rightarrow & [\![M]\!]_\rho = [\![N]\!]_\rho, \\ M \to N & \Rightarrow & [\![M]\!]_\rho = [\![N]\!]_\rho, \\ \rho(x) = a & \Rightarrow & [\![x]\!]_\rho = a. \end{array}$$

We define our model $\mathcal{A} = \bigcup_{\tau \in T} \mathcal{A}^\tau$ by closure of the following procedure. We note $\mathcal{P}(S)$ the set of the parts of S: $\{s \mid s \subset S\}$.

1. For $u \in U$ (base type), A^u is given.
2. $T_0 = \{w \leftrightarrow w' \mid w, w' \in U \cup \{\{\}\}\}$
3. Values of level n are defined by $\mathcal{A}_n = \bigcup_{\tau \in T_n} \mathcal{A}^\tau$ where

$$\mathcal{A}^{w \leftrightarrow w'} = \mathcal{P}(A^w \times A^{w'})$$

4. Streams values of level n are in $S_n = \bigcup_{r \in \mathcal{S}(T_n)} A^r$, where[1]

$$A^{\{l_i \Rrightarrow \tau_i\}_{i=1}^m} = \bigcup_{a_1 \in \mathcal{A}^{\tau_1} \setminus \{\emptyset\}} \cdots \bigcup_{a_m \in \mathcal{A}^{\tau_m} \setminus \{\emptyset\}} \{l_1 \Rrightarrow a_1, \ldots, l_m \Rrightarrow a_m\}$$

5. Types of level $n+1$ are defined by

$$T_{n+1} = T_n \cup \{w \leftrightarrow w' \mid w, w' \in \mathcal{S}(T_n) \cup T_0\}$$

6. $\mathcal{A} = \lim_{n \to \infty} \mathcal{A}_n$

[1] We must exclude $\emptyset$, since it will represent $\bot$, and $\bot$ extends to the whole relation.

$$
\begin{array}{lcl}
[\![x]\!]_\rho & = & \rho(x) \\
[\![\{l_i \Rightarrow M_i\}_{i=1}^n]\!]_\rho & = & \begin{cases} \emptyset & \text{if } \exists i,\ [\![M_i]\!]_\rho = \emptyset \\ \{(\{\}, \{l_i \Rightarrow [\![M_i]\!]_\rho\}_{i=1}^n)\} & \text{otherwise} \end{cases} \\
[\![M; N]\!]_\rho & = & [\![M]\!]_\rho ; [\![N]\!]_\rho \\
[\![\overline{M}]\!]_\rho & = & \{(x,y) \mid y [\![M]\!]_\rho x\} \\
[\![M \mid N]\!]_\rho & = & [\![M]\!]_\rho \cup [\![N]\!]_\rho \\
[\![\forall x{:}\tau.M]\!]_\rho & = & \bigcup_{a \in \mathcal{A}^\tau} [\![M]\!]_{\rho[x \mapsto a]} \\
[\![M/N]\!]_\rho & = & \begin{cases} \emptyset & \text{if } [\![N]\!]_\rho = \emptyset \\ [\![M]\!]_\rho & \text{otherwise} \end{cases} \\
[\![\bot]\!]_\rho & = & \emptyset
\end{array}
$$

Figure 8.1: Semantic function of the simply typed relational transformation calculus

$\mathcal{A}$ is well-defined, since for any τ there exists n such that $\tau \in T_n$ and then $\mathcal{A}^\tau \subset \mathcal{A}_n$.

Be careful that now, we cannot identify the values (constants or streams) of A^w with the relations of $\mathcal{A}^{\{\} \leftrightarrow w}$, since the latter is isomorphic to $\mathcal{P}(A^w)$.

We have four operations on the values of this model, *concatenation* (the usual one, limited to streams), *extension* (limited to transformations), *composition* and *disjunction* (of two compatible objects).

Since concatenation of two streams was already defined, we go directly to extension. This is the operation by which a value in $\mathcal{A}^{r_1 \leftrightarrow r_2}$ gets canonically extended into a value of $\mathcal{A}^{(r_1 \cdot r) \leftrightarrow (r_2 \cdot r)}$. If f is in $\mathcal{A}^{r_1 \leftrightarrow r_2}$ then $f * r$, the r-extension of f is defined as:

$$f * r = \{(x \cdot z, y \cdot z) \in A^{r_1 \cdot r} \times A^{r_2 \cdot r} \mid x f y, z \in A^r\}$$

Composition is the composition of binary relations. For two objects f and g respectively in $\mathcal{A}^{w_1 \leftrightarrow w}$ and $\mathcal{A}^{w \leftrightarrow w_2}$,

$$f; g = \{(x,y) \in A^{w_1} \times A^{w_2} \mid (\exists z \in \mathcal{A}^w)\ x f z \wedge z g y\}$$

Disjunction is the union of two relations. For f and g in $\mathcal{A}^{w \leftrightarrow w'}$,

$$f \cup g = \{(x,y) \in A^w \times A^{w'} \mid x f y \vee x g y\}$$

Like for the transformation calculus, we consider our model modulo extension. We introduce the equivalence $f =_* f * r$, and modulo this equivalence, $\mathcal{A}^{s_1 \leftrightarrow s_2}$ now includes all $\mathcal{A}^{r_1 \leftrightarrow r_2}$ such that for some r, $r_1 \cdot r = s_1$ and $r_2 \cdot r = s_2$.

We define $\mathcal{A}_*$ as $\mathcal{A}_{/=_*}$, and $\mathcal{A}_*^\tau$ as the sets of all classes containing an element of $\mathcal{A}^\tau$.

Again, composition and disjunction are coherent with this equivalence: $(f; g) * r = f * r; g * r$ and $(f \cup g) * r = f * r \cup g * r$.

Finally we define the translation $(M, \rho) \mapsto [\![M]\!]_\rho : \Lambda_T \times \mathcal{A}_*^{\mathcal{V}} \to \mathcal{A}_*$, from a simply typed term M and an environment ρ ($FV(M) \subset \mathcal{D}_\rho$) into our model[2] in figure 8.1.

Proposition 8.3 *$(\mathcal{A}_*, [\![_]\!]_)$ is a model of the simply typed relational transformation calculus.*

PROOF We must prove that our three axioms are verified.

- $\equiv$: the proof is direct for each basic equivalence.

[2]Same remark as for the transformation calculus: we can compile away labels, if we want...

- β: by hypothesis we have $[\![x]\!]_\rho = [\![M]\!]_\rho$, and this is valid for any occurence of x, since it was verified for a context with no disjunction on the path.
- *Match,Id*: by hypothesis.
- *Mismatch*: does not happen in a typed context without constants. By hypothesis in the presence of constants.
- *Failure*: by the structure of the model.

□

8.2 Functional version

Since the logical version already encodes the λ-calculus, one could wonder about the necessity of a functional one. One first answer is that, when computing, it is often more practical to have a functional view of things, knowing how the data flows, than a logical one, knowing only about constraints on it. Mode analysis in logic programming is an example of that. Moreover, if we see symmetrical transformations as a way to study syntactical properties of inversion, this is a tool allowing us to see the orientation of the data flows in communicating relations.

8.2.1 Syntax

We limit ourselves to use single variables as input, and full expressions as output.

$$M ::= x \mid \underline{x} \mid \{l \Rightarrow M, \ldots\} \mid M; M \mid \overline{M} \mid \forall x.M$$

The novelty is $\underline{x}$, which is a *binder*. More generally we call the operation $\underline{\cdot}$ *inversion*. We note $FB(M)$ the set of free binders in M. By definition $FB(M) \subset FV(M)$.

We do not use disjunction neither failure, since this time we are in a purely functional calculus.

Integrity condition. The above grammar defines *pre-terms*. Actual *symmetric terms* are those for which each binder does not appear free, and appears exactly once bound, for each quantifier.

Structural rules

Same as in paragraph 8.1.2, excepting those about disjunction and constraints, which become unnecessary.

Reduction rules

There are only two of them: β-reduction and failure.

$$\beta \qquad \frac{\forall x.C[\{l \Rightarrow \underline{x}\}; \{\overline{l \Rightarrow M}\}]}{C[\{\}][x \backslash M]} \ (x \notin FV(M))$$

$$\text{Fail} \qquad \frac{\{l \Rightarrow M_1\}; \{\overline{l \Rightarrow M_2}\}}{\bot} \ (*)$$

(*) where M_1 and M_2 are variables or first-level stable terms. Remark that failure is stronger here: structural matching is not allowed.

To keep confluence, failure is only generalized through composition, but not out of streams. Again, the Church-Rosser property is conjectured to hold, and λ-calculus is trivially encoded, modifying just the second definition (*cf.* 8.1.4):

$$T(\lambda x.M) = \forall x.(\{\overline{l \Rightarrow \underline{x}}\}; T(M))$$

Example 8.2 (reduction) We simply rewrite our previous example in the functional syntax.

$$\begin{array}{rc} & \{a \Rightarrow 1, f \Rightarrow \forall x.\{r \Rightarrow x+1, v \Rightarrow \underline{x}\}\}; \forall x.(\{\overline{a \Rightarrow \underline{x}}\}; \{v \Rightarrow x\}); \forall x.(\{\overline{f \Rightarrow \underline{x}}\}; \overline{x}) \\ \rightarrow & \{f \Rightarrow \forall x.\{r \Rightarrow x+1, v \Rightarrow \underline{x}\}, v \Rightarrow 1\}; \forall x.(\{\overline{f \Rightarrow \underline{x}}\}; \overline{x}) \\ \rightarrow & \{v \Rightarrow 1\}; \forall x.\{\overline{r \Rightarrow x+1, v \Rightarrow \underline{x}}\} \\ \rightarrow & \{\overline{r \Rightarrow 2}\} \end{array}$$

8.2.2 Typed functional relations

We can extend the original type system in the following way.

We extend only relation types:

$t ::= w \leftrightarrow w' \mid \underline{t}$ with the equivalence $\underline{\underline{\tau}} = \tau$.

Quantification is always done on "positive" types:

$M ::= \ldots \mid \forall x{:}w \leftrightarrow w'.M$

Now, we just need to modify *Composition*,

$$\text{Composition} \qquad \frac{\Gamma \vdash M_1 : w_1 \leftrightarrow \{l_i \Rightarrow \tau_i\}_{i=1}^n \qquad \Gamma \vdash M_2 : \{l_i \Rightarrow \underline{\tau_i}\}_{i=1}^n \leftrightarrow w_2}{\Gamma \vdash M_1; M_2 : w_1 \leftrightarrow w_2}$$

and add a new rule for input variables,

$$\text{Variable'} \qquad \Gamma[x \mapsto \tau] \vdash \underline{x} : \underline{\tau}$$

Proposition 8.4 *The simply typed functional calculus satisfies the subject reduction property.*

PROOF Follows Proposition 8.1 by checking the new cases. □

The interesting property of this new type system is that it lets one distinguish input from output: in a model, if we fix all inputs, then outputs are uniquely determined. On the other hand, we are not in a functional type system, so that it is not clear on which inputs depend outputs.

8.2.3 Generalized inversion

We define inversion on terms by:

$$\begin{array}{lcllcl} \underline{\{l \Rightarrow M\}} & = & \{l \Rightarrow \underline{M}\} & \underline{(\overline{M})} & = & \overline{(\underline{M})} \\ \underline{M_1; M_2} & = & \underline{M_1}; \underline{M_2} & \underline{\underline{M}} & = & M \\ \underline{\forall x.M} & = & \forall x.\underline{M} & & & \end{array}$$

A consequence of this definition is that, if M satisfies the integrity condition, and $\underline{M}$ too, then M is *linear*.

Definition 8.3 (linear,flat,oriented)
A term is linear *when for every x, there appears only one x and one $\underline{x}$.*
A term is flat *when it is of the form $\forall x_1 \dots x_m.\{l_1 \Rightarrow y_1, \dots\}; \{\overline{l'_1 \Rightarrow z_1, \dots}\}$.*
A term is oriented *when for the same x, x and $\underline{x}$ appear in conjugated streams.*

A very simple result about inversion on this calculus is the following one:

Proposition 8.5 *If M is linear, flat and oriented, then $M; \overline{\underline{M}}$ is the identity (on its domain).*

Chapter 9

Conclusion

The various systems presented in this thesis are an attempt to give some sort of "completeness" to the notion of Currying, by introducing commutation in it, extending it to the result, and using it on relations. In this perspective we can think that our calculi are, in some way, minimal for the properties they give.

In an early stage of the development we introduced streams, and parameterized all calculi on them. The goal of this distinction is to make clear what is the complexity of the calculus, and what is simply the complexity of stream handling. For selective λ-calculus and transformation calculus and transformation calculus, the complexity comes clearly for the stream system: chosing a simpler one, like one of those proposed in Section 2.1, is enough to make things simple. This is we think these calculi are intrinsically simple.

On the other hand, in some extensions of transformation calculus, or with symmetric transformation calculi, structural equivalences, independently of streams, become so numerous that they may look difficult to handle. However, here again we think that they are simply expressing properties of constructors, and as such are a factor of syntactic complexity, but not a semantic one, like shows the simplicity of the model for the logical symmetric transformation calculus.

We insist on this semantical simplicity to position these calculi are possible fundamental calculi for the study of various phenomena, ranging from a simpler definition of multi-argument functions to the description of state, compositional processes, or distributed computations.

9.1 Applications

Applications of these calculi can be found on different levels.

First, we can see selective λ-calculus as a tool for the analysis of lambda-calculus itself, like $\lambda\sigma$-calculus[ACCL91] may be. Good examples of such an use are the proofs given for Bohm's theorem or strong normalization. Encoding variables on labels, like in the second proof, and composition are enough to have explicit substitutions.

It can also play a role in functional programming, permiting a more generalized use of labeled parameters in function calls. This is a way to obtain a syntax both more concise, and more expressive. In particular, the properties of polymorphic typing permits its introduction into strongly typed curried functional languages, while improving greatly the information contained in types. Some methods of library searching using types as key may profit from this.

Other potential applications can be found in natural language analysis, where the commutation possibilities given by labels may reflect some phenomena. We are thinking

here of Montague grammars, and expect that it may avoid some ad hoc modifications made on lambda-calculus. The relation of symmetric transformations with the Lambek calculus [Kan92, Lam58] may also be interesting.

Second, we can see transformation calculus as a computational framework. The encoding of variables on labels is again interesting, since it gives a natural way to compile variables out, while keeping the abstraction given by their names. This model differs from the classical sequential model, since it makes no supposition about the structural stability of memory. It is closer to the dataflow model, a transformation taking data from its input to produce its output. We may even see it as a linear writing for dataflow programs, with facilities to hanlde states. However transformation calculus is not commited to a specific model, and the abstraction it gives should make it independent from them.

A very interesting point in the dataflow view of transformation calculus is the relation it gives between type an concurrency analysis. Essentially, the possibility of commutation between two transformations is linked with the independence of the labels they use. This is a reason why, even uncomplete, type inference is interesting.

Last, symmetric transformations give us a relational framework, in which one can encode specifications concerning distributed computation, in an higher order form. This suggests relations with concepts like concurrent object-oriented programming, where an object can both be seen as data, and as computation going on somewhere. The various. Other calculi where already proposed in this area, but ours is specific in that it supposes determinism of communication.

9.2 Related works

We relate selective λ-calculus and transformation calculus to existing works.

9.2.1 Selective λ-calculus

The idea of introducing labels in programming languages is not a new one. This has been done in two ways. The first one, that is common to nearly every modern languages, is records. It is present either explicitly, like in Pascal, C, ML, etc; or implicitly with association lists, methods... Formalization of this structure has been actively explored lately. This started with Cardelli [Car88], was later extended in a second order calculus [CW85], and resulted in a number of type inference systems to make it compatible with ML-style polymorphic type inference [Wan88, Sta88, JM88, Rem89], and a compilation method was given by Ohori in [Oho92], for an extension of λ-calculus containing labeled records.

On the other hand, the second use of labels, as keywords for parameter passing in functions, as it may be done in Common LISP [Ste84], ADA [Led81], or LIFE [AKP91], and its extension to currying, was still an unexplored field. The reason might be that it touches a more fundamental part of λ-calculus: applications and abstractions. We cannot now limit us to adding new structures to the calculus, but must attack it in its core. In fact some systems offer the same type of parameter passing possibilities without modifying the core [Lam88, ORH93], but they are based on an intuition of store, that is of bindings from names to values, which makes this a second parameterizing system, independent from application. To our knowledge, no typing system has been proposed for them.

Recently Dami proposed a system called HOP ("hierarchical objects with ports") [Dam92, Dam93, Dam94], whose basis is close to [Lam88], but has more similarities

with ours. Particularly it is based on a modification of abstraction and application mechanisms, and includes lambda-calculus in a very simple way. However it differs in that labels are limited to names, and after each application it is necessary to extract the result by a selection operation, which suppresses the problem of repeted labels at differents levels.

Finally last year Dzeng and Haynes proposed a typed reconstruction algorithm for Common Lisp style variable-arity procedures [DH94]. There again currying is absent, so that the problem is reduced to a variant of the typing of records.

9.2.2 Transformation calculus

Since transformation calculus only happens to be able to represent state, its origin is not to be found in this field. It is rather based on two independent threads of work. The first one is the Categorical Combinatory Logic [Cur93, Har89], in which composition and currying play a central role. The direction seems opposed: one encodes lambda-calculus into CCL (or its abstract machine version, the CAM [CCM87]), while transformation calculus extends lambda-calculus. But the intuition that algorithmicity can be found in the structures of the lambda-calculus itself is the same.

The second one is process calculi. Their use of names for communication is similar to the principle of the transformation calculus. In [Bou89], Boudol proposes the γ-calculus. The base is lambda-calculus, but applications express emissions of messages and abstractions their reception, while multiple terms can be evaluated simultaneously. Milner's π-calculus [Mil92] proceeds alike, and by labeling with names applications and abstractions, it allows the use of multiple channels. The fundamental difference with our calculus is that non-determinism of the receptor of a message make these calculi divergent, while our terms are syntactically sequenced in order to keep determinism.

A third might be Lamping's Unified Parameterization System [Lam88], which tries like us to encode state modifications into the parameter passing system. However his systems departs essentially from lambda-calculus, so that this last has to be encoded; and destructive overriding makes impossible to limit the effect of modifying variables like we do.

After these somewhat different directions, our claims makes necessary to look at the larger litterature concerning modelling of mutables in Algol, Lisp, and modern functional programming languages. Algol is the closest, since scope-free variables cannot be used out of their *life area*, like with Algol's stack discipline, where a variable cannot be exported out of its scope.

This subject starts with Landin's encoding of Algol 60 into the lambda calculus [Lan65]. Or rather, nothing starts, since the problem stays unsolved: "The semantics of *applicative expressions* can be specified formally without the recourse to a machine. [...] With *imperative applicative expressions* on the other hand it appears impossible to avoid specifying semantics in terms of a machine".

Later, to encompass the stack discipline, marked store models were developped [Gor79, MS76] but they had two problems: a lack of abstraction, and the existence of some pathological cases, described in [MS88], where equivalences in Algol are not provable in the model.

A first answer to this was Oles and Reynolds category-theoretic models [Ole85, Rey81]. The essential idea is to define blocks as functions that can be applied to a range of states with various shapes, but do not change their shapes. However, inside the block, state is temporarily extended with local variables. Thus, they do not appear in

its meaning. Our approach shares a lot with this view, since we syntactically "expand" and "shrink" our state when we create and delete a scope-free variable.

Another one is the Halpern-Meyer-Trakhtenbrot Store Model [HMT84, THM84], later refined by Meyer and Sieber [MS88], still based on stores, but using locally complete partial orders. It comes at last very close to full-abstraction, but fails in a 7^{th} example. Aknowledging the depth of the problem, Mason and Talcott even proposed an Operational Framework [MT92b, MT92a] to solve it out of denotational semantics.

Apart from this problem, there is interesting remark about the *Orthogonality of assignments and procedures in Algol* [WF93]. A theorem is enunciated, proving that normalization of an Algol program can be done in two phase, one using β and copy rules, and the other a simple stack machine. We do not obtain such a result, since we do not explicitly distinguish between imperative and functional features in the transformation calculus, but we can see the same kind of behaviour, first reducing higher order functions and eliminating composition, and then reducing β-redexes of ground types.

If we go out of the Algol tradition, we can forget about the stack discipline. As a result, most systems give a formalization of references. So does the λ_v-S-calculus [FF87, FF89] for Scheme, and a call-by-value reduction strategy. With effect inference [GL86, LG88, TJ92], restrictions on the reduction strategy can be reduced, and, for instance, parallelism can be introduced.

Still, we feel more concerned by systems going the other way, starting without a specific reduction strategy. There are a number of them, which enforce single-threadedness of variables by various typing disciplines [GH90, PW93, SRI91, Wad90a, Wad90b]. We can see an intuitive relation between the way scope-free variables are used and linear types, but still we are not relying to typing for single-threadedness.

We actually do it in a syntactical way. In that we are very close to λ_{var} [ORH93, CO94]. In fact, even the structures of the calculus have similarities: like us, they use the linear structure of spines to ensure single-threadedness. They have rules to propagate the values of mutable variables along the spine of a term, like does our structural equivalences for labeled arguments, and their **return**-elimination rule ((**return** $N) \triangleright \lambda x.M \to (\lambda x.M)N$, *cf.* [ORH93]) can be seen as a variant of $\downarrow$-elimination ($\downarrow; M \to M$) including value-passing. The essential difference is that, since we use the same mechanism for scope-free variables and value-passing, we obtain a more unified calculus. In particular, the fact they are encoding references means that they must do some kind of garbage collection (their **pure** construct) to convert a value obtained using mutable variables into a purely functional one. In the transformation calculus, since we explicitely delete variables, we do not need such an "impure" purifier.

9.3 Future works

Many problems concerning these systems are still unsolved or inexplored.

One of them is compilation. The method proposed by Ohori [Oho92] may work on typed selective λ-calculus, but transformation calculus is more complex. More generally, we would like to be able to compile these calculi in both sequential and concurrent models. For concurrent models, the object-oriented model may provide an even better basis than dataflow.

As long as typing is incomplete, there is some room left for further investigation in this direction. Particularly, the similarity in typing between transformations and their symmetrical extension makes it an interesting field of study.

We gave here some models. A more complete investigation would certainly be profitable, and open even new directions for further completeness of the calculi.

All these problem are hard enough not to expect a quick solution. But they are rich in learnings about what is currying in its full extent, and how it can connect syntax and semantics, by making easier to represent the latter in the former.

Appendix A

Other results

A.1 Applicative polymorphism

To introduce polymorphic types into transformation calculus (*cf.* Section 7.3), we used a technique different from the usual let-polymorphism [DM82]. We call it *applicative polymorphism*. Since it is quite general, we apply it here to the usual λ-calculus.

A.1.1 Terms and types

Terms are those of untyped λ-calculus.

$$M ::= x \mid \lambda x.M \mid M\,M$$

Types are the usual polymorphic ones (same as Damas-Milner).

u	$::=$	$u_1 \mid \ldots$	base types
α	$::=$	$\alpha_1 \mid \ldots$	type variables
τ	$::=$	$\alpha \mid u \mid \tau \to \tau$	monotypes
σ	$::=$	$\tau \mid \forall\alpha.\sigma$	polytypes

A.1.2 Let polymorphism

Damas-Milner polymorphism or *polymorhism a la ML.*

Terms are extended with a **let** construct:

$$M ::= \ldots \mid \text{let } x = M \text{ in } M$$

Typing rules are in Figure A.1

Type reconstruction

A well known result about this type system is the existence of principal typings, and of a type reconstruction algorithm which gives them.

Definition A.1 (principal) *In $\Gamma \vdash M : \tau$, τ (monotype) is* principal *if for all τ' (monotype) such that $\Gamma \vdash M : \tau'$, there is a substitution on type variables ρ for which $\rho(\tau) = \tau'$. (That is, τ is more generic than τ').*

Theorem A.1 *There is an algorithm $\mathcal{T}$ always terminating, such that for any pair (Γ, M) ($FV(M) \subset \mathcal{D}_\Gamma$), either $\mathcal{T}(\Gamma, M) = \tau$ and τ is principal for (Γ, M), or $\mathcal{T}(\Gamma, M) = \bot$ and M is untypable under Γ.*

Var $\quad \Gamma[x:\sigma] \vdash x:\sigma$

App $\quad \dfrac{\Gamma \vdash M:\theta \to \tau \quad \Gamma \vdash N:\theta}{\Gamma \vdash MN:\tau}$

Abs $\quad \dfrac{\Gamma[x:\theta] \vdash M:\tau}{\Gamma \vdash \lambda x.M:\theta \to \tau}$

Let $\quad \dfrac{\Gamma \vdash M:\sigma \quad \Gamma[x:\sigma] \vdash N:\sigma'}{\Gamma \vdash \text{let } x = M \text{ in } N:\sigma'}$

Gen $\quad \dfrac{\Gamma \vdash M:\sigma}{\Gamma \vdash M:\forall\alpha.\sigma} \quad \alpha \notin Var(\Gamma)$

Inst $\quad \dfrac{\Gamma \vdash M:\forall\alpha.\sigma}{\Gamma \vdash M:\sigma[\tau/\alpha]}$

Figure A.1: Inference rules for let polymorphism

Substitution equivalence

An interesting result about this system is that one can replace all let constructs by in-place substitution, and the principal type will not change.

Proposition A.1 *If $\mathcal{T}(\Gamma.M) = \tau$ then $\mathcal{T}(\Gamma, M') = \tau$, where M' is obtained from M by eliminating all* let*'s with* let $x = M$ in $N \longrightarrow K\ ([M/x]N)\ M$[1]*. $\Gamma \vdash M' : \tau$ can be proved without using* Let *nor* Generalization*.*

This proposition shows that *let*'s are doing what we expect: they introduce definitions without changing the resulting type. This is an essential difference with the usual way to do define x to be N in M: $(\lambda x.M)N$, which may introduce restrictions on the type (N loses its polymorphism).

A.1.3 Applicative polymorphism

The idea here is to do the same thing, without introducing any new construct. That is, we want at least $(\lambda x.M)N$ to behave as let $x = M$ in N in the Damas-Milner system, and even better if we can.

So, terms are those of the untyped λ-calculus, but we use polymorphic types.

Typing rules

We need more rules than in the Damas-Milner, but they have some symmetries.

Inference rules are in Figure A.2

The essential difference is in the structure of the type judgement. Together with the typing environment Γ, we have an applicative context A, which contains the list of the types of the terms M is applied to.

$$\Gamma, \mathrm{A} \vdash M : \sigma$$

means that in an environment Γ, when M is applied to the terms whose types are in A, the result of the application has type σ.

Of course, this system satisfies the subject reduction property.

[1] K is there to keep the eventual constraints in M.

Var $\quad \Gamma[x \mapsto \sigma], [] \vdash x : \sigma$

App $\quad \dfrac{\Gamma, [\sigma] :: \mathrm{A} \vdash M : \sigma' \qquad \Gamma, [] \vdash N : \sigma}{\Gamma, \mathrm{A} \vdash M\ N : \sigma'}$

Abs $\quad \dfrac{\Gamma[x \mapsto \sigma], \mathrm{A} \vdash M : \sigma'}{\Gamma, [\sigma] :: \mathrm{A} \vdash \lambda x.M : \sigma'}$

Discharge $\quad \dfrac{\Gamma, \mathrm{A} :: [\theta] \vdash M : \tau}{\Gamma, \mathrm{A} \vdash M : \theta \to \tau}$

Charge $\quad \dfrac{\Gamma, \mathrm{A} \vdash M : \theta \to \tau}{\Gamma, \mathrm{A} :: [\theta] \vdash M : \tau}$

Gen $\quad \dfrac{\Gamma, \mathrm{A} \vdash M : \sigma}{\Gamma, \mathrm{A} \vdash M : \forall \alpha.\sigma} \quad \alpha \notin Var(\Gamma) \cup Var(\mathrm{A})$

Inst $\quad \dfrac{\Gamma, \mathrm{A} \vdash M : \forall \alpha.\sigma}{\Gamma, \mathrm{A} \vdash M : \sigma[\tau/\alpha]}$

Inst' $\quad \dfrac{\Gamma, [\sigma[\tau/\alpha]] :: \mathrm{A} \vdash M : \sigma}{\Gamma, [\forall \alpha.\sigma] :: \mathrm{A} \vdash M : \sigma}$

Figure A.2: Inference rules for applicative polymorphism

Proposition A.2 *If* $\Gamma, \mathrm{A} \vdash M : \sigma$ *and* $M \to N$ *the* $\Gamma, \mathrm{A} \vdash N : \sigma$.

PROOF see the cases in Section 7.3 □

Inst' elimination

In fact, we can immediately remove one rule from our system, since *Inst'* is redundant. We only leave it to simplify proofs of type judgements.

Proposition A.3 *If* $\Gamma, [] \vdash M : \sigma$, *then this can be proved without using the rule* Inst'.

PROOF This comes from the fact we are using A linearly.

The only way to introduce a polymorphic type σ in A (looking from the bottom) is through *App*. The only reason we may want a type in A to lose its polymorphism is with *Charge*. In between, we do not duplicate types in A. This means that from the beginning we did not need σ to be polymorphic. We can change the occurence of *App* into

$$\dfrac{\dfrac{\Pi}{\Gamma, [\sigma[\tau/\alpha]] :: \mathrm{A} \vdash M : \sigma'} \qquad \dfrac{\Gamma, [] \vdash N : \forall \alpha.\sigma}{\Gamma, [] \vdash N : \sigma[\tau/\alpha]}}{\Gamma, \mathrm{A} \vdash M\ N : \sigma'}$$

and suppress *Inst'* in Π. □

Type reconstruction

Like for let-polymorphism, we have principal types and type reconstruction.

Theorem A.2 *There is an algorithm* $\mathcal{T}$ *always terminating, such that for any triple* (Γ, A, M) *(*$FV(M) \subset \mathcal{D}_\Gamma$*), either* $\mathcal{T}(\Gamma, \mathrm{A}, M) = \tau$ *and* τ *is principal for* (Γ, A, M)*, or* $\mathcal{T}(\Gamma, \mathrm{A}, M) = \bot$ *and* M *is untypable under* Γ.

- $\mathcal{T}(\Gamma[x \mapsto \sigma], [\sigma_1, \ldots, \sigma_n], x) = (\rho, \rho(\alpha))$
  where $\rho = \text{mgu}(\text{strip}(\sigma_n) \to \ldots \to \text{strip}(\sigma_1) \to \alpha = \text{strip}(\sigma))$, α fresh.
- $\mathcal{T}(\Gamma, [\sigma] :: \mathrm{A}, \lambda x.M) = \mathcal{T}(\Gamma[x \mapsto \sigma], \mathrm{A}, M)$
- $\mathcal{T}(\Gamma, [], \lambda x.M) = (\rho, \rho(\alpha) \to \tau)$ where $(\rho, \tau) = \mathcal{T}(\Gamma[x \mapsto \alpha], [], M)$, α fresh
- $\mathcal{T}(\Gamma, \mathrm{A}, M\ N) = (\rho' \circ \rho, \tau)$

$$\text{where } \begin{cases} (\rho, \theta) = \mathcal{T}(\Gamma, [], N) \\ \Gamma' = \rho(\Gamma),\ \mathrm{A}' = \rho(\mathrm{A}) \\ (\rho', \tau) = \mathcal{T}(\Gamma', [\text{protect}(Var(\Gamma') \cup Var(\mathrm{A}'), \theta)] :: \mathrm{A}', M) \end{cases}$$

- $\text{protect}(V, \theta) = \forall (Var(\theta) \setminus V).\theta$
- $\text{strip}(\forall \beta_1 \cdots \beta_n.\tau) = \tau[\alpha_1 \cdots \alpha_n / \beta_1 \cdots \beta_n]$, α_i fresh.

Figure A.3: Applicative type reconstruction

PROOF We use the same unification algorithm for monotypes as in the Hindley-Milner system. It is already proved to give the most general unifier of a set of monotypes.

The type reconstruction algorithm is in Figure A.3.

It does not put arbitrary constraints on types, so its result is the most generic possible. □

Substitution equivalence

The substitution equivalence we had for let-polymorphism does not stand anymore. We have no let's, so this would amount to an equivalence of typing before and after β-reduction. This is clearly false, since the term after β-reduction may have more polymorphism than before.

Example A.1 Let us see the typing of

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

Before reduction the principal type is (in an empty context) $\forall \alpha.\alpha \to \alpha \to \alpha$. Since λf is not on the same spine as $(\lambda x.\lambda y.x)$, it cannot inherit its polymorphism.

After reduction of the leftmost redex, we obtain $(\lambda f.\lambda x.\lambda y.f\ x\ (f\ y\ x))(\lambda x.\lambda y.x)$, and f becomes polymorphic, which gives us the type $\forall \alpha.\forall \beta.\alpha \to \beta \to \alpha$.

Appendix B

FIML

FIML is an experimental programming language based on the transformation calculus. The current version, V0.4, is implemented in Caml-light via a compiler into an optimised version of transformation combinators (*cf.* Section 6.6), which are then interpreted. We give here the documentation and examples available by anonymous FTP at `ftp://camille.is.s.u-tokyo.ac.jp/pub/fiml`.

B.1 FIML V0.4 Manual

Here is a short manual for the version 0.4. Contrasting with version 0.3, you can see the new syntax, and particularly the emphasis on reverse application, and abstraction-application mixing. Pooled references are another goodie.

B.1.1 Installation

You can install FIML V0.4 in two ways: using caml-light batch compiler, or the interactive toplevel. On Macintoshes, only the second option is available. Caml-light V0.5 or more is necessary.

For batch compilation, just do `make` in the distribution directory. You then get an executable `fiml`, which you can move anywhere in your execution path. Just remember that when you are loading programs into FIML, the default directory is the one you were before starting.

Using FIML on top of Caml-light's toplevel is no more complicated. Start CAML.

```
%camllight
>       Caml Light version 0.6


#
```

If this is the first time you use it, compile it by loading fimlm.ml.

```
#include"fimlm";;
- : unit = ()
...
- : unit = ()
```

You will have some warnings for fimlfun.ml.

To load compiled code, just use fiml.ml, or fimli.ml to load from the sources, and call the `fiml` function.

	Basics		
l	::=	$INT \mid IDENT \mid IDENT + INT$	labels
b	::=	$INT \mid true \mid false \mid "STRING"$	base values
	Patterns		
x	::=	$IDENT \mid _$	variable pattern
m	::=	$x \mid \langle l{:}m, \ldots\rangle \mid \langle l{:}m, \ldots \mid x\rangle$	stream
	$\mid$	$(c\ l{:}m \ldots) \mid (c\ l{:}m \ldots \mid x)$	constructors
	$\mid$	$(m$ `:-` $l \ldots) \mid (m$ `:-` $l \ldots$ `-o` $l \ldots)$	restricted labels
	$\mid$	$(m : t)$	type constraint
	Expressions		
M	::=	$l{:}m, \ldots; E$	function case
C	::=	$b \mid IDENT \mid (E)$	closed expression
	$\mid$	$\langle l{:}E', \ldots\rangle$	stream
	$\mid$	$\langle l{:}E', \ldots; E\rangle$	reverse application
E'	::=	$C \mid C\ l{:}C \ldots$	application
	$\mid$	$let[rec]m = E(\mid m = E) * inE$	let definition
	$\mid$	$E'oE$	composition
	$\mid$	$/l{:}E', \ldots$	stream
E	::=	$E' \mid \backslash M(\mid M)* \mid E; E$	open expression

Figure B.1: Syntax of FIML expressions

```
#include"fiml";;
- : unit = ()
...
- : unit = ()
#fiml();;
Bienvenue dans FIML-light, Version 0.4 !
An interpreter by J. Garrigue, April 1994.

#
```

If for any reason you want to go back to the CAML toplevel, use the `quit` function. You can go back to your original state with the `fiml_top` function.

```
#quit <> ;;
A tres bientot...
- : unit = ()
#sin 1.34;;
- : float = 0.973484541695
#fiml_top () ;;
#/973 ;;
it : <int> = <973>
```

B.1.2 Syntax

Expressions

The syntax of FIML expressions is gine in figure B.1. Some abbreviations and shortcuts are admitted.

Label abbreviations

- 'l:' may be ommited when it is '1'.
- 'p' is equivalent to '$p + 0$'. (Think of it as $p1$ if you look at chapter 3. Then '$p + 1$' becomes naturally $p2$, *etc.*...)

Short-cuts

- operators: `+, -, *, /, **, <#, >#, <=, >=, ||, &&, =, !=, ^ , ::`
- lists: `[A;...;D]` $\leftrightarrow$ `(cons A ...(cons D nil) ...)`,
- list matching: `[a;...;d|t]` $\leftrightarrow$ `(cons a ...(cons d t) ...)`,
  `[a;...;d]` $\leftrightarrow$ `(cons a ...(cons d nil) ...)`,
- selector macro: `C.p` $\leftrightarrow$ `<C;<l:x|_>; x>`
- if-then-else: `if ...then:C2 else:C3` $\leftrightarrow$ `(\ true; C1 | false; C2) ...`
- references: `new_ref(p): p <- E, set_ref(p): p ! r <- E, get_ref(p): p ! r`

In fact `let` is itself a macro for `<$:E1,...,$:En; \$:m1,...,$:mn; E>` (Type checking being done "in context", polymorphism is kept.)

Some other notations are equivalent:

- `E1 ; E2` and `E2 o E1`
- `<l1:E1,...>` and `(/l1:E1,...)`

";" may be omitted before "\" or "/", when not ambiguous.
e.g. : `/1\x/x = <1>;\x;<x> = <1>`.

B.1.3 Toplevel

You can evaluate either expressions or definitions. To give you an handle on the result, expressions alone are in fact interpreted as defining the identifier `it`.

E;; $\longrightarrow$ let it $= E$;;
let [rec] $m = E$ and ...;;

Type definitions use the following syntax:

`type [v;...]` $IDENT$ `= c <l:t,...>` $(|$ `c <l:t,...>` $)*$ `;;`

v	::=	$'a \mid @a \mid \$a$	generic, return and row variables
r	::=	$\$a \mid < l : t, \ldots > \mid < l : t \ldots ; \$a >$	row types
w	::=	$@a \mid r \mid c_0 \mid [v; \ldots] c_n$	return types
t	::=	$'a \mid w \mid r$ `-o` w	types

There is a number of toplevel pragmas:

`load"filename";;`	loads filename.fm
`#dialogue true ;;`	sets the dialogue mode: "E;;" is then interpreted as "it ; E;;"
`#set e ;;`	resets the value assigned to it
`#show e ;;`	shows the value of an expression
`#dialogue false ;;`	back to standard mode

B.1.4 Primitives and pragmas

- Operators:
  ```
  add,sub,mult,div,mod,pow :  <1:int,2:int> -o int
  lt,gt,le,ge :  <1:int,2:int> -o bool
  or,and* :  <1:bool,2:bool> -o bool
  eq,neq :  <1:@a,2:@a> -o bool
  concat :  <1:string,2:string> -o string
  ```
  (*) && exist only as infix

- I/Os:
  ```
  open_std :  <1:<>> -o <#std:<>>
  close_std :  <#std:<>> -o <>
  put :  <1:string,#std:<>> -o <#std:<>>
  get :  <#std:<>> -o <1:string,#std:<>>
  ```

- Pooled references:
  ```
  new_ref(p) :  <'a,p:['a;$b]pool> -o <$b ref,p:['a;$b]pool>
  set_ref(p) :  <$b ref,'a,p:['a;$b]pool> -o <p:['a;$b]pool>
  get_ref(p) :  <$b ref,p:['a;$b]pool> -o <'a,p:['a;$b]pool>
  ```

- Others:
  ```
  nil :  ['a]list = []
  cons :  <1:'a,2:['a]list> -o ['a]list
  quit :  <1:<>> -o 'a
  dummy* :  'a = <>
  ```
  (*) dummy is temporarily made available to deal with type-driven control mechanisms. Dangerous!

B.1.5 Error handling

- Syntax : gives you the line number and position of the error. May be erroneous.
- Type-checking : gives you the internal cause of the error. This may not always be enough. The line number is that of the end of the definition in which the error occured , not that of the error itself.

B.1.6 Programming examples

A number of sample programs are provided.

- `prelude.fm`
  Basic functions and transformations. Best to use it always.
- `prelude2.fm`
  A few other functions. Necessary for examples.fm.
- `unif.fm`
  A complete algorithm for unification, handling errors!
- `examples.fm`
  Other little examples.
- `types.fm`
  Some classical types.

- `combinators.fm`
  Fundamental combinators of the transformation calculus.

- `graph.fm`
  Some definitions to handle binary directed graphs using pooled references.

B.1.7 Bugs and future developments

The only known practical bug is erroneous line numbers in error messages. But there are many theoretical ones.

In fact, there are no simple solutions to many typing problems in FIML. It results in leaving a terrible hole as the dummy untyped constant. It lets you do anything you want, and may even result in an uncaught error if you use it to induce the type-checker into error.

Even not going as far as that, the current type system does not guarantee unicity of encapsulated states. It is possible to achieve it by a combination of existential and linear types, but prohibiting this would make the system heavier, and not so pleasant as an interactive toy.

Last, this is not really a bug, but the central position of state handling in this language suggests that it should at least have either a lazy or concurrent semantics. Otherwise, all this complication comparing it to ML has little meaning.

To conclude, FIML V0.5 should have a stronger type-checking, and use lazy evaluation. And, if the theory progresses enough, even some object-oriented features.

B.2 Sample FIML session

The best way to start with a programming language has always been a guided tour.

B.2.1 Starting with it

Simple and classical things: the toplevel of a functional programming language.

Into FIML

```
% fiml
Bienvenue dans FIML-light, Version 0.4 !
An interpreter by J. Garrigue, April 1994.

#
```

The "#" is the prompt. This may recall you of the CAML system.

Some values

```
#1 ;;
it : int = 1
#it+1 ;;
it : int = 2
#<1,"a",true> ;;
it : <int, string, bool> = <1, "a", true>
```

Like in CAML again, you end a query with ";;". Basically you evaluate expressions. The answer is of the form `name : type = value`, where name is the identifier defined in the process. By default this is `it`, and you can use it afterwards, as shown by the second query.

Functions

```
#\x; x ;;
it : <'a> -o 'a = <fun>
#\x; x+1 ;;
it : <int> -o int = <fun>
#it 3 ;;
it : int = 4
#\x,y; x+y ;;
it : <int, int> -o int = <fun>
#it 1;;
it : <int> -o int = <fun>
```

Functions are first class values, and they are polymorphically typed. Abstraction part starts with a "\", and end with a composing ";". Notice the flat typing for curried functions.

Definitions

```
#let x = 3 and y = "a";;
y : string = "a"
x : int = 3
#let I x = x;;
I : <'a> -o 'a = <fun>
#let rec fact n = if (n=0) then:1 else:(n * fact (n-1));;
fact : <int> -o int = <fun>
#let x = 4 in x ;;
it : int = 4
```

Use `let` and `let rec` to introduce respectively non-recursive and recursive definitions. Combined with `in` the definition is local.

B.2.2 New notations

Introducing fundamental concepts of the transformation calculus.

Application

```
#fact 3;;
it : int = 6
#<3;fact>;;
it : int = 6
#/3,5; \x,y; y-x ;;
it : int = 2
#/I; \x; <x 1,x "a"> ;;
it : <int, string> = <1, "a">
```

Of course application can be done with the usual notation, but one may write it postfix, using either of the stream notations. `<x;F>` is close to an usual mathematical notation for `F(x)`, while `"/x;F"` puts the emphasis on the abstraction *vs.* application symmetry. The last example shows that lambda-abstraction in context keeps polymorphism.

Streams

```
#let me = <name:"Garrigue", age:22> ;;
me : <age:int, name:string> = <age:22, name:"Garrigue">
#me.age ;;
it : int = 22
#me ; <surname:"Jacques"> ;;
it : <age:int, name:string, surname:string> =
        <age:22, name:"Garrigue", surname:"Jacques">
#me ; \age:x; /age:x+1 ;;
it : <age:int, name:string> = <age:23, name:"Garrigue">
```

What you may have thought were tuple in the beginning, are in fact *streams*. These are records with compositional properties. You may select a field, add a new one, or modify it by composing *transformations*.

Transformations

```
#\x; /x+1 ;;
it : <int> -o <int> = <fun>
#\x,y; /y,x ;;
it : <'a, 'b> -o <'b, 'a> = <fun>
#\a:x; /b:x ;;
it : <a:'a> -o <b:'a> = <fun>
#\p:_ ;;
it : <p:'a> -o <> = <fun>
```

Any function returning a stream may be viewed as a transformation. They may modify streams both in their values (eg. `age` above), and in their structure: switching, change in labels, erasure.

Composition and matching

```
#<1:1,3:2> o <1:3,2:4,4:5> ;;
it : <1:int, 2:int, 3:int, 4:int, 6:int>
   = <1:1, 2:3, 3:2, 4:4, 6:5>
#it; \1:_,3:_ ;;
it : <1:int, 2:int, 4:int> = <1:3, 2:4, 4:5>
#<a:4,b+1:"b+1"> o <a:true,b:"b">;;
it : <a:int, a+1:bool, b:string, b+1:string>
   = <a:4, a+1:true, b:"b", b+1:"b+1">
#it; \a:_,b:_;;
it : <a:bool, b:string> = <a:true, b:"b+1">
```

In the transformation calculus, application and abstraction are translated into stream composition and matching. Remark how indexes changes when streams are composed:

a value with label n appears at the n^{th} unocuppied position of the stream it is added to. Matching as a symmetrical effect. This applies to named labels too.

B.2.3 The dialogue mode

Not used to transformation composition? In this part we will use the dialogue mode to compose everything.

Entering dialogue mode

```
##dialogue true;;
##set <2,3> ;;
it : <int, int> = <2, 3>
#/a:1 ;;
it : <int, int, a:int> = <2, 3, a:1>
```

`#dialogue` and `#set` are two pragmas that respectively set the dialogue mode, and the value of the current stream (`it`!). In dialogue mode, when you enter an expression, it is automatically composed with the current stream, as if you had entered `it; E ;;`.

A stack machine in your lambda-calculus

```
#let tadd x y = /x+y and tsub x y = /x-y;;
tsub : <int, int> -o <int> = <fun>
tadd : <int, int> -o <int> = <fun>
##set /;;
it : <> = <>
#/3,4;;
it : <int, int> = <3, 4>
#tadd;;
it : <int> = <7>
#/4;;
it : <int, int> = <4, 7>
#tsub;;
it : <int> = <(-3)>
```

In dialogue mode FIML works exactly like a stack machine. You may put values on the stack through "/", and modify it using transformations.

Some classical definitions

```
#let dup x = /x,x and switch x y = /y,x ;;
switch : <'a, 'b> -o <'b, 'a> = <fun>
dup : <'a> -o <'a, 'a> = <fun>
#let switch' 2:x = /x and rot 3:x = /x ;;
rot : <3:'a> -o <'a> = <fun>
switch' : <2:'a> -o <'a> = <fun>
#/1,2; switch ;;
it : <int, int, int> = <2, 1, -3>
#switch' ;;
it : <int, int, int> = <1, 2, -3>
```

```
#rot ;;
it : <int, int, int> = <-3, 1, 2>
```

Basic stack operations are easily defined in terms of transformations. Remark that, thanks to numeric indexes, we can simplify `switch` into `switch'`, or naturally define `rot` as extracting the third element to put it in the first position.

B.2.4 Imperative features

Using transformations permits to write functional programs in an imperative style. This becomes even more interesting when functional evaluation is combined with imperative features.

I/O's

```
##set /;;
it : <> = <>
#put "Name: ";;
it : <#std:<>> -o <#std:<>> = <fun>
#get;;
it : <#std:<>> -o <string, #std:<>> = <fun>
#\x; put("Hello " ^ x ^ "!\n") ;;
it : <#std:<>> -o <#std:<>> = <fun>
##set open_std; it;;
Name: Jacques
Hello Jacques!
it : <#std:<>> = <#std:<>>
```

We are still in dialogue mode. We progressively construct a function using I/Os by composing transformation. Since they use the hidden label `#std`, it must finally be composed with `open_std` to execute.

Pooled references

```
#type ['a;'b]graph = L <'a> | N <'b ref,'b ref>;;
Type graph with constructors:
L : <'a> -o ['a; 'b]graph
N : <'b ref, 'b ref> -o ['a; 'b]graph
##set <dummy ; \(x : ['a; <a:<>>]pool) ; x>;;
it : <a:['a; <a:<>>]pool> = <a:<>>
#a <- L 1;;
it : <<a:<>> ref, a:[[int; 'a]graph; <a:<>>]pool> = <<1>, a:<>>
#let x = it.1;;
x : <a:<>> ref = <1>
#a <- N x x;;
it : <<a:<>> ref, <a:<>> ref, a:[[int; <a:<>>]graph;<a:<>>]pool>
   = <<2>, <1>, a:<>>
#a ! x <- L 2 ;;
it : <<a:<>> ref, <a:<>> ref, a:[[int; <a:<>>]graph;<a:<>>]pool>
   = <<2>, <1>, a:<>>
#a ! x ;;
```

```
it : <[int; <a:<>>]graph, [int; <a:<>>]graph, <a:<>> ref,
      a:[[int; <a:<>>]graph; <a:<>>]pool>
   = <(L 2), (N <1> <1>), <1>, a:<>>
```

Pooled references are a way to integrate dynamically created objects in the type framework of FIML. The abstract type `pool` takes two arguments: the type of the values it "contains", and a marker for generating references. At creation time (`#set...`), only the second one is fixed, the type for values being polymorphic. After that type is synthesized in the usual way.

Quiting dialogue mode

```
##dialogue false;;
#2;;
it : int = 2
#it+1;;
it : int = 3
```

Dialogue mode is a good way to experiment with transformations, but you can go back to usual program writing by the pragma `#dialogue false`.

B.2.5 Final remarks

```
#quit <> ;;
A tres bientot...
%
```

The official way to quit FIML is the `quit` function.

You have certainly noticed the very experimental status of this interpreter. There are – purposedly – holes in the type system, to facilitate a toy use. So, if you have a core dump, this is probably not a bug! For more details, see the manual.

B.3 Programming samples

B.3.1 prelude.fm

```
(* Standard prelude     *)

let I x = x ;;                  (* identity                    *)
let A f = I o f ;;              (* P->A functor                *)
let T f x = /f x ;;             (* raising A->P for 1-arg functions *)
let T2 f x y = /f x y ;;        (* -- for 1-2-arg functions    *)

let rec it_list *f = \ []               (* list iterator *)
        | [h|t]; f h; it_list f:f t ;;
let rec list_it *f = \ []               (* reverse order *)
        | [h|t]; list_it f:f t; f h ;;
let rec fold *f = \ [] ; /[]
        | [h|t] ; f h; \x ; fold f:f t ; T(cons x) ;;
let rec map *f = \ []; [] | [h|t]; (f h)::map f:f t ;;
let rec while *do *end = end;
```

```
        \ ok:false | ok:true; do ; while do:do end:end ;;
let rec repeat *f = \ 0 | n; f; repeat f:f (n-1);;

let length = A(it_list f:(\_;T(add 1)) 2:0) ;;
let rec append = \ [] ; I | [h|t], l ; h::append t l;;
let rec combine = \ [], [] ; [] | [h|t], [h'|t'] ; <h,h'>::combine t t' ;;
```

B.3.2 prelude2.fm

```
(* prelude2.fm *)

let B f g x = f(g x) ;;          (* function composition         *)
let C f x y = f y x ;;           (* arg-order exchanger          *)
let K x y = x ;;                 (* "K"                          *)
let S f g x = f x (g x) ;;       (* "S"                          *)
let W f x = f x x ;;             (* arg-copier                   *)

let K1 = \x ;;                   (* arg-killer                   *)

let Fst f <x|y> = <f x> o y ;;          (* first-applier        *)
let Snd f <2:x|y> = <2:f x> o y ;;      (* second-applier       *)
let Rst f <x|y> = <x> o f y ;;          (* rest-applier         *)
let map_fst *f = map f:(Fst f) ;;
let map_snd *f = map f:(Snd f) ;;
let distr_pair f <x,y> = <f x,f y> ;;
```

B.3.3 unif.fm

```
(* An all-in-one unification program            *)

type term = Sym <string,term list> | Var <int> ;;

let rec assoc_env x =
        \ env:[]; (Var x)
        | env:[<y,v>|t]; if (x=y) then:v else:(assoc_env env:t x)
;;

let rec subst n by:v =
        \ on:(Var m); if (m=n) then:v else:(Var m)
        | on:(Sym a l); Sym a (map f:(\x; subst n by:v on:x) l)
;;

let rec simplify =
        \ (Var a); assoc_env a
        | (Sym a l),*env; Sym a (map f:(simplify env:env) l)
;;

let rec not_occur n = \ on:(Var m); if (m=n) then:false else:true
        | on:(Sym a l); it_list
                f:(\ ok:true,v  /ok:not_occur n on:v
```

```
                   | ok:false,_ /ok:false) l ok:true
                \*ok; ok
;;

let rec add_env <n,v> *env = if (not_occur n on:v)
        then:<ok:true,env:<n,v>::map f:(\<x,y> /x,subst n by:v on:y) env>
        else:<ok:false,env:env>
;;

let rec unify *env <a,b> =
        /simplify env:env a,simplify env:env b,env:env;
        \ (Var a),b ; add_env <a,b>
        | a,(Var b) ; add_env <b,a>
        | (Sym a l),(Sym b m) ;
                if (a=b && length l = length m)
                then:<combine l m,ok:true;
                        it_list f:(\ ok:true ; unify | _ )>
                else:<ok:false>
;;
```

B.3.4 examples.fm

```
(* Various examples     *)

let curry f a b = f <a,b>;;             (* 2-arg currying *)

let gcd m n = <m:m,n:n;                 (* a stupid gcd *)
        while do:(\*m,*n /m:n,n:mod m n)
              end:(\n:0 /ok:false,n:0 | /ok:true)
        \*m,*n; m>
;;

let rec fibo = repeat f:(\m,n /m+n,m) 2:0 3:1; K ;;     (* short! *)

let rec fibo' n = (repeat n f:(\*m,*n /m:m+n,n:m) m:0 n:1).m ;;

let sigma = I o it_list f:(T2 add) 2:0;;

(* the standard quick_sort      *)
let partition test:f =
        it_list f:(\x; if (f x) then:(\l1:l /l1:x::l)
                        else:(\l2:l /l2:x::l)) l1:[] l2:[]
;;

let rec quick_sort *order = \[]; []
        | [h|t]; partition test:(order 2:h) t \*l1,*l2;
                quick_sort order:order l1 @ h::quick_sort order:order l2
;;
```

```
(* a "random" number generator  *)
let random n seed:x = let x = mod (x+6373) 23557 in <mod x n,seed:x>;;

let rand_list = repeat f:(T2 cons o random 1000) 2:[] ;;

(* lists *)
let rev = it_list f:(T2 cons) 2:[] ; I;;
```

B.3.5 types.fm

```
(* Standard types *)

type ['a;'b]pair = Pair <'a,'b> ;;

type ['a;'b]sum = Inl <'a> | Inr <'b> ;;

type 'a tree = Leaf | Node <'a,left:'a tree,right:'a tree> ;;


(* Matching on constructors

#let leaf = Node left:Leaf right:Leaf;;
leaf : <'a> -o 'a tree = <fun>
#let t = Node 1 left:(leaf 2) right:(leaf 3);;
t : int tree = (Node 1 left:(Node 2 left:Leaf right:Leaf)
                        right:(Node 3 left:Leaf right:Leaf))
#<t \(Node|r); r>;;
it : <int,left:int tree,right:int tree>
   = <1,left:(Node 2 left:Leaf right:Leaf),
        right:(Node 3 left:Leaf right:Leaf)>
#let left (Node left:x|_) = x;;
left : <'a tree> -o 'a tree = <fun>
#left t;;
it : int tree = (Node 2 left:Leaf right:Leaf)

*)
```

B.3.6 combinators.fm

```
(* Combinators for the transformation calculus  *)
let I x = x ;;
let K = \x ;;
let D x f = <f,f o x> ;;
let L_a x = <<a:x>> ;;
let L_1 x = <<1:x>> ;;
let L_2 x = <<2:x>> ;;
let L_3 x = <<3:x>> ;;
let E_a a:x = <x> ;;
let E_1 1:x = <x> ;;
let E_2 2:x = <x> ;;
```

```
let O = K K ;;
let P = K o D ;;
let U = P o E_2 ;;
let A = I o (U I) ;;
let R = P o L_1 ;;
let T = R o E_2 ;;
let B = R o T o A L_3 ;;
let S = E_2 o R o A L_3 o D <> o E_2 ;;
let W = D <> ;;
```

B.3.7 graph.fm

```
type ['a;'b]graph = L <'a> | N <'b ref,'b ref>;;
type 'a tree = Lf <'a> | Nd <'a tree,'a tree>;;

let rec mem x = \ []; false | [h|t]; if (x=h) then:true else:(mem x t);;

let rec (depth :- 1 seen a -o 1 seen a) seen:l x =
    if (mem x l) then:<seen:l,[]> else:<seen:x::l; a!x;
        \ (L v); /[v]
        | (N y z); depth y; depth z; T2 append> ;;

let rec depth_all x = a!x;
        \ (L v); /[v]
        | (N y z); depth_all y \l; depth_all z \l' /l@l' ;;

let rec tree_of_graph x = a!x;
        \ (L v); /Lf v
        | (N x y); tree_of_graph x \x; tree_of_graph y \y /Nd x y ;;

let rec graph_of_tree =
        \ (Lf v); a <- L v
        | (Nd x y); graph_of_tree x \x;
                    graph_of_tree y \y;
                    a <- N x y ;;

let open_a (x : ['a;<a:<>>]pool) = <a:x>;;
#show (open_std;put"\"open_a dummy\" to open the pool.\n";close_std);;
```

References

[Aba94] Martin Abadi. Baby Modula-3 and a theory of objects. *Journal of Functional Programming*, Vol. 4, No. 2, pp. 249–283, 1994.

[AC94] Roberto M. Amadio and Pierre-Louis Curien. Selected domains and lambda calculi. Rapport Technique 161, INRIA, Rocquencourt, France, March 1994.

[ACCL91] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Levy. Explicit substitutions. *Journal of Functional Programming*, Vol. 1, No. 4, pp. 375–416, October 1991.

[ADLR93] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the λ-calculus: Three years of communications without understanding. Electronic distribution, December 1993.

[AKG93a] Hassan Ait-Kaci and Jacques Garrigue. Label-selective λ-calculus. Research report 31, DEC Paris Research Laboratory, May 1993.

[AKG93b] Hassan Ait-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. In *Proc. of the Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 24–40, Bombay, India, 1993. Springer-Verlag LNCS 761.

[AKGar] Hassan Ait-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. *Theoretical Computer Science*, to appear.

[AKMng] Hassan Ait-Kaci and Kathleen Milsted. Concurrent label-selective λ-calculus. PRL research report, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, forthcoming.

[AKP91] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of LIFE. In Maluszynski and Wirsing, editors, *Proc. 3rd Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, pp. 255–274. Springer-Verlag, LNCS 528, August 1991.

[AKP93] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, Vol. 16, No. 3&4, pp. 195–234, July-August 1993.

[AL93] Andrea Asperti and Cosimo Laneve. Paths, computations and labels in the λ-calculus. In *Proc. of the IEEE Symposium on Rewriting Techniques and Applications*, pp. 152–167. Springer Verlag LNCS 690, 1993.

[Apo93] Maria Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 465–478, 1993.

[AR88] Norman Adams and Jonathan Rees. Object oriented programming in Scheme. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 277–288, 1988.

[ARS94] Lennart Augustsson, Mikael Rittri, and Dan Synek. On generating unique names. *Journal of Functional Programming*, Vol. 4, No. 1, pp. 117–123, January 1994.

[Asp92] Andrea Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, Vol. 2, No. 1, pp. 23–59, January 1992.

[Bar81] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, Holland, 1981.

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 81–93, 1990.

[BCL90] Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical Report LIENS-90-14, LIENS, July 1990.

[Bis94] Sandip K. Biswas. In-place updates in the presence of control operators. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 283–293, 1994.

[BKKS87] H. P. Barendregt, J. R. Kennaway, J. W. Klop, and M. R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, Vol. 75, pp. 191–231, 1987.

[Bou89] Gerard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *Proceedings of TAPSOFT '89*, pp. 149–161, Berlin, Germany, 1989. Springer-Verlag, LNCS 351.

[Bru72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, Vol. 34, pp. 381–392, 1972.

[Bru94] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, Vol. 4, No. 2, pp. 127–206, 1994.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, Vol. 76, pp. 138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, 1984).

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. ACM Symposium on Functional Programming and Computer Architectures*, pp. 273–280, 1989.

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, Vol. 8,, 1987.

[CF90] Erik Crank and Matthias Felleisen. Parameter-passing and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 233–244, 1990.

[CGL92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Proc. ACM Conference on LISP and Functional Programming*, June 1992.

[CGR92] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 139–150, 1992.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, Vol. 76, pp. 95–120, 1988.

[CHR91] P.-L. Curien, T. Hardin, and A. Rios. Normalisation forte du calcul des substitutions. Technical Report LIENS-91-16, DMI, Ecole Normale Superieure, Paris, France, November 1991.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *?*, 1940.

[CO94] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignments. In *Proc. of the International Conference on Theoretical Aspects of Computer Software*, pp. 347–363, 1994.

[Cur30] Haskell B. Curry. Grundlagen der kombinatorischen Logik. *Am. J. Math.*, Vol. 52, No. 3-4, pp. 509–536 and 789–834, 1930.

[Cur80] Haskell B. Curry. Some philosophical aspects of combinatory logic. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pp. 85–101, 1980.

[Cur91] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, Vol. 82, pp. 389–402, 1991.

[Cur93] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1993. First edition by Pitman, 1986.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, Vol. 17, No. 4, pp. 457–522, 1985.

[Dam92] Laurent Dami. HOP: Hierarchical objects with ports. In D. Tsichritzis, editor, *Object Frameworks*. Technical report, University of Geneva, 1992.

[Dam93] Laurent Dami. The HOP calculus. In D. Tsichritzis, editor, *Visual Objects*. Technical report, University of Geneva, 1993.

[Dam94] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Universite de Geneve, Faculte des Sciences Economiques et Sociales, Switzerland, 1994.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 151–160, 1990.

[DH94] Hsianlin Dzeng and Christopher T. Haynes. Type reconstruction for variable-arity procedures. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 239–249, 1994.

[Did48] Denis Diderot. *Jacques le fataliste et son maitre*. Delmas, Paris, 1948.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[DR93] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 296–306, 1993.

[DRW95] Catherine Dubois, Francois Rouaix, and Pierre Weis. Extensional polymorphism. In *Proc. ACM Symposium on Principles of Programming Languages*, 1995.

[FF87] Mathias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 314–325, 1987.

[FF89] M. Felleisen and D.P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, Vol. 69, pp. 243–287, 1989.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, Vol. 103, pp. 235–271, 1992.

[Fil89] Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Proc. of the Summer Conference on Category Theory and Computer Science*, pp. 224–249, September 1989.

[Fil92] Andrzej Filinski. Linear continuations. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 27–38, 1992.

[GAK93] Jacques Garrigue and Hassan Ait-Kaci. Typing of selective λ-calculus. Technical Report 93-1, University of Tokyo, Department of Information Science, 1993.

[GAK94] Jacques Garrigue and Hassan Ait-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 35–47, 1994.

[Gal91] Jean Gallier. Constructive logics. part I: A tutorial on proof systems and typed λ-calculi. Technical Report 8, DEC Paris Research Laboratory, May 1991.

[Gar92] Jacques Garrigue. Lambda-calcul a selection par labels. Rapport de D.E.A., Universite Paris VII, 1992.

[Gar93a] Jacques Garrigue. Introducing stateful objects in a transformation calculus. In *Proc. of the JSSST Workshop on Object-Oriented Computing*, Tokyo, March 1993. Iwanami Shoten.

[Gar93b] Jacques Garrigue. Lambda-calcul a selection par labels et calcul des transformations. Memoire de magistere MMFAI, Ecole Normale Superieure, Paris, 1993.

[Gar93c] Jacques Garrigue. Transformation calculus and its typing. In *Proc. of the workshop on Type Theory and its Applications to Computer Systems*, pp. 34–45. Kyoto University RIMS Lecture Notes 851, August 1993.

[Gar94] Jacques Garrigue. The transformation calculus. Technical Report 94-09, University of Tokyo, Department of Information Science, April 1994.

[GH90] J.C. Guzman and P. Hudak. Single threaded polymorphic calculus. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 333–343, 1990.

[GH93] Pietro Di Gianantonio and Furio Honsell. An abstract notion of application. In *Proc. of the International Conference on Typed Lambda Calculi and Applications*, pp. 124–138, Utrecht, The Netherlands, 1993.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 28–38, 1986.

[Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[GS89a] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, Vol. 67, pp. 203–260, 1989.

[GS89b] Jean H. Gallier and Wayne Snyder. Designing unification procedures using transformations: a survey. In Y. N. Moschovakis, editor, *Logic from Computer Science*, pp. 153–215. Springer-Verlag, 1989.

[Har89] Therese Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, Vol. 65, pp. 291–342, 1989.

[Har92a] Therese Hardin. Eta-conversion for the language of explicit substitutions. In *?*, 1992.

[Har92b] Therese Hardin. From categprical combinators to $\lambda\sigma$-calculi, a quest for confluence. Technical report, INRIA, 1992.

[Has94] Masahito Hasegawa. Contextual calculus, cartesian category and categorical data types. Master's thesis, Research Institute for Mathematical Sciences, Kyoto University, 1994.

[HK84] Paul Hudak and David Krans. A combinator-based compiler for a functional language. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 122–132, 1984.

[HMT84] J.Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 245–257, 1984.

[HMV93] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 15–25, 1993.

[HP90] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 131–142, 1990.

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

[Hue80] Gerard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, Vol. 27, No. 4, pp. 797–821, October 1980.

[Hue87] Gerard Huet. Deduction and computation. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987. NATO ASI Series, Vol. F36.

[HY94] Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 348–360, 1994.

[JD88] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 158–168, 1988.

[JG90] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 303–310, 1990.

[JM88] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 198–211, 1988.

[JW94] Suresh Jagannathan and Stephen Weeks. Analyzing stores and references in a parallel symbolic language. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 294–305, 1994.

[Kan92] Makoto Kanazawa. The Lambek calculus enriched with additional connectives. *Journal of Logic, Language, and Information*, Vol. 1, pp. 141–171, 1992.

[KW94] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 196–207, 1994.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, Vol. 65, pp. 154–170, 1958.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 316–326, 1988.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Vol. 6, No. 4, pp. 308–320, 1964.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, Vol. 8, No. 2-3, pp. 89–101 and 158–165, February 1965.

[Led81] Henry Ledgard. *ADA : An Introduction, Ada Reference Manual (July 1980)*. Springer-Verlag, 1981.

[Ler93] Xavier Leroy. Polymorphism by name for references and continuations. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 220–231, 1993.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 47–57, San Diego, California, 1988.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 235–246, 1992.

[LW90] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 291–302, 1990.

[Mac94] Ian Mackie. Lilac: a functional programming language based on linear logic. *Journal of Functional Programming*, Vol. 4, No. 4, pp. 395–433, October 1994.

[Mil90] Robin Milner. Functions as processes. Rapport de Recherche 1154, INRIA, Rocquencourt, France, February 1990.

[Mil92] Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, pp. 203–246. NATO ASI Series, Springer Verlag, 1992.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, Vol. 93, pp. 55–92, 1991.

[MS76] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[MS88] Albert R. Meyer and Kurt Sieber. Toward fully abstract semantics for local variables. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 191–203, 1988.

[MT89] Ian A. Mason and Carolyn L. Talcott. Axiomatising operational equivalence in the presence of side effects. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 284–303, Asilomar, California, 1989.

[MT92a] Ian A. Mason and Carolyn L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, Vol. 105, No. 2,, 1992.

[MT92b] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 186–197, 1992.

[Nis93] Shin-ya Nishizaki. Principal type in simply typed lambda calculus with first-class environment. JSSST Workshop on Functional Programming and Program Transformation, 1993.

[Ode90] Martin Odersky. How to make destructive updates less destructive. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 25–36, 1990.

[Ode94] Martin Odersky. A functional theory of local names. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 48–59, 1994.

[Oho92] Atsushi Ohori. A compilation method for ML-style polymorphic records. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 154–165, 1992.

[Ole85] F.J. Oles. Type algebras, functor categories, and block strcutures. In N. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 15, pp. 543–573. Cambridge University Press, 1985.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 43–56, 1993.

[OT93] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 171–184, 1993.

[PT93] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 299–312, 1993.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, Vol. 4, No. 2, pp. 207–247, 1994.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 71–84, 1993.

[Red88] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 289–297, 1988.

[Rem89] Didier Remy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 77–87, 1989.

[Rem91] Didier Remy. Type inference for records in a natural extension of ML. Technical Report 1431, INRIA, May 1991.

[Rem92] Didier Remy. Projective ML. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 66–75, 1992.

[Rey81] John C. Reynolds. The essence of ALGOL. In de Bakker and van Vliet, editors, *Proc. of the International Symposium on Algorithmic Languages*, pp. 345–372. North Holland, 1981.

[Rey82] John C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pp. 121–162. Cambridge Univesity Press, 1982.

[Rie93] J.G. Riecke. Delimiting the scope of effects. In *Proc. ACM Symposium on Functional Programming and Computer Architectures*, pp. 146–155, June 1993.

[Sat94] Masahiko Sato. A purely functional language with encapsulated assignments. In *Proc. of the International Conference on Theoretical Aspects of Computer Software*, pp. 179–202, 1994.

[SC94] A.V.S. Sastry and William Clinger. Parallel destructive updating in strict functional languages. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 263–272, 1994.

[Sch24] M. Schonfinkel. Uber die Bausteine der matematischen Logik. *Math. Annalen*, Vol. 92, pp. 305–316, 1924. An account by Heinrich Behmann of a lecture delivered in 1920.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hugues, editor, *Proc. ACM Symposium on Functional Programming and Computer Architectures*, pp. 192–214. Springer Verlag, 1991. LNCS 523.

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 88–97, 1988.

[Ste84] Guy L. Steele. *Common LISP : The Language*. Digital Press, 1984.

[THM84] B. A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. From denotational to operational and axiomatic semantics for ALGOL-like languages: An overview. In E. CLarke and D. Kozen, editors, *Logic of Programs*, pp. 474–500, Berlin, 1984. LNCS 164, Springer-Verlag.

[Tho89] Bent Thomsen. A calculus of higher order communicating systems. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 153–154, 1989.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, Vol. 2, No. 3, pp. 245–271, July 1992.

[W+90] Pierre Weiset al. *The CAML Reference Manual, version 2.6.1*. Projet Formel, INRIA-ENS, 1990.

[Wad90a] Philip Wadler. Comprehending monads. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 61–78, 1990.

[Wad90b] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

[Wan88] Mitchell Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, 1988.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, Vol. 93, pp. 1–15, 1991.

[WF93] S. Weeks and M. Felleisen. On the orthogonality of assignments and procedures in Algol. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 61–78, January 1993.

[YH90] Hirofumi Yokouchi and Teruo Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM Journal of Computing*, Vol. 19, No. 1, pp. 78–97, February 1990.