Typing of Selective λ-Calculus

Jacques Garrigue
The University of Tokyo
Department of Information Science
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japan
garrigue@is.s.u-tokyo.ac.jp

Hassan Ait Kaci
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison, France
hak@prl.dec.com

March 23, 1993

Abstract

Record calculi have recently been a very active field of research, but its reciprocal, i.e. the use of keywords in functions, is still ignored.

Selective λ-calculus is a conservative extension of lambda calculus which, by labeling abstractions and applications, enables some form of commutation between arguments. It is an enhancement for both clarity, thanks to labels, and power with the possibility to commute. We propose here a simply typed version of this calculus, and show that it extends to second order and polymorphic typing. For this last one there exists a most generic type, and we give the algorithm to find it.

This, combined with the fact selective λ-calculus extends naturally lambda calculus, giving numeric labels their intuitive meaning, provides us with a keyword extension for polymorphically typed languages like ML, compatible with the original syntax, where unlabeled abstractions and applications can be seen as labeled by 1.

1 Introduction

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 [5], was later extended in a second order calculus [6], and resulted in a number of type inference systems to make it compatible with ML-style polymorphic type inference [20, 18, 11, 17], and a compilation method was finally given by Ohori in [16], for an extension of λ-calculus containing labeled records. The problem is not completely solved, since the merge operator is still problematic (Ohori's calculus does not contain it), but becomes clearer.

On the other hand, the second use of labels, as keywords for parameter passing in functions, as it may be done in Common LISP [19], ADA [13], or LIFE [3], is 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 parameterizing possibilities without modifying the core [12, 15], 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.

Our goal is to have a system where we can apply a function on a list of labeled arguments, and be able to give only part of the arguments at once. We can call it an incremental (curried) parameter passing system with labels. What do we expect from it? First, to be incremental. We want to be able to write $f(p \Rightarrow a, q \Rightarrow b, r \Rightarrow c, ...)$ as $((f(p \Rightarrow a))(q \Rightarrow b))(r \Rightarrow c)...$ like we would do in ML without labels (we will omit parentheses). Then to provide us with commutativity of arguments. That is, for two labels p and q,

$$f(p \Rightarrow a, q \Rightarrow b) \simeq f(p \Rightarrow a)(q \Rightarrow b) = f(q \Rightarrow b)(p \Rightarrow a).$$

Since we include λ-abstraction in our system, we cannot suppose that we will always have different labels. That is, $f(p \Rightarrow a)(p \Rightarrow b)$ cannot be rewritten by commutation in order to preserve semantics, and it has still a meaning since f can be a function with some parameter labeled p, and giving back another function with a parameter labeled p.

This limit for commutation is absent if we think of numerical labels, seen as *relative* positions. They must satisfy two constraints. The first one is to respect our intuition of "numbers indicate positions". In clear we want to have

$$f(a, b, c, \ldots) = f(1 \Rightarrow a, 2 \Rightarrow b, 3 \Rightarrow c, \ldots)$$

It is important since it means that we can interpret unlabeled terms in our system. The second constraint is given by currying: we want $f(a, b, \ldots) \simeq f(a)(b) \ldots$. For labeled terms it means

$$f(1 \Rightarrow a, 2 \Rightarrow b, 3 \Rightarrow c, \ldots) \simeq f(1 \Rightarrow a)(1 \Rightarrow b)(1 \Rightarrow c) \ldots,$$

The change in labels comes from the fact *relative* positions changed. Intuitively in the left hand all arguments are directly acceded from f, and as such must receive different labels in order to be distinguished. But in the right hand b is only accessible after consumption of a, and becomes then the first argument, as does c after b is consumed.

We have only translated here unlabeled expressions, but once we have unique labels in a tuple, we can change the order freely, so that $f(1 \Rightarrow a, 2 \Rightarrow b, 3 \Rightarrow c) \simeq f(3 \Rightarrow c, 1 \Rightarrow a, 2 \Rightarrow b)$. In the right hand we mean that c should be consumed first. If we curry this expression, we obtain

$$f(3 \Rightarrow c, 1 \Rightarrow a, 2 \Rightarrow b) \simeq f(3 \Rightarrow c)(1 \Rightarrow a)(1 \Rightarrow b).$$

Since c is the last argument (highest label), currying does not affect relative positions of labels following it. After application on c, a is still the first argument and b the second. Similarly, if we want to apply successively b, c, a, we will write

$$f(2 \Rightarrow b, 3 \Rightarrow c, 1 \Rightarrow a) = f(2 \Rightarrow b)(2 \Rightarrow c)(1 \Rightarrow a)$$

After consumption of b, c becomes the second argument, and a is still the first.

Finally, if we think of all possible consumption orders, for three arguments we want all the following commutation equalities,

$$\begin{aligned} &f(1 \Rightarrow a)(1 \Rightarrow b)(1 \Rightarrow c) = f(2 \Rightarrow b)(1 \Rightarrow a)(1 \Rightarrow c) \\ &= f(2 \Rightarrow b)(2 \Rightarrow c)(1 \Rightarrow a) = f(3 \Rightarrow c)(2 \Rightarrow b)(1 \Rightarrow a) \\ &= f(3 \Rightarrow c)(1 \Rightarrow a)(1 \Rightarrow b) \end{aligned}$$

Uniqueness of result is preserved since we have different labelings too. We can define it more generally for any pair of numeric labels by,

$$\forall m > n \qquad f(m \Rightarrow a)(n \Rightarrow b) = f(n \Rightarrow b)(m - 1 \Rightarrow a)$$

which defines an equivalence on terms of our system.

Selective λ-calculus provides us with these equalities, about symbolic and numerical labels, and the symmetrical ones in respect to abstractions. As an untyped calculus, its confluence has already been proved, along with fundamental properties of λ-calculus like Böhm's theorem. We will give its full definition in Section 2.

In respect to types we can see it as the integration in the calculus of the natural isomorphism,

$$A \times B \simeq B \times A,$$

which, combined with currying,

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

gives us:

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

This becomes clearer when we think of indexed products like in category theory, with explicit projections π_1 and π_2, and write

$$\pi_1 \Rightarrow A \times \pi_2 \Rightarrow B \simeq \pi_2 \Rightarrow B \times \pi_1 \Rightarrow A,$$

or

$$\pi_1 \Rightarrow A \rightarrow (\pi_2 \Rightarrow B \rightarrow C) \simeq \pi_2 \Rightarrow B \rightarrow (\pi_1 \Rightarrow A \rightarrow C).$$

The object of this paper is to define a type system which reflects the preceding isomorphisms, which are part of those described in [4].

We start in Section 3 with an approach very close to classical λ-calculus, and provide simple types for selective λ-terms. The essential difference with classical simple types is that, in order to emphasize the intrinsic commutativity, we will put on the same level, types of arguments that may commute, and have non-commutative lists for those that may not. For instance, the $cons_{int}$ operator, namely $cons_{int}(car \Rightarrow h : \text{int}, cdr \Rightarrow t : \text{int list}) = (h :: t)$ for integer lists, should get type $(car \Rightarrow \text{int}, cdr \Rightarrow \text{int list}) \rightarrow \text{int list}$. Such a notation shows that it is possible to apply $cons_{int}$ on both labels car and cdr, and that the final (not abstracted) result is a list of integers.

Limitations of simple types being well-known, this type system is then, in Section 4, extended into a second order system, to enable parameterized types. We can then define a more general $cons$ operation by $cons[\alpha](car \Rightarrow h : \alpha, cdr \Rightarrow t : \alpha \text{ list}) = (h :: t)$ for lists of some type α, and it will get type $\forall\alpha.(car \Rightarrow \alpha, cdr \Rightarrow \alpha \text{ list}) \rightarrow \alpha \text{ list}$. Again the extension is essentially similar to that for classical λ-calculus, except that we must keep the leveled structure in types, after substitution of a type variable. Such a system may express many generic structures, but we will see that, outside of its complexity and verbosity, the distinction it makes between typing phases and application phases is somehow in contradiction with the intuition of selective λ-calculus.

The last step of this process is attained in Section 5, building a polymorphic typing system *à la* ML for selective λ-calculus. We can view it as a restriction of the second order system, which, enabling us to find a type inference algorithm, frees us of explicit typing. In short this means that we can integrate labeled parameters in any ML-like programming language and obtain a coherent result on both sides of syntax, since a classical λ-term may be interpreted as a selective one, and typing, by this algorithm. Going on with the preceding example, for the definition $cons(car \Rightarrow h, cdr \Rightarrow t) = (h :: t)$, we can infer the type $\forall\alpha.((car \Rightarrow \alpha, cdr \Rightarrow \alpha \text{ list}) \rightarrow \alpha \text{ list})$.

Such a type system seems particularly well adapted to selective λ-calculus, by the incremental aspect of typing, which is done together with application. It makes the commutation facility really transparent for functions, since one do not need to know how the function has been written, but which labeled function it represents, where we define labeled functions as a counterpart for normal forms in our system.

We tried to show the usefulness of such an extension in Section 6, with examples in an hypothetical ML-like language. Section 7 concludes.

2 Selective λ-calculus

2.1 Syntax

Selective λ-terms are formed by variables from a variable set $\mathcal{V}$ and two labeled constructors, abstraction and application.

The set of labels $\mathcal{L}$ is the union of numeric labels in $\mathcal{N} = \mathbf{N} \setminus \{0\}$ and symbolic labels $\mathcal{S}$. It is totally ordered, on $\mathcal{N}$ we have $<_{\mathcal{L}}=<_{\mathcal{N}}$, and $\forall(n,p) \in \mathcal{N} \times \mathcal{S}$, $n <_{\mathcal{L}} p$.

$$\mathcal{L} = \mathcal{N} \cup \mathcal{S}$$

We will denote variables by x, y, labels by p, q, restricting m, n to numerical ones, and λ-expressions by capitals.

Here is the syntax of selective λ-terms.

$$\begin{array}{rcll} M & ::= & x & \text{variables,} \\ & | & \lambda_p x.M & \text{abstraction,} \\ & | & M \,\widehat{}_p\, M' & \text{application.} \end{array}$$

We will say "to abstract x on p in M", "to apply M to M' through p". These terms will always be considered modulo α-conversion.

β – reduction

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

Symbolic reordering

(1) $\lambda_p x.\lambda_q y.M \to \lambda_q y.\lambda_p x.M$ $\quad p > q$

(2) $M\,\widehat{}_p\, N_1\,\widehat{}_q\, N_2 \to M\,\widehat{}_q\, N_2\,\widehat{}_p\, N_1$ $\quad p > q$

(3) $(\lambda_p x.M)\,\widehat{}_q\, N \to \lambda_p x.(M\,\widehat{}_q\, N)$ $\quad p \neq q, x \notin FV(N)$

Numeric reordering

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

(5) $M\,\widehat{}_m\, N_1\,\widehat{}_n\, N_2 \to M\,\widehat{}_n\, N_2\,\widehat{}_{\widehat{m-1}}\, N_1$ $\quad m > n$

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

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

Figure 1: Reduction rules for selective λ-calculus

We should precise here what is the intuition behind this syntax. It is clear enough for symbolic labels, which might be seen as channels (or stacks), where we can put with applications and get with abstractions, as we did in classical λ-calculus. But for numerics we use here relative labels. It would be more natural to use absolute ones. That is, to write $(\lambda(1 \Rightarrow x, 2 \Rightarrow y, 4 \Rightarrow z).M)\hat{}(1 \Rightarrow a, 4 \Rightarrow b)$ in place of $(\lambda_1 x.\lambda_1 y.\lambda_2 z.M)\,\widehat{}_1\, a\,\widehat{}_3\, b$ like relative labels impose. But we need it to get local rules. Just keep in mind the easy translation from growing absolute labels $(i_k < i_{k+1}, j_k < j_{k+1})$ to relative labels:

$$\begin{array}{c}(\lambda(i_1 \Rightarrow x_1, ..., i_k \Rightarrow x_k, ..., i_n \Rightarrow x_n).M)\\ \hat{}(j_1 \Rightarrow N_1, ..., j_k \Rightarrow N_k, ..., j_m \Rightarrow N_m)\\ \Updownarrow\\ (\lambda_{i_1} x_1 .....\lambda_{i_k - k+1} x_k .....\lambda_{i_n - n+1} x_n.M)\\ \widehat{}_{j_1}\, N_1 \cdots {}_{j_k}\widehat{}_{-k+1}\, N_k \cdots {}_{j_m}\widehat{}_{-m+1}\, N_m\end{array}$$

This translation justifies label modifications in reduction rules or types.

2.2 Reduction system

The reduction rules are given in figure 1. In (1)-(3) at least one of p or q should be symbolic, in (4)-(7), m and n are numeric. Rules (1) and (2) are directly translated into (4) and (5), with the appropriate changes in labels. Rule (3) has to be separated in (6) and (7) in order to distinguish cases where $m < n$ and $m > n$. Rules (1) and (4) are there for confluence. They are not necessary if we are not interested in abstracted results, since we can have applicators (pair label-argument,$\widehat{}_p M$) meet abstractors (pair label-binding variable,$\lambda_p x$) even without them. The mention $x \notin FV(N)$ in (3), (6) and (7) is not a limitation since x is binding, and may be renamed in $\mathcal{V} \setminus FV(N)$ by α-conversion.

We call *selective λ-calculus* the free combination of these rules. This calculus is meaningful, in that it is confluent.

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

PROOF in 12 pages. We will not produce it here. See [2] □

Proposition 2.2 *Rules (1)–(7) form a Noetherian system.*

2.3 Examples

We give here two examples of reductions, one using symbolic rules and the other numerical ones. Understanding these systems separately is enough to understand them together.

2.3.1 Symbolic

We suppose that $p < q < r < s$,

$$\begin{array}{ll} & (\lambda(p \Rightarrow x, q \Rightarrow y, r \Rightarrow z).M)(r \Rightarrow N_1, s \Rightarrow N_2, p \Rightarrow N_3) \\ \simeq & (\lambda_p x.\lambda_q y.\lambda_r z.M)\,\widehat{}_r\, N_1\,\widehat{}_s\, N_2\,\widehat{}_p\, N_3 \\ \to_3 & (\lambda_p x.((\lambda_q y.\lambda_r y.M)\,\widehat{}_r\, N_1))\,\widehat{}_s\, N_2\,\widehat{}_p\, N_3 \\ \to_2 & (\lambda_p x.((\lambda_q y.\lambda_r y.M)\,\widehat{}_r\, N_1))\,\widehat{}_p\, N_3\,\widehat{}_s\, N_2 \\ \to_\beta & (\lambda_q y.\lambda_r z.[N_3/x]M)\,\widehat{}_r\, N_1\,\widehat{}_s\, N_2 \\ \to_3 & (\lambda_q y.((\lambda_r z.[N_3/x]M)\,\widehat{}_r\, N_1))\,\widehat{}_s\, N_2 \\ \to_\beta & (\lambda_q y.([N_3/x][N_1/z]M))\,\widehat{}_s\, N_2) \\ \to_3 & \lambda_q y.([N_3/x][N_1/z]M\,\widehat{}_s\, N_2) \end{array}$$

2.3.2 Numerical

$$\begin{array}{ll} & (\lambda(2 \Rightarrow x, 1 \Rightarrow y, 4 \Rightarrow z).M)(4 \Rightarrow N_1, 6 \Rightarrow N_2, 2 \Rightarrow N_3) \\ \simeq & (\lambda_2 x.\lambda_1 y.\lambda_2 z.M)\,\widehat{}_4\, N_1\,\widehat{}_5\, N_2\,\widehat{}_2\, N_3 \\ \to_4 & (\lambda_1 y.\lambda_1 x.\lambda_2 z.M)\,\widehat{}_4\, N_1\,\widehat{}_5\, N_2\,\widehat{}_2\, N_3 \\ \to_7 & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\,\widehat{}_3\, N_1))\,\widehat{}_5\, N_2\,\widehat{}_2\, n_3 \\ \to_5 & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\,\widehat{}_3\, N_1))\,\widehat{}_2\, N_3\,\widehat{}_4\, N_2 \\ \to_7 & (\lambda_1 y.\lambda_1 x.((\lambda_2 z.M)\,\widehat{}_2\, N_1))\,\widehat{}_2\, N_3\,\widehat{}_4\, N_2 \\ \to_\beta & (\lambda_1 y.\lambda_1 x.[N_1/z]M)\,\widehat{}_2\, N_3\,\widehat{}_4\, N_2 \\ \to_7 & (\lambda_1 y.((\lambda_1 x.[N_1/z]M)\,\widehat{}_1\, N_3))\,\widehat{}_4\, N_2 \\ \to_\beta & (\lambda_1 y.[N_3/x][N_1/z]M)\,\widehat{}_4\, N_2 \\ \to_7 & \lambda_1 y.([N_3/x][N_1/z]M\,\widehat{}_3\, N_2) \end{array}$$

3 Simply typed calculus

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.

3.1 Types

We define our types by a grammar; labels are generated by

$$p ::= n \mid s$$

with n on $\mathcal{N}$ and s on $\mathcal{S}$.

We distinguish between base types,

$$u ::= u_1 \mid u_2 \mid \ldots$$

and general types,

$$t ::= u \mid (n \Rightarrow t, ..., s \Rightarrow t^+, ...) \to u$$

where t^+ is a non-nil list of types, and labels n or s should be strictly growing in the enumeration. For numeric labels we adopt the absolute notation, which explains the absence of repetition.

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. Which makes type inference quite intuitive.

3.2 Typed terms

The original syntax of terms is extended in

$$M ::= x \mid \lambda_p x : t.M \mid M\,\widehat{}_p\, M'.$$

which requires any abstracted variable to be explicitly typed.

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

$$\frac{\Gamma[x \mapsto \theta] \vdash M : \tau \qquad A_p(\tau') = \tau \qquad T_p(\tau') = \theta}{\Gamma \vdash \lambda_p x{:}\theta.M : \tau'} \tag{II}$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : T_p(\tau)}{\Gamma \vdash M \,\widehat{p}\, N : A_p(\tau)} \tag{III}$$

Figure 2: Typing rules for the simply typed calculus

$$\begin{array}{rcl}
\tau & = & (\nu_1 \Rightarrow \tau_1, ..., \nu_i \Rightarrow \tau_i, ..., \nu_n \Rightarrow \tau_n, \\
 & & \sigma_1 \Rightarrow \tau_1^+, ..., \sigma_i \Rightarrow \tau_i^+, ..., \sigma_m \Rightarrow \tau_m^+) \rightarrow \upsilon \\
\hline
A_{\nu_i}(\tau) & = & (\nu_1 \Rightarrow \tau_1, ..., \nu_{i+1} - 1 \Rightarrow \tau_{i+1}, ..., \nu_n - 1 \Rightarrow \tau_n, \\
 & & \sigma_1 \Rightarrow \tau_1^+, ..., \sigma_m \Rightarrow \tau_m^+) \rightarrow \upsilon \\
T_{\nu_i}(\tau) & = & \tau_i \\
A_{\sigma_i}(\tau) & = & (\nu_1 \Rightarrow \tau_1, ..., \nu_n \Rightarrow \tau_n, \\
 & & \sigma_1 \Rightarrow \tau_1^+, ..., \sigma_i \Rightarrow \tau_{i2}^+ ... \tau_{il_i}^+, ..., \sigma_m \Rightarrow \tau_m^+) \rightarrow \upsilon \\
T_{\sigma_i}(\tau) & = & \tau_{i1}^+
\end{array}$$

Figure 3: Definition of A_p and T_p

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.

3.3 Typing rules

We write type judgements of the form

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

where Γ is a mapping from a subset of $\mathcal{V}$ to types, to express that M has type τ in the typing context Γ.

Figure 2 gives the inference rules for type judgements on simply typed selective λ-calculus, using definitions of A_p (resulting type after application on p) and T_p (type accepted on p) in figure 3. Here again the translation between absolute and relative labels explains the structure of $A_{\nu_i}(\tau)$. Rule (II) uses the fact that knowing $A_p(\tau)$ and $T_p(\tau)$ defines completely τ.

This type system, of course provides us with the fundamental property of typing, subject reduction (stability of typing).

Proposition 3.1 *If $\Gamma \vdash M : \tau$ and $M \rightarrow N$ then $\Gamma \vdash N : \tau$.*

3.4 Strong normalization

The strong normalization was a consequence of the addition of types to classical λ-calculus. If the selective version has same properties, it should be obtained too.

Theorem 3.2 *The simply typed selective λ-calculus is strongly normalizing.*

PROOF We already know that the calculus is confluent, by the Church-Rosser theorem for untyped selective λ-calculus. We only need termination.

A preliminary remark is that proving that the number of β-steps in the reduction has an upper bound is enough, since we know that the system without β is Noetherian even for the untyped calculus.

We associate to each subterm a function that calculates an upper bound for the maximal number of β-steps to reduce it in function of its arguments. For instance, for $\lambda_p f:((r \Rightarrow aa) \rightarrow a).\lambda_q x:a.f \,\widehat{r}\, x \,\widehat{r}\, x$, the associated function is $\lambda_p f.\lambda_q x.(f(r \Rightarrow x, r \Rightarrow x) + 2 \times x + 2) : (p \Rightarrow (r \Rightarrow \omega\omega) \rightarrow \omega, q \Rightarrow \omega) \rightarrow \omega$. One can see at first glance that such a function has same order (depth of type) as the original term. And since a function can only be applied to a function of inferior order, we get a structure of calculation like this when we look for a result: $f_4(g_3, h_3, i_1, j_0) = a \times g_3(c_2, c_0) + b \times h_3(c_2) + d \times i_1(c_0) + j_0 + e$, where indices indicate order, and a, b, d, e are integer coefficients. So that we are sure to obtain a finite tree, whose depth is the order of the original term.

It means that if for some terms such a function is defined and gives finite results, then we can get, for any term combining them by applications, an upper bound.

It gives us the induction step to prove this property.

We construct function schemes for this function. A function scheme has not to be well typed: this is done when we transform it into a real function. The induction is on the size of terms.

1. For a single variable x of type τ, the function scheme is a free variable x' with corresponding type τ' (every base type is replaced by ω).

2. For an abstracted term $\lambda_p x:\tau.M$, by induction hypothesis we have a function scheme f for M. The function scheme is $\lambda_p x':\tau'.f$.

3. For an application $M \,\widehat{p}\, N$, by induction hypothesis we have f for M and g for N. The function scheme is $f \,\widehat{p}\, g + 1 + g$. The 1 is for the β-reduction corresponding to this application, and the g if N is evaluated before substitution in M.

We transform function schemes into functions by the following rules:

- we replace all variables that are still free by zeros or zero functions (according to the expected type).
- all insufficiently applied functions are applied to zeros or zero function (according to the expected type).
- right hands of "+" are typed as integers. They should be added to the final result of the left hand (necessarily typed ω, since this is the only base type here).

This gives us the function $f : (p_1 \Rightarrow \tau_1, ...) \rightarrow \omega$ that we were looking for. By induction its results are finite.

We have again an upper bound for the longest reduction path by applying this function to zeros or zero functions. □

4 Second order selective λ-calculus

Girard in [9] (quoted by [10]) proposed a second order λ-calculus, in which types can contain variables and be handled as terms. We adapted this to selective λ-calculus, in the most straightforward way, adding the same syntactic and type constructs.

For this, types are extended in

$$v ::= \alpha \mid \beta \mid \ldots$$

$$w ::= u \mid v \mid \forall v.t$$

$$t ::= w \mid (n \Rightarrow t, ..., s \Rightarrow t^+, ...) \rightarrow w$$

where v represents variables and w return types (types that can be found on the right side of an arrow).

The new syntax adds two new constructs to handle types,

$$M ::= x \mid t \mid \lambda_p x:t.M \mid \Lambda v.M \mid M \,\widehat{p}\, M' \mid M \cdot M'$$

$\Lambda v.M$ is type abstraction, and $M \cdot M'$ is type application. In our straight-forwardness, they do not mix up in reordering.

Allowing commutation between these constructs and previous abstractions and applications is possible. We could define $\Lambda v.M$ as $\lambda_T v:T.N$ and $M \cdot M'$ as $M \,\widehat{T}\, M'$, giving highest priority to type instantiation by changing the order in $\forall p \in \mathcal{L}, T <_{\mathcal{L}} p$. But type inference becomes more complex, since some terms which, by any reduction, give well-typed terms in a finite number of steps, cannot be typed by simple inference rules (the sort used for second order λ-calculus).

4.1 Reduction rules

We add a new version of β-reduction, for types:

$$(\beta_T) \qquad (\Lambda\alpha.M) \cdot N \to M[\alpha \backslash N]$$

4.2 Type normalization

We will not modify inference rules, but a problem appears for substitutions, when we substitute type variables with types. In the syntax of types, a return type w may only be a base type, a type variable or a generalized type, but not any type.

The problem can be solved generally by normalizing types during substitutions. We give here the algorithm to do that.

We proceed recursively from the right, starting from

$$\begin{array}{l}((\ldots, \nu_n \Rightarrow \tau_n, \ldots, \sigma_m \Rightarrow \tau_m^+) \to v) \\ \qquad [v \backslash ((\ldots, \nu'_{n'} \Rightarrow \tau'_{n'}, \ldots, \sigma'_{m'} \Rightarrow \tau'^+_{m'}) \to w)]\end{array}$$

- We first substitute v in all τ_i, τ_i^+, so that it appears only once, as final return type.
- If $\sigma'_{m'} = \sigma_k$ we just concatenate τ_k^+ and $\tau'^+_{m'}$ under σ_k and suppress $\sigma'_{m'}$ in the substitution.

  $$\begin{array}{l}((\ldots, \sigma_k \Rightarrow \tau_k^+ \tau'^+_{m'}, \ldots) \to v) \\ \qquad [v \backslash ((\ldots, \sigma'_{m'-1} \Rightarrow \tau'^+_{m'-1}) \to w)]\end{array}$$

  If there is no σ_i equal to $\sigma'_{m'}$ then there is k such that $\sigma_k < \sigma'_{m'} (< \sigma_{k+1})$. We transfer the label too.

  $$\begin{array}{l}((\ldots, \sigma_k \Rightarrow \tau_k^+, \sigma'_{m'} \Rightarrow \tau'^+_{m'}, \ldots) \to v) \\ \qquad [v \backslash ((\ldots, \sigma'_{m'-1} \Rightarrow \tau'^+_{m'-1}) \to w)]\end{array}$$

  Repeat this step until there is no symbolic label.
- For numerical labels we need a count of unused labels. $ff(i)$ is the minimal integer such that $[1, ff(i)]$ contains $ff(i) - i$ labels of $(\vec{\nu_n})$ (that is the i^{th} unused numerical label.) The transfer is then unique, towards $ff(\nu'_{n'})$.

  $$\begin{array}{l}((\ldots, \nu_k \Rightarrow \tau_k, ff(\nu'_{n'}) \Rightarrow \tau'_{n'}, \ldots) \to v) \\ \qquad [v \backslash ((\ldots, \nu'_{n'-1} \Rightarrow \tau'_{n'-1}) \to w)]\end{array}$$

  Repeat this step until there is no label.
- Then we just substitute the return type, which is structurally correct.

  $$((\ldots) \to v)[v \backslash w] = ((\ldots) \to w)$$

The correctness of this algorithm is based on the absolute-relative label translation.

4.3 Typing rules

The inference rules for typing judgements are given in figure 4. The three new rules are IV: Type introduction, V: Generalization and VI: Instantiation. T is the type of types. These rules are syntactically similar to rules for classical lambda calculus, the difference being in the sus-cited substitution algorithm for (VI).

Proposition 4.1 *If $\Gamma \vdash M : \tau$ in the second order typed selective λ-calculus, and $M \to N$, then $\Gamma \vdash N : \tau$.*

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

$$\frac{\Gamma \vdash \theta : T \quad \Gamma[x \mapsto \theta] \vdash M : \tau \quad A_p(\tau') = \tau}{\Gamma \vdash \lambda_p x.M : \tau' \quad T_p(\tau') = \theta} \tag{II}$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : T_p(\tau)}{\Gamma \vdash M \,\widehat{p}\, N : A_p(\tau)} \tag{III}$$

$$\Gamma \vdash \tau : T \quad (\tau \text{ is a well formed type}) \tag{IV}$$

$$\frac{\Gamma[\alpha \mapsto T] \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau} \tag{V}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\tau \quad \Gamma \vdash \sigma : T}{\Gamma \vdash M \cdot \sigma : \tau[\alpha \backslash \sigma]} \tag{VI}$$

Figure 4: Typing rules for second order calculus

Theorem 4.2 *The second order selective λ-calculus is strongly normalizing.*

PROOF Strong normalization of simply-typed selective λ-calculus and second order λ-calculus. □

Now we should go back and see why, if we consider Λ as λ_T, and "$\cdot$" as $\widehat{T}$, there are terms that should be typable, and have no type in this system. An example is

$$(\lambda_T \alpha : T.\lambda_p x : \text{int}.\lambda_q y : ((1 \Rightarrow \text{int}) \to \alpha).y \,\widehat{1}\, x) \,\widehat{p}\, 2 \,\widehat{T}\, \text{int}$$

In the typing system we defined, $\widehat{T}\,\text{int}$, being behind $\widehat{p}\,2$, cannot be reduced by rule (VI), giving no type to this term. However, the abstraction on p being independent of α, this is perfectly equivalent to

$$(\lambda_T \alpha : T.(\lambda_p x : \text{int}.\lambda_q y : ((1 \Rightarrow \text{int}) \to \alpha).y \,\widehat{1}\, x) \,\widehat{p}\, 2) \,\widehat{T}\, \text{int}$$

which is typable. We could of course extend rule (VI) for commutation, but then we have the inverse problem. That is

$$(\lambda_T \alpha : T.\lambda_p x : \text{int}.\lambda_q y : ((1 \Rightarrow \text{int}) \to \alpha).y \,\widehat{1}\, x) \,\widehat{q}\, I_{\text{int}} \,\widehat{T}\, \text{int}$$

has type $(p \Rightarrow \text{int}) \to \text{int}$, whereas

$$(\lambda_T \alpha : T.\lambda_p x : \text{int}.\lambda_q y : ((1 \Rightarrow \text{int}) \to \alpha).y \,\widehat{1}\, x) \,\widehat{q}\, I_{\text{int}}$$

is untypable. We might have subterms of typable terms which would be untypable. Prudence makes us refuse such an eventuality.

Nonetheless, the system we defined is powerful, since we can instantiate types independently during a calculation. It will prove useful. But it is cumbersome if we do not need all this power. And it makes us loose the freedom of selective λ-calculus, by enforcing a typing order independent of the application order. This is only a problem in the (rare) case when one would want to instantiate only partially a term's type, before applying it partially.

5 Polymorphic selective λ-calculus

Whereas a second order typing system might be used in programming languages, the ML trend for typing is the more simple (because more restrictive) polymorphism. That is to put all type quantifiers outside of the type itself, and instantiate types 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.

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

$$\frac{\Gamma[x \mapsto \theta] \vdash M : \tau \quad A_p(\tau') = \tau \quad T_p(\tau') = \theta}{\Gamma \vdash \lambda_p x.M : \tau'} \tag{II}$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : T_p(\tau)}{\Gamma \vdash M \,\widehat{}_p\, N : A_p(\tau)} \tag{III}$$

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

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

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

Figure 5: Typing rules for polymorphic selective λ-calculus

5.1 Syntax and types

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

$$M ::= x \mid \lambda_p x.M \mid M \,\widehat{}_p\, M' \mid \text{let } x = M \text{ in } M'$$

Like in Damas and Milner's definition [7] types are divided into monotypes and polytypes. Monotypes are ranged by t in

$$w ::= u \mid v$$

$$t ::= w \mid (n \Rightarrow t, ..., s \Rightarrow t^+, ...) \rightarrow w$$

where u stands for base types and v for type variables, and polytypes by σ in

$$\sigma ::= t \mid \forall v.\sigma$$

5.2 Typing rules

The typing rules are given in figure 5. The rules (IV)-(VI) are in no way specific to selective λ-calculus. In fact, since type quantifiers are external they are independent from the structure of monotypes. Their roles are IV: Generalize, V: Instantiate and VI:Let introduction.

Proposition 5.1 *If $\Gamma \vdash M : \tau$ in polymorphically typed selective λ-calculus, and $M \rightarrow N$, then $\Gamma \vdash N : \tau$.*

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). As for second order selective λ-calculus, we need the special algorithm of substitution in 4.2. Which 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 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 5.2 *Polymorphic selective λ-calculus is strongly normalizing.*

5.3 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 (cf 4.2). That is, we have a good form of E-unification [8], where the equivalence relation on terms can be expressed by an oriented rewriting system.

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

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

$$\text{Type structure} \quad \frac{\phi \wedge u = (p_1 \Rightarrow \tau_1, ...) \rightarrow \theta}{\bot}(u \text{ base type})$$

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

$$\text{Decomposition} \quad \frac{\phi \wedge (\nu_1 \Rightarrow \tau_1, ..., \sigma_1 \Rightarrow \tau_{11} ... \tau_{1l_1}, ...) \rightarrow \theta = (\nu_1 \Rightarrow \tau'_1, ..., \sigma_1 \Rightarrow \tau'_{11} ... \tau'_{1l_1}, ...) \rightarrow \theta'}{\phi \wedge \tau_1 = \tau'_1 \wedge ... \wedge \tau_{11} = \tau'_{11} \wedge ... \wedge \tau_{1l_1} = \tau'_{1l_1} \wedge ... \wedge \theta = \theta'}$$

Completion I, II, III,

$$\frac{\phi \wedge (\nu_1 \Rightarrow \tau_1, ..., \nu_i \Rightarrow \tau_i, ..., \sigma_1 \Rightarrow \tau_1^+, ...) \rightarrow \theta = (\nu_1 \Rightarrow \tau'_1, ..., \nu'_i \Rightarrow \tau'_i, ..., \sigma'_1 \Rightarrow \tau_1'^+, ...) \rightarrow \theta'}{\phi \wedge (\ldots) \rightarrow \theta = (..., \nu_i \Rightarrow \tau_i, \nu'_i \Rightarrow \tau'_i, ...) \rightarrow \alpha \wedge \theta' = (\nu_i + 1 - i \Rightarrow \tau_i) \rightarrow \alpha},$$

$$(\nu_i < \nu'_i, \alpha \text{ fresh})$$

$$\frac{\phi \wedge (\nu_1 \Rightarrow \tau_1, ..., \sigma_1 \Rightarrow \tau_1^{(l_1)}, ..., \sigma_i \Rightarrow \tau_i^+, ...) \rightarrow \theta = (\nu_1 \Rightarrow \tau'_1, ..., \sigma_1 \Rightarrow \tau_1'^{(l_1)}, ..., \sigma'_i \Rightarrow \tau_i'^+, ...) \rightarrow \theta'}{\phi \wedge (\ldots) \rightarrow \theta = (..., \sigma_i \Rightarrow \tau_i^+, \sigma'_i \Rightarrow \tau_i'^+, ...) \rightarrow \alpha \wedge \theta' = (\sigma_i \Rightarrow \tau_i^+) \rightarrow \alpha},$$

$$(\sigma_i <_{\mathcal{L}} \sigma_i, \alpha \text{ fresh})$$

$$\frac{\phi \wedge (\nu_1 \Rightarrow \tau_1, ..., \sigma_1 \Rightarrow \tau_1^{(l_1)}, ..., \sigma_i \Rightarrow \tau_{i1} ... \tau_{il_i}, ...) \rightarrow \theta = (\nu_1 \Rightarrow \tau'_1, ..., \sigma_1 \Rightarrow \tau_1'^{(l_1)}, ..., \sigma_i \Rightarrow \tau'_{i1} ... \tau'_{il'_i}, ...) \rightarrow \theta'}{\phi \wedge (\ldots) \rightarrow \theta = (..., \sigma_i \Rightarrow \tau'_{i1} ... \tau'_{il'_i} \tau_{i,l'_i+1} ... \tau_{il_i}) \rightarrow \alpha \wedge \theta' = (\sigma_i \Rightarrow \tau_{i,l'_i+1} ... \tau_{il_i}) \rightarrow \alpha},$$

$$(l'_i < l_i, \alpha \text{ fresh})$$

Figure 6: Rewriting rules for type unification

Theorem 5.3 *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 6. 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, τ^+ for type lists, $\tau^{(l)}$ being a l-uple of types.

This rewriting system has a strongly normalizing strategy, up to equalized or new variables substitution.

To prove this we will show that it doesn't change the semantics of the equation system, and terminates representing a unifier. We use model semantics, where we view the system as the set of all possible type assignments for variables originally present.

The first three rules 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.

Redundancy suppresses meaningless equations. Combinated with elimination it suppresses repeated equations too. *Elimination* substitutes variables (using type normalization), while keeping their referent. Conditions are there to enforce termination.

Decomposition takes two types with same (first level) structure, and identifies their subcomponents. But to get types to the same structure we need *completion*. There are three versions to apply successively. The equation must be between functional types. The first rule is for missing labels in the numeric part. We take the first missing one and add it to the type, introduce a new type variable, and identify the result type (hopefully a type variable) with the difference. $\nu_i + 1 - i$ is for going back from the absolute label to the relative one. 2 and 3 proceed in the same way, for missing symbolic label, or insufficient number of types on a symbolic label.

Clearly none of these rules changes the semantics.

For termination, first we can remark that this system without decomposition and completion is terminating. A variable is solved when it appears only once, and as side of an equation. After elimination, α is solved. The side conditions guarantee that a solved variable will stay solved, even if we have a reversible equation, like $\alpha = \beta$. And once it is solved elimination cannot apply on its equation.

To add decomposition and completion we define a measure by the lexicographic order on *(unsolved variables,sum of sizes)*, where the size of a type is the total number of base types, variable occurrences, and type constructors (i.e. "$\rightarrow$") it contains (recursively).

We can verify that each rule reduces this measure. True of course for failures. True for redundancy, for at least size. Unsolved variables for elimination. Size for decomposition: some variables may be solved, but none created.

Completion must be restrained. We start completion only on a system irreducible for preceding rules, and once we chose an equation, we go on with completion on this equation, completing label differences from left to right, only eliminating or failing with the newly introduced equation between each step. Since they are return types, θ and θ' are either base types or variables. If θ' is a base type, the new equation causes a failure. If it is a variable, it is immediately eliminated, and the number of unsolved variables does not change. Completion adding only labels that were in the other type, it can happen only a limited number of times consecutively on an equation. After the last time, we have still the same number of unsolved variables. We can now decompose, and solve the last introduced variable with θ. So that on the whole we have, in a finite number of steps, reduced the number of unsolved variables.

Last we should show that an irreducible conjunction for these rules represents an unifier. All the equations have form $\alpha = \theta$ with α solved and θ some type. All variables are either solved either unconstrained, so that we can make an unifier σ of this list, by $\sigma(\alpha) = \theta$ for each equation. We can suppress solved new variables.

This unifier is the most general one, since it fully translates the relevant part of the equation system. □

This algorithm may look complex. This impression is due to *completion* rules, which are its backbone. It would be much shorter if we work in a little more abstract way, introducing the notion of free-record type. That is the sequence present on the left side of an arrow.

$$r ::= (n \Rightarrow t, \ldots, s \Rightarrow t^+, \ldots)$$

$$
\begin{array}{rrl}
Tp(\Gamma,\phi) = & x \mapsto & (\text{match } \Gamma(x) \text{ with} \\
 & & \quad \forall(fv).\tau \rightarrow (\phi, NV(fv,\tau)) \\
 & & \quad \mid \tau \rightarrow (\phi,\tau)) \\
\mid & \lambda_p x.M \mapsto & \text{let } (\phi',\tau) = Tp(\Gamma[x \mapsto \alpha],\phi,M) \\
 & & \text{in } (\phi', ((p \Rightarrow \alpha) \rightarrow \beta)[\beta \backslash \tau]) \\
\mid & M \mathbin{\widehat{}}_p M' \mapsto & \text{let } (\phi',\tau) = Tp(\Gamma,\phi,M) \text{ in} \\
 & & \text{let } (\phi'',\theta) = Tp(\Gamma,\phi',M') \text{ in} \\
 & & (\phi'' \wedge (\tau = (p \Rightarrow \theta) \rightarrow \alpha), \alpha) \\
\mid & \text{let } x = M \text{ in } M' \mapsto & \\
 & & \text{let } (\phi',\tau) = Tp(\Gamma,\phi,M) \text{ in} \\
 & & \text{let } fv = FV(\tau) \setminus FV(\Gamma) \text{ in} \\
 & & Tp(\Gamma[x \mapsto \forall(fv).\tau],\phi',M')
\end{array}
$$

Figure 7: Type inference algorithm

where we keep the same constraint of growing labels. We call it free-record since, in contrast with classical records, a label may be assigned not only one, but a list of values.

In fact these free-record types form a monoid structure, where we even have the uniqueness of quotient. We define $r = r_1 \cdot r_2$ as $r \rightarrow w = (r_1 \rightarrow \alpha)[\alpha \backslash (r_2 \rightarrow w)]$, using the algorithm of 4.2; and we can show that r_1 is the only solution to $r = X \cdot r_2$, and r_2 the only one for $r = r_1 \cdot X$. We can even define a generic prefix division, with quotient q and rest r, where r is the shortest free-record type such that, for $r_1 \div r_2$, $r_1 \cdot r = r_2 \cdot q$. We remark that the quotient of $r_1 \div r_2$ is the rest of $r_2 \div r_1$, so that we write $r_1 \div r_2$ for this quotient, $r_2 \div r_1$ being the rest. Then we can replace all completion rules by only one

$$
\text{Comp} \quad \frac{\phi \wedge r_1 \rightarrow \theta = r_2 \rightarrow \theta'}{\begin{array}{c}\phi \wedge r_1 \rightarrow \theta = r_2 \rightarrow \theta' \wedge \\ \theta = (r_2 \div r_1) \rightarrow \alpha \wedge \theta' = (r_1 \div r_2) \rightarrow \alpha\end{array}}
$$

In fact this is the existence of this monoid with quotient that explains the existence of an unification algorithm; and to implement it we only need to define this quotient.

5.4 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 7.

The principal function of this algorithm, Tp, takes a typing environment Γ (bindings from variable names to types), an initial equation system, ϕ, and a selective λ-term, to give back a couple (new equation system,type of the term). We suppose that equation systems are always normalized, as well as Γ and the returned type in consequence.

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. "\" is set subtraction.

We will not prove this algorithm, since it is the same as for classical λ-calculus, except for the new unification, and the use of type normalization for the abstraction case. That is, correctness and completeness of this algorithm are only dependent on correctness and completeness of type unification we proved above.

6 Application to languages

This type synthesis algorithm have been written in CAML for the type checker of an hypothetical language. We use the CAML syntax [21], modified according to the basic syntax in figure 8, where pairs label-variable or label-value are noted $p:v$, c represents constants, x variables, m matching patterns, M matching cases, E open expressions and C closed expressions. We can easily add other constructs to this syntax.

$$
\begin{array}{lcl}
p & ::= & n \mid s \\
m & ::= & c \mid x \\
M & ::= & p{:}m \ldots \rightarrow E \\
C & ::= & c \mid x \mid (E) \\
E & ::= & C \mid \text{fun } M(\mid M)^* \mid C\ p{:}C \ldots \mid \text{let } m = E \text{ in } E
\end{array}
$$

Figure 8: A simple ML-like language with labels

To make the language more intuitive, the `fun` construct uses absolute labels. Symmetrically we use absolute labels for application too, which means that `(f 1:x 2:y)` is no longer equivalent to `((f 1:x) 2:y)`. We can of course write `((f 1:x) 1:y)` which means that we have only lost *formal* associativity, but this is one reason we need to distinguish closed and open expressions in the syntax.

When labels are not specified we assume that their value is the actual position of the argument. For instance `(f a b)` is `(f 1:a 2:b)`, `(f env:e a)` is `(f env:e 1:a)`, `(f 1:a b)` is `(f 1:a 2:b)`, and we can give a meaning to every combination by intermediate interpretations like `(f a 1:b)` being `(f 1:a 2:b)`, *etc.*..The idea is just to interpret from left to right, shifting values when a numeric position is already occupied.

Here are some little examples of computed types.

```
#fun car:a cdr:b -> a::b;;
 (car:'a cdr:'a list -> 'a list)

#let rec append = fun l1:[] l2:l -> l
#      | l1:[h|t] l2:l -> h::append l1:t l2:l;;
 append : (l1:'a list l2:'a list -> 'a list)

#let cons a b = a::b;;
 cons : (1:'a 2:'a list -> 'a 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:(cons 23);;
 (1:int list list -> int list list)

#map f:(cons 2:[1;2]);;
 (1:int list -> int list list)
```

You can see here two advantages of this system: types are more expressive, a good choice of keywords documents the function; and we can change application order with numeric labels. In the last case it would have needed a combinator.

At the same time a default appears. Types become more specific, and `map` could not be applied on `(append l2:[1;2])` for instance. But this can be solved by introducing a meta language for relabeling, where we could write `map f:(#{1:l1} append l2:[1;2])`, which is anyway more understandable than `map (C append [1;2])`, and more proper than `map (fun l -> append l [1;2])`, if not shorter.

The preceding examples were only functions with two arguments. In such a case one may doubt about the necessity of labels. When the order of arguments is clear enough, the risk of error is low. Still some two-argument functions are not so clear. 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 the second. One could still argue that a quick glance at the type suppresses the ambiguity. But this is not always true, and 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)
```

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. Which means that we must avoid anarchic use of labels, and promote a standardized one.

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 can even try to give types to control structures, like `if ... then ... else ...`, and reintroduce the practical notion of macros we had in Lisp:

```
value if : (1:bool else:'a then:'a -> 'a)
```

which can be used as `if` (*condition*) `then:` (*case true*) `else:` (*case false*). Of course this is not a function like others, since we want a different reduction strategy. But we can still profit from commutation, since we only need to have arguments for `else` and `then`, but not the condition, at compile time: we easily define a lazy pair, where arguments are calculated on request.

7 Conclusion and further work

We have proposed here three typing systems for selective λ-calculus: simple types, second order and polymorphic. The polymorphic one seems to give the best integration with the calculus. Integrated in a functional programming language with currying, it should be a powerful tool, extending currying facilities and helping to memorize multi-argument functions.

In presenting here a second order calculus, we have two reasons. First, polymorphism can be seen as a restriction of second order, and it uses results obtained for second order, like type normalization. The second one is that there are things impossible with polymorphism, like using different type instances of an argument. It is not a very disturbing limitation in a functional use of selective λ-calculus, but it proved fatal when encoding transformation calculus, an extension with some imperative features, in selective λ-calculus, whereas second order was still working.

An interesting subject is how to mix record operations and selective λ-calculus. The idea comes from the natural encoding of free records in the untyped calculus, as

$$\{l_1 \Rightarrow a_1, \ldots, l_n \Rightarrow a_n\} \longrightarrow \lambda_{sel} s.(s\ \widehat{l_1}\ a_1 \ldots \widehat{l_n}\ a_n)$$

where s should be a function selecting a label and discarding the others individually (we have no way to discard them globally), like $\lambda_{l_1} x_1 . \ldots . \lambda_{l_n} x_n . x_k$. We can even have function using more than one label. This is in fact the basic idea for transformation calculus. Still there are differences between this definition of free records, which allows label repetitions, and classical records, which do not. And the problem of type inference is only partially solved.

Another application of this calculus might be found in parallel processing. If we now see labels on a stream as identifying threads, the commutation capability directly interprets a concurrent evaluation. This is an idea very close to the dataflow paradigm, but we hope to replace flow analysis by type synthesis. Another, but not contradictory, view is to see labels as names, like for process communication. It shows a link, which can easily be made more evident, with calculi like Milner's π-calculus [14]. The conjunction of those two views seems an interesting prospective.

The last, but more immediate, problem, and a capital one, is compilation. Two different versions of selective λ-calculus using de Bruijn indices, through explicit substitutions [1], have been developed. They reflect two different levels of compilation: the semantic, where we keep labels; and the material, where they can be replaced by simple integer positions on a stack. This might be the basis for an efficient compilation method, which should, this is a characteristic of selective λ-calculus, be built on a completely curried vision. That is, there should be no overhead due to currying, except maybe a few low level operations.

Aknowledgements

The authors wish to thank particulary Atsushi Ohori for his comments and suggestions.

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[2] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus. Technical report, DEC Paris Reasearch Laboratory, 1993.

[3] Hassan Aït-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)*, pages 255–274. Springer-Verlag, LNCS 528, August 1991.

[4] Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical report LIENS-90-14, LIENS, July 1990.

[5] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, 1984).

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):457–522, 1985.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[8] Jean H. Gallier and Wayne Snyder. Designing unification procedures using transformations : a survey. In Piergiorgio Odiffredi, editor, *Logic and Computer Science*. Academic Press, 1990.

[9] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse d'etat, Université Paris VII, 1972.

[10] Gérard Huet. Deduction and computation. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987.

[11] L. A. Jategaonkar and J. C. Mitchell. Ml with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, 1988.

[12] John Lamping. A unified system of parameterization for programming languages. In *Proc. ACM Conference on LISP and Functional Programming*, pages 316–326, 1988.

[13] Henry Ledgard. *ADA : An Introduction, Ada Reference Manual (July 1980)*. Springer-Verlag, 1981.

[14] Robin Milner. The polyadic π-calculus: A tutorial. LFCS Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1991.

[15] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, 1993.

[16] Atsushi Ohori. A compilation method for ML-style polymorphic records. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[17] Didier Rémy. Typechecking records and variants in a natural extension of ml. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 77–87, 1989.

[18] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[19] Guy L. Steele. *Common LISP : The Language*. Digital Press, 1984.

[20] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, 1988.

[21] Pierre Weis et al. *The CAML Reference Manual, version 2.6.1.* Projet Formel, INRIA-ENS, 1990.