The Typed Polymorphic Label-Selective λ-Calculus

Jacques Garrigue
garrigue@is.s.u-tokyo.ac.jp

Department of Information Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japan

Hassan Aït-Kaci
hak@prl.dec.com

Digital Equipment Corporation
Paris Research Laboratory
85 Avenue Victor Hugo
92500 Rueil-Malmaison, France

Abstract. Formal calculi of record structures have recently been a focus of active research. However, scarcely anyone has studied formally the dual notion—*i.e.*, argument-passing to functions by keywords, and its harmonization with currying. We have. Recently, we introduced the label-selective λ-calculus, a conservative extension of λ-calculus that uses a labeling of abstractions and applications to perform unordered currying. In other words, it enables some form of commutation between arguments. This improves program legibility, thanks to the presence of labels, and efficiency, thanks to argument commuting. In this paper, we propose a simply typed version of the calculus, then extend it to one with ML-like polymorphic types. For the latter calculus, we establish the existence of principal types and we give an algorithm to compute them. Thanks to the fact that label-selective λ-calculus is a conservative extension of λ-calculus by adding numeric labels to stand for argument positions, its polymorphic typing provides us with a keyword argument-passing extension of ML obviating the need of records. In this context, conventional ML syntax can be seen as a restriction of the more general keyword-oriented syntax limited to using only implicit positions instead of keywords.

1 Introduction

The use of symbolic labels in programming languages is not new. This has been done in two ways. The first one, common to nearly all languages, is as field designators in record structures. Relatively recently, formalisms for records have been proposed. This started with Cardelli [6], was later extended to a second order calculus [7], and was followed by a number of record-type inference systems compatible with ML-style polymorphic type inference [22, 20, 13, 19]. Even more recently, a compilation method was proposed by Ohori [18], for an extension of λ-calculus containing polymorphically typed records.

Another way to use labels in programming languages has been as keywords for parameter-passing in procedure or function calls. This is the case in Common LISP [21], ADA [15], and LIFE [4]. However, in Common LISP or ADA, currying is not supported, which makes the situation rather mild. Although currying is supported in LIFE, even with keywords given in a different order, it is restricted nonetheless and does not accommodate implicit positions as it should. Indeed, fully flexible currying with the presence of keywords as well as explicit and implicit positions was until recently a still unexplored issue. Some proposals do offer this convenience of parameter-passing without modifying the core calculus [14, 17]. However, these are based on using a notion of store; that is, bindings from names to values. This introduces another parameterizing system, independent from λ-calculus. Even so, to our knowledge, no typing system has been proposed for them.

Our own proposal, as originally reported in [2], is to support this new convenience of labeling arguments directly in λ-calculus and accommodate selective unordered currying through commutation of arguments. In our view, the role of arguments is determined by their labels, which interact with their order.

Selective λ-calculus introduces two types of commutations. The first, and most immediate, is between symbolic labels. By analogy with tuples, when currying an expression $f(p \Rightarrow a, q \Rightarrow b, \ldots)$ we obtain an expression $((f(p \Rightarrow a))(q \Rightarrow b))(\ldots)$. But since there is no reason to apply f in this specific order, using the freedom provided by labels allows to curry in a different order; *e.g.*, $((f(q \Rightarrow b))(p \Rightarrow a))(\ldots)$. Suppressing superfluous parentheses, and limiting our consideration to two arguments, we obtain that the following equality must hold in our calculus:

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

However, this is true under a restriction: p and q must be distinct labels. Successive applications on the same label must not commute. Indeed, if the labels are equal, the order of these applications must be obeyed to be unambiguous.

© 1994 ACM. Appeared in the Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1994, pp. 35–47.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Here is an example of the use of these symbolic labels for the list constructor, in an ML-like language, together with inferred types.[1]

```
#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 commutation 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 applied the equality used for symbolic labels, we would obtain $f(1 \Rightarrow a)(2 \Rightarrow b) = f(2 \Rightarrow b)(1 \Rightarrow a)$. But, since it is better to see unary application as implicitly using the label 1 and keep conventional currying, we would rather write $f(1 \Rightarrow a)(1 \Rightarrow b)$, or simply, $f\ a\ b$ as usual. To make this possible, we must define commutation differently on numbers: namely, $f(2 \Rightarrow b)(1 \Rightarrow a) = f(1 \Rightarrow a)(1 \Rightarrow b)$. This can be generalized as:

$$f(m \Rightarrow a)(n \Rightarrow b) = f(n \Rightarrow b)(m-1 \Rightarrow a) \quad \text{if} \quad m > n.$$

For instance we can use it as follows, (omitting explicitly labeling with `1=>`):

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

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

This second commutation 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 relative precedence order of input on a given *channel*.

Selective λ-calculus provides the above equalities for symbolic and numerical labels for both application and abstraction. As an untyped calculus, its confluence has been established [3], along with fundamental properties of λ-calculus like Böhm's theorem [12].

Similarly, the introduction of label-selective types providing simple types for selective λ-terms is done in the same manner as that of simple types in classical λ-calculus. The essential difference is that, in order to emphasize the intrinsic commutativity, we will put on the same level all argument types to a function. 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 *car* and *cdr* labels, and that the result is a list of integers.

Then we build a polymorphic typing system *à la* ML for selective λ-calculus. As for ML-style polymorphism, a type inference algorithm exists, which obviates the need for explicit typing. In other words, this means that we can integrate labeled parameters in any ML-like programming language. Continuing with the previous 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 is particularly well-adapted to selective λ-calculus, thanks to the incrementality of typing, which goes together with application. On the other hand a second order type system, separating type application, would limit commutation possibilities by introducing new dependencies between abstractions.

Section 2 gives a practical and theoretical motivation for our type system. We then define symbolic and numerical label-selective λ-calculus in Section 3 and 4, combining them in a product system in Section 5. Sections 6 and 7 present respectively simple typing and polymorphic typing of the selective λ-calculus. To avoid cluttering the casual reader's attention with unnecessary details, we have relegated all proofs to the appendix.

2 Motivation

The calculus we present has practical and theoretical motivations. In practice, the use of labels for argument selection enhances clarity and obviates the need of argument-shuffling combinators. From a theoretical perspective, the commutation laws of labeled arguments readily render natural type isomorphisms in λ-calculus.

2.1 Keywords: an enhancement for clarity

We start here by giving some examples of how the use of keywords, and their appearance in types, may help the programmer. Our view is already partially proven by the ubiquitous use of records as data structures. While theoretically everything could be done with tuples, one will often prefer using a record, gaining abstraction over a representation using explicitly ordered formats.

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

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

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

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

The advantage of this labeling system is twofold: it is more expressive and it allows doing partial application selectively on any label.

[1] We use a notation close to CAML [10]. "`let`" denotes a definition, "`::`" the list constructor. Since "=>" is left unused (abstraction uses "->"), we use it for labeling.

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

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

There is no special reason for them to respect this particular order. In fact, the opposite order of arguments would appear more natural, since currying with a given list is more likely. Here, a quick glance at the type eliminates any ambiguity. However, this is not always sufficient. Even if such was the case, the following types would certainly be more perspicuous:

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

With this, one can define such a function as:

```
#let digit = mem in=>[0;1;2;3;4;5;6;7;8;9];;
 digit : {1=>int} -> bool
```

This clearly improves legibility.

Still, one may shrug this argument off since with two arguments, there are only two possibilities of order. With more arguments, however, this quickly becomes irksome. Clearly, remembering arguments order for functions of more than three arguments—and those are not so uncommon—is out of the question.

Let us give some more examples. Consider, for instance, `it_list` and `list_it` (fold left and right), with types:

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

An explicit labeling such as:

```
it_list : {1=>'a list,
           function=>{1=>'b,2=>'a} -> 'b,
           zero=>'b} -> 'b
list_it : {1=>'a list,
           function=>{1=>'a,2=>'b} -> 'b,
           zero=>'b} -> 'b
```

would be more expressive, making the types easier to understand.

We have deliberately restricted our examples to generic functions, for which currying is useful. If we consider functions interfacing a window manager, for example, the number of arguments per function is such that the use of labels is a necessity. In that case, however, one could do with records, since currying is not so important. Nevertheless, 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 an ML dialect preferring currying.

2.2 Relative positions versus combinators

If the main benefit from using symbolic labels is expressiveness, that of relative positions is in conciseness—and efficiency.

Consider, for example:

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

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

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

Of course, the same effect can be obtained using the C combinator defined as:

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

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

But, besides legibility, the hidden loss is efficiency: a combinator is an explicit closure to build and reduce, whereas label commutation enables direct access into the argument stack with offsets. Moreover, for more than two arguments, currying on the kth argument would necessitate $k-1$ such swaps, or use a special combinator for each position—just as expensive.

In addition to this obviously practical benefit, relative position labels provide a coherent bridge connecting classical currying and record currying.

2.3 A generic commutation capability

With respect to types, we can see these extensions as the integration into λ-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:

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

This isomorphism becomes clearer when using indexed products, as in category theory, with explicit projections π_1 and π_2:

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

and thus:

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

Therefore, we obtain a type system in which these isomorphisms, which are part of those described in [5], are directly included.

If we want to keep a confluent calculus, however, it is necessary to sacrifice either generality (two identical keywords may not commute) or referential stability of positions (new projections after commutation). For this reason positions are necessary to allow commuting in any case. They ensure that association between an abstraction and an application is invariant even if their respective positions change. This is important operationally as they allow direct access to distant arguments (*i.e.*, deep in the stack). While symbolic labels are a useful extension of currying, numerical ones are similar to de Bruijn indices [9].

3 λ-Calculus with multiple channels

To meet the behavior that we illustrated with keywords, we define an extension of the λ-calculus, the *symbolic selective λ-calculus*, with symbolic labels.

Selective λ-terms consist of variables, taken from a set $\mathcal{V}$, and two labeled constructions: abstraction and application. We shall assume a non-empty, totally ordered, set of symbols $\mathcal{S}$, to use as labels. We will denote variables by x, y, labels by p, q, and λ-expressions by capital letters. The syntax of selective λ-terms is then given as:

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

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.

To make this compatible with the classical λ-calculus, we shall distinguish a special label, written ι, to use as default.[2] That is, any unlabeled abstraction or application is interpreted as being labeled by ι. In other words, classical λ-calculus is the special case when $\mathcal{S} = \{\iota\}$.

The reduction rules for this calculus are given in Figure 1. β-Reduction only happens on abstraction-application pairs with the same label.[3] Otherwise they commute by rule (3). Rules (1) and (2) simply normalize the order of abstractions and applications.

For convenience, we will sometimes use a variant syntax using record notation. A *record* is an expression of the form $(p_1 \Rightarrow M_1, \ldots, p_n \Rightarrow M_n)$ the p_is are labels and the M_is are terms. We shall use these expressions with the following syntactic equivalence:

$$\begin{array}{lcl} \lambda(p_1 \Rightarrow x_1, \ldots, p_n \Rightarrow x_n).M & \equiv & \lambda_{p_1} x_1. \cdots . \lambda_{p_n} x_n.M \\ M(p_1 \Rightarrow M_1, \ldots, p_n \Rightarrow M_n) & \equiv & (\ldots (M \mathbin{\widehat{}}_{p_1} M_1) \ldots \mathbin{\widehat{}}_{p_n} M_n). \end{array}$$

[2] It will be convenient, though not necessary, to assume that ι is the least element of $\mathcal{S}$.

[3] The notation $[N/x]M$ denotes the term obtained from M after substituting all the free occurrences of variable x with the term N.

Reduction:

$$(\beta) \quad (\lambda_p x.M) \mathbin{\widehat{}}_p N \to [N/x]M$$

Reordering:

$$\begin{array}{lll} (1) & \lambda_p x.\lambda_q y.M \to \lambda_q y.\lambda_p x.M & p > q \\ (2) & M \mathbin{\widehat{}}_p N \mathbin{\widehat{}}_q P \to M \mathbin{\widehat{}}_q P \mathbin{\widehat{}}_p N & p > q \\ (3) & (\lambda_p x.M) \mathbin{\widehat{}}_q N \to \lambda_p x.(M \mathbin{\widehat{}}_q N) & p \neq q,\ x \notin FV(N) \end{array}$$

Figure 1. Reduction rules for symbolic selective λ-calculus

Example: We suppose that $p < q < r < s$:

$$\begin{array}{ll} & \big(\lambda(p \Rightarrow x, q \Rightarrow y, r \Rightarrow z).M\big)(r \Rightarrow N, s \Rightarrow P, p \Rightarrow Q) \\ \equiv & (\lambda_p x.\lambda_q y.\lambda_r z.M) \mathbin{\widehat{}}_r N \mathbin{\widehat{}}_s P \mathbin{\widehat{}}_p Q \\ \to_3 & (\lambda_p x.((\lambda_q y.\lambda_r y.M) \mathbin{\widehat{}}_r N)) \mathbin{\widehat{}}_s P \mathbin{\widehat{}}_p Q \\ \to_2 & (\lambda_p x.((\lambda_q y.\lambda_r y.M) \mathbin{\widehat{}}_r N)) \mathbin{\widehat{}}_p Q \mathbin{\widehat{}}_s P \\ \to_\beta & (\lambda_q y.\lambda_r z.[Q/x]M) \mathbin{\widehat{}}_r N \mathbin{\widehat{}}_s P \\ \to_3 & (\lambda_q y.((\lambda_r z.[Q/x]M) \mathbin{\widehat{}}_r N)) \mathbin{\widehat{}}_s P \\ \to_\beta & (\lambda_q y.([Q/x][N/z]M)) \mathbin{\widehat{}}_s P \\ \to_3 & \lambda_q y.(([Q/x][N/z]M) \mathbin{\widehat{}}_s P) \end{array}$$

We call *symbolic selective λ-calculus* the free combination of these rules and α-conversion.

Theorem 1 *The symbolic selective λ-calculus is confluent.*

Proof: Consequence of the proof for selective λ-calculus, in [2] ∎

4 λ-Calculus with relative positions

This calculus is very similar to the previous one. Its syntax is identical; the only difference is that the labels are positive natural numbers:

$$M ::= x \mid \lambda_n.M \mid M \mathbin{\widehat{}}_n M' \quad \text{where } n \in \mathcal{N} \subseteq \mathbb{N} - \{0\}.$$

Again, for compatibility with the classical λ-calculus, we shall use position 1 as default. That is, any unlabeled abstraction or application is interpreted as being labeled by 1. In other words, classical λ-calculus is the special case when $\mathcal{N} = \{1\}$.

The reduction rules are also similar, but with a twist. They are are given in Figure 2. The main idea here is to preserve coherence between argument position numbers and the property used for currying that all functions are unary. Hence, it is necessary to adjust a position number relatively to the form on its left.

Similarly to what we do for symbolic labels, we will also use a number-labeled record-syntax variant of the raw syntax for convenience. However, unlike the freely commuting

Reduction:

$(\beta)\quad (\lambda_n x.M)\,\widehat{}_n N \to [N/x]M$

Reordering:

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

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

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

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

Figure 2. Reduction rules for numerical selective λ-calculus

symbolic labels, the numbers used as labels in record notation do not correspond directly to the relative position labels of the raw syntax. Namely, translating from the record syntax to raw syntax must readjust an argument's position index by subtracting an offset equal to the number of arguments of lesser position indices on its left. More precisely, let $(n_1 \Rightarrow M_1, \ldots, n_k \Rightarrow M_k)$ be a record expression where $n_i \in \mathcal{N}$ for $i = 1, \ldots, k$. Then, for any $i = 1, \ldots, k$ in this expression, its *relative position offset* is the number $o(i)$ of labels in the set $\{n_1, \ldots, n_{i-1}\}$ that are strictly less than n_i. For example, the relative position offsets of the record expression:

$(4 \Rightarrow M_1, 1 \Rightarrow M_2, 5 \Rightarrow M_3, 2 \Rightarrow M_4, 2 \Rightarrow M_5)$

are: $o(1) = 0, o(2) = 0, o(3) = 2, o(4) = 1, o(5) = 1$.

Hence, the syntactic equivalence is given by:

$$\lambda(n_1 \Rightarrow x_1, \ldots, n_k \Rightarrow x_k).M \;\equiv\; \lambda_{n_1} x_1 . \ldots . \lambda_{n_k - o(k)} x_k . M$$

$$M(n_1 \Rightarrow M_1, \ldots, n_k \Rightarrow M_k) \;\equiv\; (\ldots (M\,\widehat{}_{n_1} M_1) \ldots \,\widehat{}_{n_k - o(k)}\, M_k).$$

Example:

$$\begin{aligned}
& (\lambda(2 \Rightarrow x, 1 \Rightarrow y, 4 \Rightarrow z).M)(4 \Rightarrow N, 6 \Rightarrow P, 2 \Rightarrow Q) \\
\equiv\quad & (\lambda_2 x.\lambda_1 y.\lambda_2 z.M)\,\widehat{}_4 N\,\widehat{}_5 P\,\widehat{}_2 Q \\
\to_4\quad & (\lambda_1 y.\lambda_1 x.\lambda_2 z.M)\,\widehat{}_4 N\,\widehat{}_5 P\,\widehat{}_2 Q \\
\to_7\quad & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\,\widehat{}_3 N))\,\widehat{}_5 P\,\widehat{}_2 n_3 \\
\to_5\quad & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\,\widehat{}_3 N))\,\widehat{}_2 Q\,\widehat{}_4 P \\
\to_7\quad & (\lambda_1 y.\lambda_1 x.((\lambda_2 z.M)\,\widehat{}_2 N))\,\widehat{}_2 Q\,\widehat{}_4 P \\
\to_\beta\quad & (\lambda_1 y.\lambda_1 x.[N/z]M)\,\widehat{}_2 Q\,\widehat{}_4 P \\
\to_7\quad & (\lambda_1 y.((\lambda_1 x.[N/z]M)\,\widehat{}_1 Q))\,\widehat{}_4 P \\
\to_\beta\quad & (\lambda_1 y.[Q/x][N/z]M)\,\widehat{}_4 P \\
\to_7\quad & \lambda_1 y.([Q/x][N/z]M\,\widehat{}_3 P)
\end{aligned}$$

Theorem 2 *The numerical selective λ-calculus is confluent.*

Proof: Consequence of the proof for selective λ-calculus. ∎

5 The selective λ-calculus

The selective λ-calculus combines orthogonally the symbolic and the numerical selective λ-calculi by using $\mathcal{L} = \mathcal{S} \times \mathcal{N}$ as set of labels.[4] Thus its syntax is:

$M ::= x \mid \lambda_\ell.M \mid M\,\widehat{}_\ell M' \quad$ where $\ell = pn \in \mathcal{L} = \mathcal{S} \times \mathcal{N}$.

The reduction system is the combination in Figure 3. Applying these rules simply amounts to applying independently the symbolic and numeric systems. One may see reordering rules as structural equalities, and β-reduction as unique reduction rule. Since the combination is orthogonal, it inherits confluence from both systems.

Reduction:

$(\beta)\quad (\lambda_\ell x.M)\,\widehat{}_\ell N \to [N/x]M$

Symbolic reordering:

$(1)\quad \lambda_{pm} x.\lambda_{qn} y.M \to \lambda_{qn} y.\lambda_{pm} x.M \qquad p > q$

$(2)\quad M\,\widehat{}_{pm} N\,\widehat{}_{qn} P \to M\,\widehat{}_{qn} P\,\widehat{}_{pm} N \qquad p > q$

$(3)\quad (\lambda_{pm} x.M)\,\widehat{}_{qn} N \to \lambda_{pm} x.(M\,\widehat{}_{qn} N) \qquad p \neq q, x \notin FV(N)$

Numeric reordering:

$(4)\quad \lambda_{pm} x.\lambda_{pn} y.M \to \lambda_{pn} y.\lambda_{pm-1} x.M \qquad m > n$

$(5)\quad M\,\widehat{}_{pm} N\,\widehat{}_{pn} P \to M\,\widehat{}_{pn} P\,\widehat{}_{pm-1} N \qquad m > n$

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

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

Figure 3. Reduction rules for selective λ-calculus

Theorem 3 *The selective λ-calculus is confluent.*

Proof: This is a consequence of the proof for the sum system in [2]. We extend easily the use of numerical indices, which can be seen as being limited to a only one keyword in the sum system, to all keywords thanks to channel independence. ∎

To let this system include the symbolic calculus and numerical calculus as sub-calculi, we will identify a symbolic keyword p in the former with the label $(p, 1)$, and a numeric index n in the latter with the label (ι, n). Thus, the classical unlabeled λ-calculus is also syntactically embedded in selective λ-calculus by taking $(\iota, 1)$ as the default label of all abstractions and applications.

[4] In [2] this particular variant was defined as a *product system*, and what we called there 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.

6 Simple types

As in classical λ-calculus, we introduce simple types. There are two benefits. First, we gain a better understanding of the label-selective calculus itself by explicating the type structure that it needs. Second, simply typed selective λ-calculus gains the same nice expected properties; *e.g.*, strong normalization of well-typed terms.

6.1 Syntax and types

The original syntax of terms is extended to:

$$M ::= x \mid \lambda_\ell x{:}t.M \mid M \,\widehat{\ell}\, M'.$$

which requires abstracted variables to be explicitly typed.

We define the syntax of label-selective simple types with the following grammar:

$$\begin{array}{lcll} \ell & ::= & pn & \text{(labels)} \\ u & ::= & u_1 \mid u_2 \mid \ldots & \text{(base types)} \\ r & ::= & \{\ell \Rightarrow t, \ldots\} & \text{(record types)} \\ t & ::= & u \mid r \to u & \text{(general types)} \end{array}$$

where the expression $\{\ell \Rightarrow t, \ldots\}$ denotes a finite partial function from $\mathcal{L}$ to types, including the empty function $\{\}$. We shall identify a functional type of the form $\{\} \to u$ with the base type u. Note that record types are *not* types of expressions of our term language. They are used exclusively as the left subexpression of function types.

The idea behind this syntax of types is to convey that an application can be done indifferently through any label that is present in the type, on a value of corresponding type.

6.2 Record concatenation

We shall provide a simple-type inference system as expected. In order to do so, we must define a record-type concatenation operation needed for extending the domain type of a functional type. Before we give it formally, it is preferable to build some preliminary intuition. We will illustrate the essential mechanism on an example.

To simplify the discussion, let us first restrict ourselves to numeric labels only. Consider the two record types $r = \{2 \Rightarrow t_1, 4 \Rightarrow t_2\}$ and $s = \{2 \Rightarrow u_1, 3 \Rightarrow u_2\}$. Extending the type r on the right with s must be done such that the relative positions be kept in coherence. Now, r expects t_1 in second position and t_2 in fourth position. In other words, positions 1, 3, 5 and up, are "free" in r in the sense that if more arguments were to be expected by an extension of r, they could use these free slots in sequence. Consider now extending r with s. The first argument's position in s is 2. Hence, in r's context, this argument corresponds to the second "free" slot; *i.e.*, position 3. The following one in s is in position 3, and hence corresponds to the third "free" slot in r; *i.e.*, position 5. Thus, the record type resulting from the concatenation of r and s is $r \cdot s = \{2 \Rightarrow t_1, 3 \Rightarrow u_1, 4 \Rightarrow t_2, 5 \Rightarrow u_2\}$.

The case of multiple channels is not more complicated since the above scheme is to be used on each channel independently. Intuitively, this operation reminds of stream merging. In fact, this is exactly what is happening as the indices on a given channel in a record indicate the expected positions, but only relative to this specific record. Extending the record with more indices on this channel necessitates adjusting the new indices by taking their positions with respect to the sequence of indices unused by the initial record. We now proceed to defining formally this record-type concatenation operation.

Let $r = \{\ell_1 \Rightarrow t_1, \ldots, \ell_n \Rightarrow t_n\}$ be a record type. We shall denote by $\mathcal{D}_r = \{\ell_1, \ldots, \ell_n\}$ the set of labels defined in r. Recall that our record labels are not simple symbols, but pairs of the form pn, a symbol and a position index.

Definition 1 (occupied position) *The n^{th} position on p in a record type r is said to be* occupied *if r is such that $pn \in \mathcal{D}_r$.*

Given a record type r, we denote by $o_r(pn)$ the *offset* of n on p in r to be the number of occupied positions on symbol p in r with index less than or equal to n. That is, $o_r(pn) = |(\{p\} \times [1, n]) \cap \mathcal{D}_r|$.

For example, consider the two following record types:

$$r = \{p2 \Rightarrow t_1, p4 \Rightarrow t_2, q1 \Rightarrow t_3, q2 \Rightarrow t_4, q5 \Rightarrow t_5\},$$
$$s = \{p2 \Rightarrow u_1, p3 \Rightarrow u_2, q2 \Rightarrow u_3, q3 \Rightarrow u_4\}.$$

The offsets of the labels of s in r are, respectively, $o_r(p2) = 1$, $o_r(p3) = 1$, $o_r(q2) = 2$, and $o_r(q3) = 2$.

Given a record type r and a given symbol p, we need to identify the least index of p in r that is not an occupied position. More precisely, it is useful to know the nth such free position for symbol p in r.

Definition 2 (free position) *The n^{th}* free position *for p in r is given by $\phi_{r,p}(n) = \min\{i \in \mathcal{N} \mid i - o_r(pi) = n\}$.*

For example, given the two previous example's record types, the free positions in r available for the indices in s are, respectively, $\phi_{r,p}(2) = 3$, $\phi_{r,p}(3) = 5$, $\phi_{r,q}(2) = 4$, and $\phi_{r,q}(3) = 6$.

For fixed r, this function is extended to work also on a record type by distributing it on each label. Namely, for any $s = \{p_i n_i \Rightarrow t_i\}_{i=1}^k$, $\phi_r(s) = \{p_i \phi_{r,p_i}(n_i) \Rightarrow t_i\}_{i=1}^k$.

For example, for the types r and s used above, we have: $\phi_r(s) = \{p3 \Rightarrow u_1, p5 \Rightarrow u_2, q4 \Rightarrow u_3, q6 \Rightarrow u_4\}$. It is not coincidental that the label domain of the $\phi_r(s)$ is disjoint from that of r. It is easy to show that this is true in general.

Definition 3 (concatenation) *Record-type* concatenation *is defined as $r \cdot s = r \uplus \phi_r(s)$, where $\uplus$ denotes union of functions with disjoint domains.*

Going back to the two record types r and s used in the examples above, we have $r \cdot s = \{p2 \Rightarrow t_1, p3 \Rightarrow u_1, p4 \Rightarrow t_2, p5 \Rightarrow u_2, q1 \Rightarrow t_3, q2 \Rightarrow t_4, q4 \Rightarrow u_3, q5 \Rightarrow t_5, q6 \Rightarrow u_4\}$.

Proposition 4 *Label-selective record-types form a monoid;* i.e.*, concatenation is associative with neutral element* $\{\}$.

Proof: This is because $\phi_{r\cdot s} = \phi_r \circ \phi_s$ (see appendix). ∎

6.3 Record matching

It is essential for a syntax-directed inference system, like the typing system that we are about to give, to be able to solve syntactic equations of the form $r \cdot x = s$. More specifically, to extract a subexpression r out of a record type expression it is convenient to write the latter as $r \uplus s$ (*i.e.*, splitting it), and let that be the result of an expression $r \cdot x$, solving for x.

Remarkably, there is an inverse to record-type concatenation that allows solving such an equation and thus may be used to identify a given record type as the result of the concatenation of two other record types. We call this operation *record-type matching*.[5] It will be used with great benefit in typing rules as well as for polymorphic type unification and type inference as shown in the next section.

Let r and s be two record types with disjoint label domains (*i.e.*, such as could be obtained by partitioning one into two). Let pi be a label in s. For p, the position i can be seen as the result of having concatenated r with the same type originally at position $i - o_r(pi)$. In fact, for all the label indices i of p in s, this defines an inverse function for $\phi_{r,p}$ as $\phi^{-1}_{r,p}(i) = i - o_r(pi)$. That is, $\phi^{-1}_{r,p}(i)$ computes the index corresponding to i on channel p skipping the occupied positions on p in r that are less than or equal to i.

As before, for fixed r, ϕ^{-1} is extended to record types. Namely, for any $s = \{p_i n_i \Rightarrow t_i\}_{i=1}^k$ such that $p_i n_i \notin \mathcal{D}_r$ for all $i = 1, \ldots, k$, $\phi_r^{-1}(s) = \{p_i \phi^{-1}_{r,p_i}(n_i) \Rightarrow t_i\}_{i=1}^k$.

Definition 4 (matching) *Record-type* matching *is defined as* $r \uplus s = r \cdot \phi_r^{-1}(s)$*, where* $\uplus$ *denotes union of functions with disjoint domains.*

Let $r = \{p1 \Rightarrow t_1, q2 \Rightarrow t_2\}$ and $s = \{p2 \Rightarrow u_1, q3 \Rightarrow u_2\}$. The unique solution to the matching equation $r \uplus s = r \cdot x$ is $x = \phi_r^{-1}(s) = \{p1 \Rightarrow u_1, q2 \Rightarrow u_2\}$.

6.4 Typing rules

We now have all we need to define well-typedness. We will denote by Γ a *typing environment*; *i.e.*, a mapping from term variables to types. The notation $\Gamma[x \mapsto \tau]$ denotes the typing environment that coincides with Γ everywhere, except on x for which it gives the type τ.

Definition 5 *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 derivable in the type inference system of Figure 4.*

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

[5] Although *division* should be more appropriate.

$$\Gamma[x \mapsto \tau] \vdash x : \tau \qquad \text{(I)}$$

$$\frac{\Gamma[x \mapsto \theta] \vdash M : r \to \tau}{\Gamma \vdash \lambda_\ell x{:}\theta.M : \{l \Rightarrow \theta\} \cdot r \to \tau} \qquad \text{(II)}$$

$$\frac{\Gamma \vdash M : \{\ell \Rightarrow \theta\} \cdot r \to \tau \qquad \Gamma \vdash N : \theta}{\Gamma \vdash M \mathbin{\widehat{\ell}} N : r \to \tau} \qquad \text{(III)}$$

Figure 4. Typing of simply-typed label-selective calculus

Proposition 5 (subject reduction) *Reduction preserves the types;* i.e.*, if* $\Gamma \vdash M : \tau$ *and* $M \to N$ *then* $\Gamma \vdash N : \tau$.

Theorem 6 (strong normalization) *The simply-typed label-selective* λ*-calculus is strongly normalizing.*

7 Polymorphic selective λ-calculus

While there exist typing systems that are more powerful than ML's (*e.g.*, second-order polymorphic λ-calculus), the style of polymorphism used in ML is much simpler. This is essentially due to restricting type quantification to appear only at the outset of type expressions, which facilitates type instantiation to be done implicitly following applications. The main advantage of this type system is that, for λ-calculus, any term has a principal (*i.e.*, most general) type that can be reconstructed from the shape of the term alone. This obviates explicit type declarations: a simple type unification algorithm synthesizes missing types.

We show here that this form of polymorphism is valid also for label-selective λ-calculus. This means that, from a typing point of view, the addition of labels is coherent with polymorphically typed λ-calculus.

7.1 Syntax and types

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

$$M ::= x \mid \lambda_\ell x.M \mid M \mathbin{\widehat{\ell}} M' \mid \textbf{let } x = M \textbf{ in } M'$$

and the reduction rule corresponding to the new construct is:

$$\textbf{let } x = M \textbf{ in } N \to [x/M]N.$$

As in Damas and Milner's definition [8], types are partitioned into monotypes, ranged over by t, and polytypes, ranged over by σ. Thus, the language of types is given by:

$$\begin{array}{lcll} w & ::= & u \mid v & \text{(return types)} \\ r & ::= & \{\ell \Rightarrow t, \ldots\} & \text{(record types)} \\ t & ::= & w \mid r \to w & \text{(monotypes)} \\ \sigma & ::= & t \mid \forall v.\sigma & \text{(polytypes)} \end{array}$$

where return types u stand for base types and v for type variables. Here again, record types are *not* types of expressions of the term language.

7.2 Type substitution

The distinction we introduce here between return types and monotypes is specific to selective λ-calculus. Indeed, as we shall see, the main difficulty in our system, when compared to λ-calculus with ML-style polymorphic types, is that function types are always kept flat. Observe, indeed, that function types are not return types. For example, $\{\ell \Rightarrow \alpha\} \to (\{\ell' \Rightarrow \beta\} \to \gamma)$ is *not* a valid type expression in our type language. It is possible, however, to obtain such an expression as the result of substituting a valid type for a type variable in another valid type. For example, doing a direct substitution with $\{1 \Rightarrow \gamma\} \to \delta$ for β in type $\{1 \Rightarrow \beta\} \to \beta$ would result in $\{1 \Rightarrow (\{1 \Rightarrow \gamma\} \to \delta)\} \to (\{1 \Rightarrow \gamma\} \to \delta)$. 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.

The solution is to define type substitution with a built-in flattening of the domain type. We will denote this operation as $[\tau'\backslash\alpha]\tau$ (*i.e.*, substitute type τ' for type variable α in τ) and it is performed as expressed by the following simple rule:

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

With this rule, our example above results in the valid type $\{1 \Rightarrow (\{1 \Rightarrow \gamma\} \to \delta), 2 \Rightarrow \gamma\} \to \delta$

This illustrates how our domain of types is radically different from the conventional Herbrand universe with the arrow and base type constructors, whose well-known term unification is exploited for ML-type inference. We shall thus need to provide our explicit unification algorithm. It is a nice property of our system that unique most general unifiers exist for our type terms. As we shall see, this is essentially due to the well-foundedness of normalization to flattened types which does not change the size of types.

7.3 Typing rules

The typing rules are given in Figure 5. It is interesting to remark that Rules (IV)–(VI) are in no way specific to selective λ-calculus. Since type quantifiers are external, they are independent of the structure of monotypes. Thus, these rules are exactly the same used in classical λ-calculus. Their roles are *generalization* (IV), *instantiation* (V), and **let**-*introduction* (VI). The only, but important, difference between these rules and the classical ones is hidden in the use of our flattening type substitution $[\tau\backslash\alpha]\sigma$ in Rule (V).

Again, all the desirable properties hold for the polymorphically typed selective λ-calculus, as expressed by the two following propositions.

Proposition 7 (subject reduction) *If $\Gamma \vdash M : \tau$ in polymorphically typed selective λ-calculus, and $M \to N$, then $\Gamma \vdash N : \tau$.*

$$\Gamma[x \mapsto \sigma] \vdash x : \sigma \qquad \text{(I)}$$

$$\frac{\Gamma[x \mapsto \theta] \vdash M : r \to \tau}{\Gamma \vdash \lambda_\ell x.M : \{l \Rightarrow \theta\} \cdot r \to \tau} \qquad \text{(II)}$$

$$\frac{\Gamma \vdash M : \{l \Rightarrow \theta\} \cdot r \to \tau \qquad \Gamma \vdash N : \theta}{\Gamma \vdash M \,\widehat{\ell}\, N : \tau} \qquad \text{(III)}$$

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

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : [\tau\backslash\alpha]\sigma} \qquad \text{(V)}$$

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

Figure 5. Typing rules for polymorphic selective λ-calculus

Theorem 8 (strong normalization) *Polymorphic selective λ-calculus is strongly normalizing.*

7.4 Type unification

The key for type synthesis is unification. We give here a unification algorithm for the label-selective monotypes defined above. It can be expressed as a simple E-unification problem [11], where the equational theory is that deciding equality of record types. Then, our type substitution operation using record-type concatenation constitutes a complete set of reduction for this theory. We next give this unification procedure as a complete set of equivalence-preserving transformations on a set of type equations.

A set of type equations φ is said to be in *solved form* if every equation in it is of the form $\alpha = \tau$ such that the type variable α occurs only once in φ; *viz.*, as this equation's lefthand-side. As usual, such a solved-form defines a variable substitution that can be applied to type expressions.

Figure 6 contains the complete set of transformations for the unification of label-selective monotypes. We use the notation α for type variables, ω for return types, and τ or θ for any type expression. (Recall that $\{\} \to \omega$ is identified with ω.)

These rules work on a set (a conjunction) of type equations, transforming it into another such set. Upon termination, having started from a set φ of equations, the resulting equation set is either $\bot$, the inconsistent equation indicating

(Base type)

$$\frac{\varphi,\ u = v}{\bot} \qquad u \neq v,\ u, v \text{ base types}$$

(Function type)

$$\frac{\varphi,\ u = r \to \omega}{\bot} \qquad u \text{ base type},\ r \neq \{\}$$

(Variable recurrence)

$$\frac{\varphi,\ \alpha = \tau}{\bot} \qquad \tau \neq \alpha,\ \alpha \in \mathit{Var}(\tau)$$

(Variable elimination)

$$\frac{\varphi,\ \alpha = \tau}{[\tau\backslash\alpha]\varphi, \alpha = \tau} \qquad \alpha \in \mathit{Var}(\varphi) - \mathit{Var}(\tau),\ \text{if } \tau \text{ variable, then } \tau \in \mathit{Var}(\varphi)$$

(Variable orientation)

$$\frac{\varphi,\ \tau = \alpha}{\varphi,\ \alpha = \tau} \qquad \alpha \text{ variable},\ \tau \text{ not variable}$$

(Redundancy)

$$\frac{\varphi,\ \theta = \theta}{\varphi}$$

(Decomposition)

$$\frac{\varphi,\ \{\ell \Rightarrow \theta\} \cdot r \to \omega = \{\ell \Rightarrow \theta'\} \cdot r' \to \omega'}{\varphi,\ \theta = \theta',\ r \to \omega = r' \to \omega'}$$

(Label completion)

$$\frac{\varphi,\ \{\ell \Rightarrow \theta\} \cdot r \to \omega = r' \to \omega'}{\varphi,\ \{\ell \Rightarrow \theta\} \cdot r \to \omega = \{\ell \Rightarrow \theta\} \uplus r' \to \alpha,\ \omega' = \phi_{r'}^{-1}\{\ell \Rightarrow \theta\} \to \alpha} \qquad r' \neq \{\},\ \ell \notin \mathcal{D}_{r'},\ \alpha \text{ fresh}$$

Figure 6. Equation-rewriting rules for type unification

that no solution exists, or $\mathbf{sol}(\varphi)$, a set of equations in solved form equivalent to φ.

In either case, this process can be seen as returning a substitution. In the first case, it is the failing substitution $\bot$ such that $\bot\ (\tau) = \bot$ for all types τ, where $\bot$ denotes the inconsistent type. In the second case, the solved form $\mathbf{sol}(\varphi)$ is the most general unifier (MGU) of φ (up to variable renaming). The rules are written as rewrite rules using a comma as an associative and commutative set constructor, and the equal sign as a symmetric equation constructor. That is, in these rules the particular order of equations in the set as well as the orientation of an equation are irrelevant. As established by the following theorem, they are solution-preserving and there is a deterministic strategy that makes them always terminate.

Theorem 9 (label-selective type unification) *There is an algorithm that computes the most general unifier of a set of equations on monotypes or reports failure if there is none.*

7.5 Type inference

It is now easy to derive a type inference algorithm by combining type unification with the typing rules of Figure 5. It is sufficient to following the syntactic structure of a given term, accumulating new equations in a set, as shown in Figure 7. The function *Tp* takes a typing environment Γ (a function from term variables to types) and a selective λ-term M, and returns a pair $\langle \varphi, \tau \rangle$ where φ is a set of type equations in solved form (*i.e.*, a type substitution), and τ is the principal type of M. The function **strip** applies to a type expression $\forall\alpha_1.\cdots\forall\alpha_n.\tau$, where $n \geq 0$, and returns the expression obtained from τ where all the α_is, if any, are replaced with fresh names.[6] The expression $FV(\tau)$ is the set of free variables in τ, and by extension $FV(\Gamma) = \bigcup_x FV(\Gamma(x))$. The expression $\mathbf{sol}(\varphi)$, where φ is a set of type equations, is the solved form of φ (*i.e.*, the MGU of φ). It is the result of applying the transformation rules of Figure 6 to φ until none applies. The expression $\varphi(\tau)$ is the result of applying the substitution φ to the type expression τ. By extension, $\varphi(\Gamma)$ is the function defined by $\varphi(\Gamma)(x) = \varphi(\Gamma(x))$.

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, this algorithm is complete and correct. This is because only necessary equations are added, except for generalization and instantiation, which are handled in the most general way in the variable and **let** cases.

[6] If $n = 0$ there is no quantifier, and thus **strip** returns the given type expression as is.

$$
\begin{array}{rcll}
Tp(\Gamma, x) & = & \langle \varnothing, \mathbf{strip}(\Gamma(x)) \rangle & \\
Tp(\Gamma, \lambda_\ell x.M) & = & \langle \varphi, [\tau \backslash \beta](\{\ell \Rightarrow \alpha\} \to \beta) \rangle & \text{where } \left\{ \begin{array}{l} \langle \varphi, \tau \rangle = Tp(\Gamma[x \mapsto \alpha], M) \\ \alpha, \beta \text{ fresh} \end{array} \right. \\
Tp(\Gamma, M \,\widehat{\ell}\, M') & = & \langle \mathbf{sol}(\varphi \cup \varphi' \cup \{\tau = \{\ell \Rightarrow \tau'\} \to \alpha\}), \alpha \rangle & \text{where } \left\{ \begin{array}{l} \langle \varphi, \tau \rangle = Tp(\Gamma, M) \\ \langle \varphi', \tau' \rangle = Tp(\Gamma, M') \\ \alpha \text{ fresh} \end{array} \right. \\
Tp(\Gamma, \mathbf{let}\ x = M\ \mathbf{in}\ M') & = & \langle \mathbf{sol}(\varphi \cup \varphi'), \tau' \rangle & \text{where } \left\{ \begin{array}{l} \langle \varphi, \tau \rangle = Tp(\Gamma, M) \\ \langle \varphi', \tau' \rangle = Tp(\Gamma', M') \\ \Gamma' = \Gamma[x \mapsto \forall \big(FV(\tau) - FV(\varphi(\Gamma))\big).\tau] \end{array} \right.
\end{array}
$$

Figure 7. Type inference algorithm

8 Conclusion and further work

We have proposed two typing systems for label-selective λ-calculus: simple types and ML-style polymorphic types. The latter are smoothly accommodated thanks to the existence of a simple but flexible record-type concatenation operation that facilitates building label-selective currying right into type substitution and unification. Integrated into a polymorphic functional programming language with currying, this provides a powerful tool, extending currying facilities and helping to memorize multi-argument functions.

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

$$\{\ell_1 \Rightarrow a_1, \ldots, \ell_n \Rightarrow a_n\} \longrightarrow \lambda_{sel} s.(s\, \widehat{\ell_1}\, a_1 \ldots \widehat{\ell_n}\, a_n)$$

where *sel* is a distinguished fixed channel and s is a function selecting a label and discarding the others individually (we have no way to discard them at once), like $\lambda_{\ell_1} x_1 \ldots . \lambda_{\ell_n} x_n . x_k$. We can even have functions using more than one label. This is in fact the basic idea for a *transformation calculus*. However, there are some essential differences between a classical definition of records and this encoding as it accommodates numerical indices. We suspect that type inference of such a calculus with useful operations might turn out to be rather complex.

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 [16]. The conjunction of those two views seems an interesting prospective.

The last, but more immediate, concern 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: one that is faithful to label names, and one where they can be replaced by numeric stack offsets. This might be the basis for an efficient compilation method, which should be built on a completely curried vision. That is, there should be no overhead caused by currying. The efficient compilation method given by Ohori for a record calculus [18] gives us some evidence that this is possible.

Acknowledgements

The authors are indebted to Atsushi Ohori for invaluable discussions.

Appendix: Proofs of theorems

Proposition 4 *Record-type concatenation is associative.*

Proof: Let us show that $\phi_r \circ \phi_s = \phi_{r \uplus \phi_r(s)}$. We proceed with inverses:

$$
\begin{array}{rl}
\phi^{-1}_{r \uplus \phi_r(s), p}(i) & = i - o_{r \uplus \phi_r(s)}(pi) \\
& = i - o_r(pi) - o_s(\phi^{-1}_{r,p}(i)) \\
& = \phi^{-1}_{s,p}(\phi^{-1}_{r,p}(i)).
\end{array}
$$

We then have:

$$\begin{aligned} r \cdot (s \cdot t) &= r \uplus \phi_r(s \uplus \phi_s(t)) \\ &= r \uplus \phi_r(s) \uplus \phi_r(\phi_s(t)) \\ &= r \uplus \phi_r(s) \uplus \phi_{r \uplus \phi_r(s)}(t) \\ &= (r \cdot s) \cdot t. \end{aligned}$$

∎

Proposition 5 (subject reduction) *If $\Gamma \vdash M : \tau$ and $M \to N$ then $\Gamma \vdash N : \tau$.*

Proof: 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 $(\lambda_\ell x{:}\theta.P)\,\widehat{\iota}\,Q$. Then, the basis of the proof tree is:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash P : r \to \tau}{\Gamma \vdash \lambda_\ell x{:}\theta.P : \{l \Rightarrow \theta\} \cdot r \to \tau} \quad \Gamma \vdash Q : \theta}{\Gamma \vdash M : r \to \tau}$$

After reduction the result is $N = [x/Q]P$. We obtain a derivation tree for $\Gamma \vdash [x/Q]P$ from those of $\Gamma[x \mapsto \theta] \vdash P : r \to \tau$ and $\Gamma \vdash Q : \theta$ as follows: (1) doing all α-conversions necessary to the substitution of x by Q; (2) suppressing x in the environments (except where it is redefined by an abstraction); (3) where x appears without being defined in the environment, replacing $\Gamma' \vdash x : \theta$ by the derivation tree of $\Gamma' \vdash Q : \theta$. This poses no problem since $\forall y \in FV(Q)\ \Gamma(y) = \Gamma'(y)$.

If the reduction is a reordering, we have seven cases. We will only work out in detail cases (3), (6) and (7).

Case (3): If the reduction is (3), then the derivation tree must have the following form:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash M : \{qn \Rightarrow \theta'\} \cdot r \to \tau}{\Gamma \vdash \lambda_{pm} x{:}\theta.M : \{pm \Rightarrow \theta, qn \Rightarrow \theta'\} \cdot r \to \tau} \quad \Gamma \vdash N : \theta'}{\Gamma \vdash (\lambda_{pm} x{:}\theta.M)\,\widehat{qn}\,N : \{pm \Rightarrow \theta\} \cdot r \to \tau}$$

Since $x \notin FV(N)$, we can obtain the following derivation tree after reordering:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash M : \{qn \Rightarrow \theta'\} \cdot r \to \tau \qquad \Gamma[x \mapsto \theta] \vdash N : \theta'}{\Gamma[x \mapsto \theta] \vdash M\,\widehat{qn}\,N : r \to \tau}}{\Gamma \vdash \lambda_{pm} x{:}\theta.(M\,\widehat{qn}\,N) : \{pm \Rightarrow \theta\} \cdot r \to \tau}$$

Case (6): If the reduction is (6), then $n < m$, and the derivation tree must have the following form:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash M : \{pn \Rightarrow \theta'\} \cdot r \to \tau}{\Gamma \vdash \lambda_{pm} x{:}\theta.M : \{pn \Rightarrow \theta', pm \Rightarrow \theta\} \cdot r \to \tau} \quad \Gamma \vdash N : \theta'}{\Gamma \vdash (\lambda_{pm} x{:}\theta.M)\,\widehat{pn}\,N : \{p(m-1) \Rightarrow \theta\} \cdot r \to \tau}$$

Since $x \notin FV(N)$, we can obtain the following derivation tree after reordering:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash M : \{pn \Rightarrow \theta'\} \cdot r \to \tau \qquad \Gamma[x \mapsto \theta] \vdash N : \theta'}{\Gamma[x \mapsto \theta] \vdash M\,\widehat{pn}\,N : r \to \tau}}{\Gamma \vdash \lambda_{p(m-1)} x{:}\theta.(M\,\widehat{pn}\,N) : \{p(m-1) \Rightarrow \theta\} \cdot r \to \tau}$$

Case (7): If the reduction is (7), then $m < n$, and the derivation tree must have the following form:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash M : \{p(n-1) \Rightarrow \theta'\} \cdot r \to \tau}{\Gamma \vdash \lambda_{pm} x{:}\theta.M : \{pm \Rightarrow \theta, pn \Rightarrow \theta'\} \cdot r \to \tau} \quad \Gamma \vdash N : \theta'}{\Gamma \vdash (\lambda_{pm} x{:}\theta.M)\,\widehat{pn}\,N : \{pm \Rightarrow \theta\} \cdot r \to \tau}$$

Since $x \notin FV(N)$, we can obtain the following derivation tree after reordering:

$$\dfrac{\dfrac{\Gamma[x \mapsto \theta] \vdash M : \{p(n-1) \Rightarrow \theta'\} \cdot r \to \tau \quad \Gamma[x \mapsto \theta] \vdash N : \theta'}{\Gamma[x \mapsto \theta] \vdash M_{\widehat{p(n-1)}}\,N : r \to \tau}}{\Gamma \vdash \lambda_{pm} x{:}\theta.(M\,\widehat{pn}\,N) : \{pm \Rightarrow \theta\} \cdot r \to \tau}$$

∎

Theorem 6 *The simply typed selective λ-calculus is strongly normalizing.*

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

First, let us define *zero functions*, and the operation of *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.

Let $\tau = \{\ell_1 \Rightarrow \tau_1, \ldots, \ell_n \Rightarrow \tau_n\} \to u$ be a simple type. The *zero-function* for τ, noted 0^τ, is the function $\lambda(\ell_1 \Rightarrow x_1 : \tau_1^*, \ldots, \ell_n \Rightarrow x_n : \tau_n^*).0$, of type τ^*, where $*$ is defined by induction as $(\{\ell_1 \Rightarrow \tau_1, \ldots\} \to u)^* = \{\ell_1 \Rightarrow \tau_1^*, \ldots\} \to \mathit{int}$ (we replace every base type with *int*).

To *rectify* a function f of type $\tau = \{\ell_1 \Rightarrow \tau_1, \ldots, \ell_n \Rightarrow \tau_m\} \cdot r \to u$ to $r \to u$ one simply applies it to the corresponding zero-functions: $\mathit{rect}(r \to u, f : \tau) = f(\ell_1 \Rightarrow 0^{\tau_1}, \ldots, \ell_n \Rightarrow 0^{\tau_n})$.

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.

For a variable $\Gamma \vdash x : \tau$, the associated function is $\lambda_x x{:}\tau^*.x$.

For an abstraction $\Gamma \vdash \lambda_\ell x : \theta.M : \{l \Rightarrow \theta\} \cdot r \to \tau$, the associated function is $\lambda_\ell x{:}\theta^*.T_{\Gamma[x \mapsto \theta]}(M)$.

For an application $\Gamma \vdash M \,\widehat{\ell}\, N : r \to \tau$, with $\Gamma \vdash N : \theta$, the associated function is:

$$T_\Gamma(M \,\widehat{\ell}\, N) = \lambda_{x_1} x_1 . \cdots . \lambda_{x_n} x_n . \big(\; (T_\Gamma(M) \,\widehat{}_{x_1}\, x_1 \ldots \,\widehat{}_{x_k}\, x_k \,\widehat{\ell}\, N^a) + \mathit{rect}(\mathit{int}, N^a : \theta^*) + 1 \; \big)$$

where:

1. $FV(N) = \{x_1, \ldots, x_n\}$;
2. $FV(M) \cap FV(N) = \{x_1, \ldots, x_k\}, 0 \le k \le n$;
3. $N^a = T_\Gamma(N) \,\widehat{}_{x_1}\, x_1 \ldots \,\widehat{}_{x_n}\, x_n$;
4. for $f : \{\ell_1 \Rightarrow \theta_1, \ldots, \ell_n \Rightarrow \theta_n\} \to \mathit{int}$ and $a : \mathit{int}$,
$$f + a = \lambda(\ell_1 \Rightarrow x_1 : \theta_1, \ldots, \ell_n \Rightarrow x_n : \theta_n) . (f(\ell_1 \Rightarrow x_1, \ldots, \ell_n \Rightarrow x_n) + a).$$

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, whether or not there is an abstraction to reduce. 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 $\Gamma \vdash M : \{\ell_1 \Rightarrow \theta_1, \ldots, \ell_n \Rightarrow \theta_n\} \to \tau$ 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 that is coherent with its typing, M will terminate. Moreover, an upper bound of its longest reduction path is given by $\mathit{rect}(\mathit{int}, T_\Gamma(M) : (r \to \tau)^*)$. ∎

Proposition 7 (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:

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

We first perform all α-conversions necessary to the substitution of x by M. After reduction, we obtain a tree with root $\Gamma \vdash [x/M]N : \tau$ 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 $\Gamma' \vdash M : \sigma$; observing that $\forall y \in FV(M)\ \Gamma(y) = \Gamma'(y)$. ∎

Theorem 8 *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 $K \,\widehat{1}\, ([x/P]N) \,\widehat{1}\, P$, where $K = \lambda_1 x.\lambda_1 y.x$. We need K for the case where x does not appear in N. Since the result is monomorphic everywhere, the argument for the simply typed calculus holds. ∎

Theorem 9 (label-selective type unification) *There is an algorithm that computes the most general unifier of a set of equations on monotypes or reports failure if there is none.*

Proof: We first prove the correctness of the rewriting system of Figure 6. That is, for each rewrite rule, any solution of the denominator is a solution of the numerator, and conversely, any solution of the numerator can be extended into a solution of the denominator, possibly by introducing new variables missing in the numerator.

The rules labeled *Base type*, *Variable recurrence*, *Function type* detect inconsistencies in the equations. That is, respectively, 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 applies, the system has no unifier.

The *Variable elimination* rule substitutes variables (using flattening type substitution), while keeping their referents. Let σ be a solution of the numerator. Then, by construction, $\sigma(\alpha) = \sigma(\tau)$, thus it is also a solution of the denominator; and conversely.

The *Variable orientation* rule simply reorients an equation. It is not really necessary and is provided only to obtain the solved form with all solved variables on the left. Clearly, it leaves unchanged the set of unifiers. So does the *Redundancy* rule which just suppresses tautological equations.

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

Whenever a label appears only on one side of an equation, it is necessary to introduce it in the other side. This is done by the *Label completion* rule using record-type matching. Any unifier of the denominator is also a solution of the numerator, since $\{\ell \Rightarrow \theta\} \uplus r' \to \alpha = r' \cdot \phi_{r'}^{-1}\{\ell \Rightarrow \theta\} \to \alpha$, which is

by unification equal to $r' \to \omega'$. Conversely, if σ is a unifier of the numerator, then it maps ω' to a functional type of the form $\phi_r^{-1}\{\ell \Rightarrow \sigma(\theta)\} \cdot r'' \to \omega''$, which can be extended for the denominator by adding $\sigma(\alpha) = r'' \to \omega''$.

We next prove that there is a terminating strategy. Termination follows for the well-foundedness of a decreasing measure. A variable is *solved* when it appears only once, and as the lefthand-side of an equation. We exhibit a strategy that reduces the lexicographical measure *(number of unsolved variables,sum of sizes)*, where the size of a type is the total number of labels, variable occurrences and base types it contains.

The three failing rules terminate. Redundancy, and Decomposition reduce the sum of sizes. Variable elimination and Variable orientation reduce the number of unsolved variables.

Label completion by itself does not reduce the measure. But if it is always used it in combination with Decomposition on the same equation, eliminating or failing as soon as possible, this always reduces the number of unsolved variables. If ω' is 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.

Last, we must show that our result is in solved form. 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. ∎

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 31–46 (1990).
2. Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus. PRL Research report 31, Digital Equipment Corporation, Paris Research Laboratory (May 1993).
3. Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. In *Proceedings of the 13th International Conference on Foundations of Software Technology and Theoretical Computer Science (Bombay, India)*, LNCS 761. Springer-Verlag (December 1993).
4. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195–234 (July-August 1993).
5. Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical Report LIENS-90-14, LIENS (July 1990).
6. Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164 (1988).
7. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):457–522 (1985).
8. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 207–212 (1982).
9. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392 (1972).
10. Pierre Weis *et al. The CAML Reference Manual, version 2.6.1*. Projet Formel, INRIA-ENS (1990).
11. Jean Gallier and Wayne Snyder. Designing unification procedures using transformations: a survey. In Y. N. Moschovakis, editor, *Logic from Computer Science*, pages 153–215. Springer-Verlag (1989).
12. Jacques Garrigue. Label-selective λ-calculus. Rapport de D.E.A., Université Paris VII (1992).
13. Lalita Jategaonkar and John Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 198–211 (1988).
14. John Lamping. A unified system of parameterization for programming languages. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 316–326 (1988).
15. Henry Ledgard. *ADA : An Introduction, Ada Reference Manual (July 1980)*. Springer-Verlag (1981).
16. 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).
17. Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 43–56 (1993).
18. Atsushi Ohori. A compilation method for ML-style polymorphic records. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–165 (1992).
19. Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 77–87 (1989).
20. R. Stansifer. Type inference with subtypes. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 88–97 (1988).
21. Guy L. Steele. *Common LISP, The Language*. Digital Press (1984).
22. Mitchell Wand. Complete type inference for simple objects. In *Proceedings of IEEE Symposium on Logic in Computer Science* (1988).