変換計算への状態を持ったオブジェクトの導入

Introducing Stateful Objects in a Transformation Calculus

Jacques Garrigue *

Summary. When representing objects in a strongly typed functional language, two problems appear. One is to express inheritance, the other is to represent states. A lot of work has already been done on the first, through extensions of the typing system, but interest in the second is more recent. We limit ourselves here to this last problem of state representation.

To do this we use transformation calculus, an extension of selective λ-calculus, itself an extension of lambda calculus allowing selection of arguments through labels. One can see this last one as a generalization of currying, the standard way of expressing multi-argument function in lambda calculus, to keyword-marked arguments. These two calculi include the lambda calculus as a sub-calculus, and we show that transformation calculus and selective λ-calculus, as untyped calculi, are in some way isomorphic.

However, transformation calculus is meant to express state transformations, by using labels as identifiers. As such it proposes a basis for a functional representation of stateful objects. This is done by separating each object in methods, viewed as transformations, and a labeled state. Each object having its own label, a method modifying its state is just a transformation modifying the value attached to this label.

By such a representation, we can obtain powerful types, where side-effects are made evident, in perfect homogeneity with functional types. In this perspective, our goal here is more to use stateful objects as a representation for non-functional parts of a program, than to introduce object-orientation in functional programming.

* 東京大学 大学院 理学系研究科 情報科学専攻 米澤研究室. garrigue@is.s.u-tokyo.ac.jp

1 Introduction

The problem of introducing the concepts of object oriented programming in a strongly typed calculus has been devoted many researches in the last decade, particularly since the the work of Cardelli [Car88], which introduced the idea of representing inheritance by subtyping in a calculus where objects are viewed as records.

However this mainly concerns inheritance. States, which are useful for many algorithms, and essential in objects, are a rather new field of study. An idea, followed by both Lamping [Lam88] and Odersky, Rabin, Hudak [ORH93], is to formalize the role of references, often used in functional languages to obtain mutable values. Their poor integration in available languages, on the sides of typing and referential transparency, makes the cautious programmer avoid them, but a better definition could justify their use. However these two calculi are still untyped.

Selective λ-calculus, which at first sight may look similar to the two precedent calculi, is based on the very simple idea of labeling abstractions and applications, allowing β-reduction only between abstractions and applications of same label, and letting them commute otherwise. Numerical labels need a more complex handling, but we present here a simplified calculus, allowing only symbolic labels. The essential novelty is that there is no longer any distinction between references and λ-abstraction: the same labeled operation does the two, selecting an argument and abstracting a value. We define this calculus in Section 2.

However this calculus does not contain explicitly the notion of state or even record. A short look shows that it may encode records. This is not a surprise, since it already contains labels. By explicitly expressing records, we extend it into a transformation calculus. The name comes from the ability of this calculus, where composition is a primitive operation, to express state transformations as updates on a record (allowing label repetition). If it inherits from selective λ-calculus the fundamental characteristic of assimilating selection (which amounts to looking up a reference) and abstraction in a unique operation, it distinguishes clearly the scopes of bound variables and states: one is linked with term containment, the other with a segment of the operating flow. It is presented in Section 3.

The idea here of representing states as records is not new. However, since a fixed type of record would reduce the generality of the function using it, they are often replaced by the preceding idea of stores acceded through references, untyped, or only with type attached to their elements. The calculus proposed by Ohori [Oho92], which permits polymorphic record updates, strengthens the record approach.

Close to the above-cited two works, but in a semantical approach, is that by Johnson and Duggan [JD88], based on stores, which, while keeping the distinction between continuation and function, presents important similarities with our proposal. In particular their *partial continuations* can be seen as transformations, and what they call "applicative continuation semantics" is a consequence of the translation theorem we give for the transformation calculus.

After presenting these two calculi we introduce our idea of representing an object as a set of transformations concerning a label. This label can be seen as the identifier of our object, and encapsulation means making this label accessible only from its methods. This functional approach differs from a frequent one in the object oriented field, which sees methods as being invocated by messages sent to the object [AR88, Red88], which is considered as "active" and "atomic" (methods and state are together *in* the objet). In our approach an object is simply a set of transformations implementing methods, plus a label holding its state. In ignoring the notion messages we do like [Car88], but he suggested to include the state in the record representing the methods. The distinction we make here avoids self-references, which made necessary recursively quantified types. These differences of view do not necessarily mean that syntax should be different. Thinking in an object-oriented way, that only means that, as long as we provide the same external features, we may view the internal part of a concept completely differently.

To summarize, we will first present selective λ-calculus and transformation calculus in Sections 2 and 3. Then we show how transformation calculus provides a representation for states in Section 4, and apply it to stateful objects in Section 5.

2 Selective λ-calculus

We present here a restricted version of selective λ-calculus. The original version [Gar92, AKG93] distinguishes between numeric and symbolic labels, trying to give to the first their intuitive meaning. Since this is not directly relevant to the problem we are addressing here, we will use only symbolic labels, from an ordered set $\mathcal{L}$. However, to preserve partly the original intuition, we will include a label "1" in $\mathcal{L}$, which is meant to be the default label. That is, any expression written using only "1" as label is equivalent to its non-labeled version.

The syntax of selective λ-calculus is as follows. M represents terms, p

β-reduction

(β) $(\lambda_p x.M)\,\widehat{p}\,N \rightarrow M[x\backslash N]$

Reordering

(1) $\lambda_p x.\lambda_q y.M \rightarrow \lambda_q y.\lambda_p x.M \qquad p > q$

(2) $M\,\widehat{p}\,N\,\widehat{q}\,L \rightarrow M\,\widehat{q}\,L\,\widehat{p}\,N \qquad p > q$

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

Figure 1. Rewriting rules for selective λ-calculus

labels. "′"s mark different occurences of the same non-terminal.

$$\begin{array}{lll} M ::= x & & \text{variable} \\ \quad | \ \lambda_p x.M & & x \text{ abstracted on } p \text{ in } M \\ \quad | \ M\,\widehat{p}\,M' & & M \text{ applied on } p \text{ to } M' \end{array}$$

The intuitive idea, that abstraction and application of same label form redexes, is implemented by the conditional rewriting rules in figure 1 (FV stands for free variables), structurally extended in the classical way.

This rewriting system has been proved to be confluent [AKG93]. Rules are directed to enforce termination of reordering (rules 1-3), semantically they can be seen as equalities.

Another important result is that selective λ-terms are polymorphically typable, and a type inference algorithm has been found [GAK93]. On typable terms, the preceding rewriting system is strongly normalizing.

We give here two short examples of functions typechecked in a syntactic extension of CAML [W$^+$90] using labels. Pairs label-argument are denoted by "p:v". Omission of the label means the default label "1:v".

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

#let rec it_list f:f start:s = fun [] -> s
#       | [h|t] -> it_list f:f start:(f s h) 1:t ;;
 it_list : (1:'a list f:(1:'b 1:'a -> 'b) start:'b -> 'b)
```

3 Transformation calculus

Transformation calculus comes from a study of the properties of some models of selective λ-calculus. Particularly, to emphasize the specificity of selective

λ-calculus, one is brought to introduce records. When translating this in a syntactical form, we can see that records are already here.

$$\lambda_{sel} x.(x\, \widehat{l_1}\, M_1 \ldots \widehat{l_n}\, M_n)$$

where the label *sel* means that we expect a selector in x, can be used as a record. In order to read a field, we must apply it to a selector. A selector is a function that discards all other fields and gives back the desired one, like

$$(\lambda_{l_1} y \ldots \lambda_{l_k} x \ldots \lambda_{l_n} y.x)$$

to select the field l_k and discard all other l_i.

But to write such a record we have to introduce a variable, and construct a functional term. We can look at the syntax, and see that $\widehat{l_1}\, M_1 \ldots \widehat{l_n}\, M_n$ would be enough to describe our record. This is not a term, so we define a meta-operation, "$\cdot$", syntactic juxtaposition, taking a term and such a record, and creating a new term by applying it successively to all the elements of the record.

$$M \cdot \widehat{l_1}\, M_1 \ldots \widehat{l_n}\, M_n = M\, \widehat{l_1}\, M_1 \ldots \widehat{l_n}\, M_n$$

This does not modify the calculus itself, but provides us with a new notation. Similarly, to emphasize the symmetry between application and abstraction, we will change the notation for abstraction, and assimilate it to application.

$$\lambda_p x.M \longrightarrow M\, \overset{\vee}{p}\, x$$

This notation makes information flow from right to left in expressions, for variables as well as labels. We can even extend syntactic juxtaposition to have abstractions as well as applications on its right side. With such a notation we can give a new formulation to β-reduction.

$$M \cdot \overset{\vee}{p}\, x\, \widehat{p}\, N \cdot P \rightarrow M[x \backslash N] \cdot P$$

We cut here our term in three parts: a left term, the redex, and a right sequence. In the reduction, the redex disappears modifying the left term and leaving the right part unchanged.

Using these new notations, we can give the definition of transformation calculus, which is selective λ-calculus plus a constant "$\{\}$" or *empty stream*. The name "stream" comes from the intuition we are transforming a stream of arguments, but this denotes only records with eventual repetition of labels, and not an infinite structure.

$$\begin{aligned} M ::= {} & x \mid M\, \overset{\vee}{p}\, x \mid M\, \widehat{p}\, M' \\ & \mid \{\} \qquad \text{empty stream} \\ & \mid M \circ M' \; M \text{ composed with } M' \end{aligned}$$

Reduction

$(\beta)\ M \overset{\vee}{_p}\ x \mathbin{\hat{p}} N \to M[x \backslash N]$

$(\circ)\ M \circ (\{\} \cdot P) \to M \cdot P \qquad FV(M) \cap BV(P) = \emptyset$

Reordering

$(1)\ M \overset{\vee}{_q}\ y \overset{\vee}{_p}\ x \to M \overset{\vee}{_p}\ x \overset{\vee}{_q}\ y \qquad p > q$

$(2)\ M \mathbin{\hat{p}} N \mathbin{\hat{q}} L \to M \mathbin{\hat{q}} L \mathbin{\hat{p}} N \qquad p > q$

$(3)\ M \overset{\vee}{_p}\ x \mathbin{\hat{q}} N \to M \mathbin{\hat{q}} N \overset{\vee}{_p}\ x \qquad p \neq q, x \notin FV(M')$

$(4)\ (M \circ N) \mathbin{\hat{p}} L \to M \circ (N \mathbin{\hat{p}} L)$

$(5)\ M \circ (N \overset{\vee}{_p}\ x) \to (M \circ N) \overset{\vee}{_p}\ x \quad x \notin FV(M)$

Figure 2. Rewriting rules for transformation calculus

We have some new rewriting rules to handle composition (cf. figure 2). Composition has some sort of "associativity" with abstraction and application, which is expressed by rules (4) and (5).

We can structurally distinguish transformations, based on the empty stream $(\{\} \cdot P)$, from functions, based on variables $(x \cdot P)$. During a reduction, a function may be converted in a transformation, by substituting its head variable with $\{\}$, but a transformation will always stay a transformation.

In fact (5) is not necessary to get a coherent calculus, but is semantically correct, and provides better-looking normal forms. It makes evident the translation Tr between transformation calculus, with labels in $\mathcal{L}$, and selective λ-calculus with labels in $\mathcal{L}' = \mathcal{L} \cup \{cont\}$, and $\forall p \in \mathcal{L}, p <_{\mathcal{L}'} cont$, $<_{\mathcal{L}} = <_{\mathcal{L}'}$ everywhere else.

$$\begin{aligned} Tr(x) &= x \\ Tr(M \overset{\vee}{_p}\ x) &= \lambda_p x.Tr(M) \\ Tr(M \mathbin{\hat{p}} N) &= Tr(M) \mathbin{\hat{p}} Tr(N) \\ Tr(\{\}) &= \lambda_{cont} x.x \\ Tr(M \circ N) &= Tr(N) \mathbin{\widehat{cont}} Tr(M) \end{aligned}$$

cont in the above translation stands for continuation. When we look at a transformation from this point of view, it becomes a function that takes a continuation to form a new one, just as did partial continuations in [JD88].

Definition 1 *$S\lambda$ is the set of selective λ-terms, TC the set of terms in transformation calculus.*

$S\lambda^*$ is the set of selective λ-terms such that $\overset{\vee}{_{cont}}$ appears only as $\lambda_{cont} x.x$.

$S\lambda^*/(3)$ is the set obtained by mapping every term in $S\lambda^*$ to the set of all its converted forms by $(\lambda_{cont} x.M) \mathbin{\hat{p}} N \leftrightarrow \lambda_{cont} x.(M \mathbin{\hat{p}} N)$ $(x \notin FV(N))$.

$$
\begin{array}{lll}
m & ::= x \mid c(p{:}m \ldots) \mid \{p{:}m \ldots\} \mid \{p{:}m \ldots \mid x\} & \text{pattern} \\
F & ::= p{:}m \ldots \to E & \text{function case} \\
T & ::= p{:}m \ldots \to p{:}C \ldots & \text{transformation case} \\
C & ::= c \mid x \mid \{p{:}C \ldots\} \mid (E) & \text{closed expression} \\
E & ::= C \mid \text{let [rec] } m\ p{:}m \ldots = E \text{ in } E & \text{open expression} \\
 & \quad \mid \text{match } E \text{ with } m \to E\ (\mid m \to E)^* & \text{match case} \\
 & \quad \mid \text{fun } M(\mid M)^* \mid C\ p{:}C \ldots & \text{application} \\
 & \quad \mid \text{tra } T(\mid T)^* \mid E \circ E \mid E; E & \text{compositions}
\end{array}
$$

Figure 3. Grammar based on transformation calculus

Proposition 1 *By extension of the previous definition, Tr is an isomorphism from $(TC, \to)$ to $(S\lambda^*/(3), \to)$ (we extend "$\to$" from its definition on $S\lambda$):*

$$\forall M, N \in TC, M \to N \Leftrightarrow Tr(M) \to Tr(N)$$

From the above proposition we can deduce confluence.

Theorem 2 *The transformation calculus is confluent.*

Here are some examples of expressions in the transformation calculus. We adopted the syntax in figure 3. "`tra`" introduces a directly expressed transformation, but it is only syntactic sugar for "`fun ...-> { ...}`". We have two forms of composition, linked by the relation $f \circ g = g; f$. The distinction between closed and open expressions is only for parsing. The polymorphic types we give here extend those for selective λ-calculus. The two new symbols are the "`-o`" for types transformations, abbreviating types "`...-> {...}`" the way `tra` does for functions, and "`;a`" for row variables.

```
#(tra a:x b:y -> c:(x+y) d:(x-y)) o {1:true a:1 b:10 c:"a"} ;;
 {1:true c:11 c:"a" d:-9} : {1:bool c:int c:string d:int}

#let .p {p:x|y} = x ;;
 .p : (1:{p:'a ;b} -> 'a)

#let rec partition f:f = fun [] -> {l1:[] l2:[]}
#      | [h|t] -> if (f h) then:(tra l1:l -> l1:(h::l))
#            else:(tra l2:l -> l2:(h::l)) o partition f:f t ;;
 partition : (1:(1:'a -> bool) 2:'a list
                           -o l1:'a list l2:'a list)

#let rec it_list f:f = fun [] -> {}
#      | [h|t] -> (it_list f:f t) o (f h) ;;
```

```
it_list : (1:'a list f:(1:'a ;b -o ;b) ;b -o ;b)
```

The first example is the direct application of a transformation to a record. It shows a characteristic of transformation calculus: assignment to a label is not destructive, only reading is (it is a function application). As a result the same label may appear more than once in the same stream, holding values of different types.

".p" uses an extended pattern, which permits an easy selection of a field. We will use it as a postfix operator in the following.

In `partition`, values of the entry list are dispatched on two output lists, according to a discriminating function. Using transformations, only the modified list is acceded.

`it_list` is a functional that maps a list to a sequence of operations on a stream. It works like would do an imperative `foreach`. While `partition` is typable in a restricted system, this one needs row variables (`;b`). These take the end of the stream, which is left open.

4 States

It is quite easy now to see how we can use this calculus to express state transformations. Typically a calculation using state variables will take the form:

$$(M \overset{\vee}{}_{l_v} v) \circ T_n \circ \ldots \circ T_1 \circ (\{\} \widehat{}_{l_v} V)$$

Expressing it directly in the calculus may be not very speaking, so here is a more verbose version. We reverse the reading order, to get it left-to-right, by using ";".

$$\{l_v : V\}; T_1; \ldots; T_n; (\text{fun } l_v : v \to M)$$

That is, first initialize the variable by putting a value on the label l_v, then a sequence of transformations that may use l_v by getting the value and putting a new one in its place, as does the following,

$$T_k = (\{\} \widehat{}_{l_v} V \cdot P) \overset{\vee}{}_{l_v} v = \text{fun } l_v : v \to \ldots (\ldots; \{l_v : V\})$$

And finally an abstraction eliminates l_v from the stream, and uses its value. Integrating this sequence of calculation in another one that uses l_v too is not a problem since we may have more than one value on a label. During all this sequence the external value for l_v was conserved in second position, and comes back to first position after elimination, as would happen in a stack.

Example

```
#let rec while do:f end:g =
#       g ;
```

```
#       (fun ok:false -> {}
#          | ok:true -> f ; while do:f end:g) ;;
 while : (do:(;a -o ;b) end:(;b -o ok:bool ;a) ;b -o ;a)

#let gcd m n = {m:m n:n} ;
#       while do:(tra m:m n:n -> m:n n:(m mod n))
#             end:(tra n:0 -> ok:false n:0 | -> ok:true) ;
#       fun m:m n:n -> m ;;
 gcd : (1:int 2:int -> int)
```

`while` is a functional equivalent to that of C. It first applies an `end:` condition to the stream, and if it sets `ok:` to true, applies the `do:` transformation. The do-while construct is obtained by using "`while do:{} end:g`".

We use this functional to calculate the GCD of two integer by Euclid's algorithm. One could remark that this one is not very demonstrative since it can be written in one line in a purely functional style.

5 Stateful objects

Now, how do we implement stateful objects using this mechanism? The idea is to associate an object with a label. That is, for each object we should have a label to which its state would be attached. By making its use exclusive, we can assimilate modifications of the object's state to modifications of the value associated to the label, through a transformation.

This gives us a simple definition of the methods attached to an object. They are transformations accessing or modifying the value on its label. This means that at the transformation calculus level an object is separated into a state and a set of methods, the only relation between the two being an exclusive label.

To obtain the continuity between state and methods, we just need to add a concept of class definition. A class definition is a record in which each element represents a method. We have a specific label, *self*, and a specific variable name, *self* too, which we may only use in class definitions. The label will be used to access state variables, and the identifier for other methods. That is, most methods will be a combination of the following two forms.

$$(\{\} \widehat{_{self}}\, new \cdot P) \overset{\vee}{_{self}}\, old.$$

$$self.\mathrm{meth}_1; \ldots; self.\mathrm{meth}_n$$

We first get state variables in *old*, make functional calculations on them, and put a *new* state. Alternately we can call successively other methods of our object. Of course we need initialization and destruction methods too,

if we want to encapsulate labels. Their simplest form will be "`tra 1:x -> self:x`" and "`tra self:x -> 1:x`".

Here is an example illustrating class definition. The integer we use as screen state is in fact a dummy state. The only real modification is done by the low level operation `plot_screen` but by reading and writing the screen number we simulate a modification.

```
#let class screen = {plot:(fun x y self:n ->
#                            {self:n} o plot_screen nb:n x y)
#                    init:(tra x -> self:x)
#                    del:(tra self:x -> x)} ;;
 screen : {del:(self:int -o 1:int)
           init:(1:int -o self:int)
           plot:(1:int 2:int self:int -o self:int)} class

#let class point = {init:(tra x -> self:x)
#                   del:(tra self:x -> x)
#                   move:(tra x' y' self:{x y} ->
#                             self:{(x+x') (y+y')})
#                   plot:(fun scr:s self:{x y} ->
#                           {self:{x y}} o s.plot x y)} ;;
 point : {del:(self:(int*int) -o 1:(int*int))
          init:(1:(int*int) -o self:(int*int))
          move:(1:int 2:int self:(int*int) -o self:(int*int))
          plot:(scr:{plot:(1:int 2:int ;a -o ;b) ;c}
                self:(int*int) ;a -o self:(int*int) ;b)} class
```

If "*self*" is semantically identical in its label and name forms, they are not interpreted on the same level. As a name it only indicates its own class, and can be solved as a recursive class definition. We however need this specific name, since we generally need to generate an object, instance of a class, to use its methods. "*self*" means here that we are using methods of the same object, yet not created.

As a label, "*self*" is only solved when we create an object. A simple way to do it is the command "`let object` *objname* `=` *classname*". It first generates a new label l, and then defines a constant *objname*, the record of methods of *classname* where every occurrence of *self* (it only subsists as a label) is replaced by l. Here are the resulting method records.

```
#let object screen1 = screen ;;
 screen1 : {del:(screen#1:int -o 1:int)
            init:(1:int -o screen#1:int)
            plot:(1:int 2:int screen#1:int -o screen#1:int)}
```

```
#let object p1 = point ;;
 point : {del:(point#1:(int*int) -o 1:(int*int))
          init:(1:(int*int) -o point#1:(int*int))
          move:(1:int 2:int point#1:(int*int)
                         -o point#1:(int*int))
          plot:(scr:{plot:(1:int 2:int ;a -o ;b) ;c}
                point#1:(int*int) ;a -o point#1:(int*int) ;b)}
```

Once our objects created, we use them by composing methods from different objects. "`it`" is just an interactive variable which contains the previous result. We first initialize our objects, then move the point, plot it on the screen, and suppress its object.

```
#p1.init {0 0} ; screen1.init 1 ;;
 {point#1:{0 0} screen#1:1} : {point#1:(int*int) screen#1:int}

#it ; p1.move 10 15 ;;
 {point#1:{10 15} screen#1:1} :
                {point#1:(int*int) screen#1:int}

#it ; p1.plot scr:screen1 ;;
 {point#1:{10 15} screen#1:1} :
                {point#1:(int*int) screen#1:int}

#it ; p1.del ;;
 {1:{10 15} screen#1:1} : {1:(int*int) screen#1:int}
```

We have given here a practical approach, but it lacks encapsulation. Since `init` and `del` are accessible from anywhere in a calculation, we may well try to use a method on an object not yet initialized, or already destroyed, or even create more than one state for the same object (label).

In an ideally encapsulated scheme, the definition of the label should have the same locality as the object, that is all object should be defined locally to a calculation, being initialized at its beginning and destroyed at its end, with no initialization nor destruction method accessible during this calculation, nor explicit access to its label.

In this last scheme we have a construct "`let object` *objname* = *classname (init-value)* `in` *transformation*". *objname* is accessible only in the *transformation* part, and gives access to methods using a local label for the object state. These methods, except for the initialization one which is used only once on *init-value* when creating the object, should get an object state as input, and output a new one with same label. This is only a constraint on the use of the

self label. Such a scheme gives a perfect encapsulation, object states being accessible only through their methods, and having exactly the same locality.

Nonetheless, in an interactive environment for instance, one may want to define an object globally. In this case we must allow initialization and destruction methods, letting the user control the object's existence by himself, the methods being created by a "`let object` *objname* = *classname*" command at toplevel, like above. We lose in security, since we may try accessing to an object already destroyed, or still not created. More dangerous, we may create more than one instance of the same object, with same name and same state label. But at the same time it is a powerful way of interactive control: we may locally change the state accessed by an object, initializing it once more, and after that recover the original state, by destroying this temporary state. And this while keeping the same identity.

6 Conclusion

We presented here a calculus, the transformation calculus, which extends lambda calculus with labels and records. By selective currying of input as well as output of transformations, this calculus is able to express state modifications in a quite natural way.

We used this ability to construct in this calculus a representation of stateful objects, with different levels of encapsulation. One should notice that stateful may have other meanings that "holding a value". For instance, an object handling I/O may be seen as stateful, even if it does not need to keep a representation of its state. In this case, maintaining a label with a dummy value enforces sequencing for this object, which is the expected behavior.

This remark is important, since it means that we can write programs in this calculus without taking care of evaluation order. An appropriate type analysis should be enough to determine it, according to sequencing constraints introduced by data updates. Parallelization is then only a typing problem.

For this reason, a very important issue for transformation calculus is typing. Full polymorphic typing of this calculus is possible, however the inference algorithm is rather complex, and leaves constraints between variables. A more straightforward type inference algorithm exists. It is enough for all the types given in this paper, but is not, in theory, complete.

Complete or not, polymorphic type inference is interesting, since it solves typing of functional and imperative parts in a unified way, without introducing distinctions between types and effects [GL86, TJ92]. We shall continue in this direction, and study operational semantics for this calculus, exploiting the information given by types.

On the object-based side, we can see that what we have proposed here is

mostly a way of encapsulating a resource (I/O...), or some atomic information. If we want to represent more complex information we will need to combine different objects' states, or only parts of them. For that, we will have to rethink the notion of unique label, in some more general one, to permit, and restrict, operations like sharing.

参考文献

[AKG93] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus. Research report 31, DEC Paris Research Laboratory, May 1993.

[AR88] Norman Adams and Jonathan Rees. Object oriented programming in Scheme. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 277–288, 1988.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, Vol. 76, pp. 138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, 1984).

[GAK93] Jacques Garrigue and Hassan Aït-Kaci. Typing of selective λ-calculus. Technical Report 93-1, University of Tokyo, Department of Information Science, 1993.

[Gar92] Jacques Garrigue. Label-selective λ-calculus. Rapport de D.E.A., Université Paris VII, 1992.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 28–38, 1986.

[JD88] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 158–168, 1988.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 316–326, 1988.

[Oho92] Atsushi Ohori. A compilation method for ML-style polymorphic records. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 154–165, 1992.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 43–56, 1993.

[Red88] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 289–297, 1988.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, Vol. 2, No. 3, pp. 245–271, July 1992.

[W+90] Pierre Weis, et al. *The CAML Reference Manual, version 2.6.1.* Projet Formel, INRIA-ENS, 1990.