The Transformation Calculus

Jacques Garrigue

Research Institute for Mathematical Sciences
Kyoto University, Kitashirakawa-Oiwakecho
Sakyo-ku, Kyoto 606-01 JAPAN
Tel ++81-75-753-7211
Fax ++81-75-753-7272
E-mail garrigue@kurims.kyoto-u.ac.jp

Abstract. The lambda-calculus, by its ability to express any computable function, is theoretically able to represent any algorithm. However, notwithstanding their equivalence in expressiveness, it is not so easy to find a natural translation for algorithms described in an imperative way.
The transformation calculus, which only extends the notion of currying in lambda-calculus, appears to be able to correct this flaw, letting one implicitly manipulate a state through computations.
This calculus remains very close to lambda-calculus, and keeps most of its properties. We proved confluence of the untyped calculus, and strong-normalization in presence of a typing system.

1 Introduction

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

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

in place of

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

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

We now look at types. There, currying can be seen as isomorphism of types [3]:

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

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

The calculus we will present here generalizes currying to these two kinds of symmetries: between arguments, and between input and output. For the first one, we are just taking over the mechanism of *label-selective currying* developed previously [1, 11].

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

The resulting system, *transformation calculus*, is a conservative extension of lambda calculus. Why such a name? Because this essentially syntactic extension —semantics remain very similar— provides us with a new way of representing state transformations, *i.e.* state being represented by labeled input parameters, that may get returned by our term. Handling state as a supplementary parameter that gets returned with the result is not new. But by extending currying we get more flexibility, in two ways. First, since a part of the state is no more than a labeled parameter, we can dynamically extend it by simply adding a new parameter at some point in our term. Second, selective currying lets a transformation ignore parts of the state it doesn't need. They will just be left unmodified.

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

The rest of this paper is composed as follows. In Section 2 we introduce progressively the different features which form the transformation calculus. Section 3 is devoted to the formal definition of the transformation calculus. Sections 4 and 5 respectively define and give the fundamental properties of scope-free variables and a simply typed transformation calculus. Related works are presented in Section 6. Finally, Section 7 concludes. For lack of space, no proofs ar given in this paper[1].

2 Composition and streams

We first introduce informally and progressively the features of our calculus. We start from the classical pure lambda-calculus, that is[2]

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

with β-reduction

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

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

2.1 Implicit currying

Currying is the fundamental transformation by which multi-argument functions are encoded in the lambda-calculus. It can appear in abstractions as well as applications. For instance $f(a,b)$ will be encoded as $(b).(a).f$, and $\lambda(x,y).M$ becomes $\lambda x.\lambda y.M$.

This operation does not modify the nature of calculations, since clearly $(a,b).\lambda(x,y).M$ and $(b).(a).\lambda x.\lambda y.M$ reduce to the same $[a/x,b/y]M$ (provided x and y are distinct variables).

[1] They can be found, together with denotational semantics for the simply typed calculus, in the report version [9].

[2] Application, denoted by a dot, is written postfix, and is left associative.

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

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

where abstracted variables under the same λ should be distinct. Implicit currying is expressed by the two structural equivalences[3]:

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

where all x_i's are distinct.

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

2.2 Composition

The next step is to introduce a binary *composition* operator[4] (";") and a *transformation* constructor ("$\downarrow$").

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

A *transformation* is a term such that, provided enough input, it gets a transformation constructor at its head position.

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

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

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

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

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

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

[4] Both the dots of abstraction and application bind tighter than composition.

Composed with another term, it plays the same role as the $C = \lambda fxy.fyx$ combinator; but in a postfix way.

$$
\begin{array}{ll}
 & (a,b,c)\ .(sw;K) \\
\equiv_{.;} & (a,b,c)\ .sw;K \\
\xrightarrow{*} & (b,a,c)\ .\downarrow;K \\
\equiv_{.;} & (b,a,c)\ .(\downarrow;K) \\
\rightarrow_{\downarrow} & (b,a,c)\ .K \\
\xrightarrow{*} & \qquad (c)\ .b
\end{array}
$$

Since we are in the lambda-calculus, we can define the fix-point operator Y. We just define then loops in terms of this operator. The functional for a while-do loop can be defined as

$$
\begin{array}{l}
\text{while} = Y(\lambda whl. \\
\qquad \lambda(end, do).(end; \\
\qquad\quad \lambda b.\text{if } b \text{ then } do; (end, do).whl \text{ else } \downarrow))
\end{array}
$$

The end-condition is a transformation that adds to its input a boolean b, false to end, true to go on, leaving the rest in position. do may change the values from the input, but not their number. Such a functional works on a state of any size.

An imperative version of Euclid's algorithm for the greatest common divisor can then be written

$$
\begin{array}{l}
(\lambda(x).(x \neq 0, x).\downarrow, \\
\ \lambda(x,y).(y \bmod x, x).\downarrow).\text{while}; \\
\lambda(x,y).y
\end{array}
$$

We notice here an important difference between this "while" functional and something equivalent written using pairing. Here our end-condition only uses x, whereas a functional using pairing would have required it to receive the whole state even though y is not needed. This remark will become even more important when we will add to our calculus the power of *selective currying*.

2.3 Selective currying

Combining lambda-calculus and a stack machine should be enough to express algorithms both in their functional and imperative form. Variables are denoted by positions on the stack. However, these positions being relative to the stop of the stack, they change if we add anything on top of if: ***e.g.*** c is the third element in $(a,b,c).\downarrow$, but it becomes the 4$^{\text{th}}$ in $(a,b,c).\downarrow; (d).\downarrow \rightarrow (d,a,b,c).\downarrow$, where d is either a parameter for the next function or a new variable. That makes it quite uneasy to actually write algorithm using variable in such a calculus.

The possibility of mixing parameters and variables is good, since it means that we can see everything with a functional insight. What we would like is to have a more uniform way to handle a position. The idea is to use several named stacks simultaneously. As stacks were represented by tuples, we now use sets of named tuples, or *streams*. For instance, $\{\epsilon \Rightarrow (a,b), p \Rightarrow c, q \Rightarrow (d,e)\}$ denotes a set of 3 independent stacks: (a,b) named ϵ[5], (c) named p, and (d,e) named q. Defining a label as being a pair (*stack name,*

[5] ϵ is a default name, such that a tuple (a,b,c) without name will be interpreted as the stream $\{\epsilon \Rightarrow (a,b,c)\}$.

stack index), we can also write a stream as a set of labeled values: $\{\epsilon 1 \Rightarrow a, \epsilon 2 \Rightarrow b, p \Rightarrow c, q1 \Rightarrow d, q2 \Rightarrow e\}$ [6].

Using this we can write a transformation incrementing the value labeled i as

$$\lambda\{\epsilon \Rightarrow x, i \Rightarrow y\}.\{i \Rightarrow x + y\}.\downarrow$$

Since i is a named position its index is not modified by operations on other names.

We see here a new version of Euclid's algorithm, using labels for both the function and the while functional.

$$\begin{aligned}
\text{while} = {} & \lambda\{end \Rightarrow end, do \Rightarrow do\}. \\
& \quad end; \lambda\{ok \Rightarrow ok\}. \\
& \quad\; \text{if } ok \text{ then } do; \{end \Rightarrow end, do \Rightarrow do\}.\text{while else } \downarrow
\end{aligned}$$

$$\begin{aligned}
\text{gcd} = {} & \lambda(x, y).\{m \Rightarrow x, n \Rightarrow y\}.\downarrow; \\
& \{end \Rightarrow \lambda\{m \Rightarrow m\}.\{ok \Rightarrow m \neq 0, m \Rightarrow m\}.\downarrow, \\
& \;\; do \Rightarrow \lambda\{m \Rightarrow m, n \Rightarrow n\}.\{m \Rightarrow n \text{ mod } m, n \Rightarrow m\}.\downarrow\}.\text{while}; \\
& \lambda\{m \Rightarrow m, n \Rightarrow n\}.n
\end{aligned}$$

On such an example the addition of labels may look as pure verbosity, but what we obtain here is very close to what we would write as an imperative algorithm. We only have to add trivial abstractions of the form $\lambda\{m \Rightarrow m\}$ in order to transform an assignment-like syntax (where $\{m \Rightarrow M\}$ is read as $m := M$) into functions.

2.4 Stream behavior

The examples we presented above worked all right, but what happens with "incorrect" terms, that are not well-behaved?

We had already such terms in classical lambda-calculus. For instance, if we encode an if-then-else by a pair $\lambda s.(s\ t\ e)$, where s is expected to be an encoded boolean, and t and e the two cases, we expect in most cases t and e to be well-behaved, that is if t encodes a pair, then e should also encode a pair. Otherwise, we will have unexpected behavior trying to apply a projection on it.

This problem of behavior is even more pernicious with the transformation calculus. Again in an if-then-else we expect the two branches to have similar behavior. But even if the second one gives back a stream with more labels than the first, it may well not appear, as long as we only use transformations that only access labels present in the first stream. This is still an incoherence.

So, by *well-behaved*, we will mean here that for any acceptable input with same stream structure, a transformation should give back a stream with same labels. That is, its *stream-behavior*, the stream structure of the result with respect to the stream structure of the input, should not be dependent on encoded values in the input.

For instance,

$$\lambda b.\text{if } b \text{ then } \{l \Rightarrow M\}.\downarrow \text{ else } \downarrow$$

[6] We allow omitting the index when there is only one value in the stack, as for p.

is not well behaved since it returns either a stream with label l or an empty stream, depending on the value of b.

This is difficult to give a precise definition of *well-behaved* terms in an untyped framework, since it depends on what encodings we use. In a typed framework that amounts to subject reduction, and we give in Section 5 a simply typed transformation calculus that satisfies it (*i.e.* all typable terms are well-behaved).

2.5 Scope-free variables

We have insisted on how this calculus was a potential basis for an integration of imperative and functional styles in the design of algorithms. Here we introduce a general method to directly map the imperative notion of variable into the transformation calculus.

In fact, what we mean by *scope-free variable* is slightly stronger than a mutable variable. We call it scope-free, since it is not syntactically scoped like in structured programming, neither is it global. We can say that it is local to a sequence of transformations, composed together.

A scope free variable is essentially a name v whose use in labels is exclusively reserved in the concerned sequence of transformations. This sequence is delimited by the creation of the variable with value a, encoded $\{v \Rightarrow a\}.\!\downarrow$, and its destruction by an abstraction, $\lambda\{v \Rightarrow x\}.\!\downarrow$. Between these, all transformations using or modifying this variable should once take it (through abstraction) and then put it back (by application), identical or modified. Typically a modification can be written $\lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}.\!\downarrow$. That is, the sequence has form:

$$\{v \Rightarrow a\}.\!\downarrow; \ldots; \lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}.\!\downarrow; \ldots; \lambda\{v \Rightarrow x\}.M$$

Since some transformations may be functionals, the recognition of such a structure is not immediate, but for instance m and n in the last version of Euclid's algorithm are scope-free variables.

The most interesting property of scope-free variables is that, like scoped variables, they have no effect outside of the sequence they are used in. That is, we can use the same label v outside of the sequence our scope-free variable is local to, without interference. A scope-free variable may even be used in a subsequence of another scope-free variable using the same label:

$$\{v \Rightarrow a\}.\!\downarrow; \ldots; \underline{\{v \Rightarrow b\}.\!\downarrow; \ldots; \lambda\{v \Rightarrow x\}.\!\downarrow}; \ldots; \lambda\{v \Rightarrow x\}.\!\downarrow$$

In the underlined subsequence the external scope-free variable is identifiable by the label $v + 1$ but comes back to v after.

Still, we must be careful that scope-free variables are not variables in the meaning of lambda-calculus: they appear on a completely different level, that of labels. Nor are they pervasive like would be references. We do not add side-effects to functions, but just provide some implicit way to manipulate a "stream" of arguments. That means that a function that is not called directly on this stream (through composition) will not access the scope-free variables it contains, and as such cannot have any imperative behavior with respect to this stream. This is this limitation which permits us to assimilate scope-free variables with arguments, and still be a conservative extension of lambda-calculus.

We give two examples of the use of scope-free variables. The first one is a simple encoding of an imperative programming language *a la* Algol. The second one shows how scope-free variable are stronger than scoped ones.

Here is the program and its translation.

```
begin
  var x=5, y=10;
  x := x+y;
  begin
    var x=3;
    y := x+y
  end
  x := x-y;
  return(x)
end
```

$$
\begin{array}{l}
\{x \Rightarrow 5, y \Rightarrow 10\}.\downarrow; \\
\lambda\{x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x+y, y \Rightarrow y\}.\downarrow; \\
\quad \{x \Rightarrow 3\}.\downarrow; \\
\quad \lambda\{x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x, y \Rightarrow x+y\}.\downarrow; \\
\quad \lambda\{x \Rightarrow x\}.\downarrow; \\
\lambda\{x \Rightarrow x, y \Rightarrow y\}.\{x \Rightarrow x-y, y \Rightarrow y\}.\downarrow; \\
\lambda\{x \Rightarrow x, y \Rightarrow y\}.x
\end{array}
$$

We expect this program to evaluate to $5+10-(3+10)=2$.

$$
\begin{array}{rl}
 & \{x \Rightarrow 5, y \Rightarrow 10\}.\downarrow; \ldots \\
\{x \Rightarrow 5, y \Rightarrow 10\} & .\lambda \cdots .\{x \Rightarrow x+y, y \Rightarrow y\}.\downarrow; \ldots \\
\{x \Rightarrow 15, y \Rightarrow 10\} & .\{x \Rightarrow 3\}.\downarrow; \ldots \\
\{x1 \Rightarrow 3, x2 \Rightarrow 15, y \Rightarrow 10\} & .\lambda \cdots .\{x \Rightarrow x, y \Rightarrow x+y\}.\downarrow; \ldots \\
\{x1 \Rightarrow 3, x2 \Rightarrow 15, y \Rightarrow 13\} & .\lambda\{x \Rightarrow x\} \downarrow; \ldots \\
\{x \Rightarrow 15, y \Rightarrow 13\} & .\lambda \cdots .\{x \Rightarrow x-y, y \Rightarrow y\}.\downarrow; \ldots \\
\{x \Rightarrow 2, y \Rightarrow 13\} & .\lambda\{x \Rightarrow x, y \Rightarrow y\}.x \\
 & 2
\end{array}
$$

Note here that since we encode dynamic binding[7] for scope-free variables, we would get the same result even if the central part was a call to the same piece of code defined elsewere: with scope-free variable, even Basic's subprograms, which have no variable passing, would be a nice feature, since we can create a scope-free variable before the call to pass a parameter, and destroy it after.

The translation we propose here is a general one. By defining variables at the beginning of blocks and destroying them by abstractions at the end, we can translate any Algol-like program (with dynamic binding for mutable variables), even containing procedures and functions.

The above example still respects a scoping discipline: variables are created and destroyed in opposite order. To show the specificity of scope-free variables, we must disobey it.

Not respecting a scoping discipline seems quite dangerous for variables, and of little use in purely computing programs. However, if we think of IO's, then the situation is

[7] Dynamic binding is generally considered as bad, because destroying referential transparency. However, if we distinguish between static (defined only once, like λ-variables) and mutable variables, the notion of referential transparency for the last is not so clear. Since they are already not referentially transparent w.r.t. their values, the simpler modeling offered by dynamic binding can be seen as an advantage.

different. Consider a program with structure

$$\overline{A;\underline{B}}\underline{;C}$$

in which we want the console to be redirected in part $A;B$, and the screen to be changed in $B;C$. We suppose that we have mutable variables con and scr to indicate respectively which console and which screen should be used. Moreover we do not know which were the console and screen before entering A.

A dirty method is to use temporary variables c and s, to store the old values:

```
c:=con; con:=newc; A; s:=scr;
scr:=news; B; con:=c; C; scr:=s
```

The problem is that these temporary variables may be modified by error in A, B or C.

So a better solution is to use static variables, only set once:

```
let c = !con in
  con:=newc; A ;
  let s = !scr in
    scr:=newc; B ; con:=c; C ; scr:=s
end end
```

However, because of the scope discipline, c is still defined in C, whereas we do not need it anymore. We can see here an inconsistency between the scope of c, which is $A;B;C$, and its expected area of use, $A;B$.

We think that the scope-free variable way to do it is cleaner:

$$\begin{array}{l}\{con \Rightarrow newc\}.\!\downarrow; A; \{scr \Rightarrow news\}.\!\downarrow; B;\\ \lambda\{con \Rightarrow c\}.\!\downarrow; C; \lambda\{scr \Rightarrow s\}.\!\downarrow\end{array}$$

We didn't define any new variable, but did just temporarily hide the original value by the redirected one. And there are no "dangling" definitions (variables still defined out of their area of use).

3 Syntax of transformation calculus

In this section we define the untyped transformation calculus.

The definition is done in three steps. 1) We define streams[8], as a tool for defining the calculus. 2) We give a syntactic definition of terms in the transformation calculus, and add a structural equivalence on these terms[9]. 3) Then we define reduction rules for these equivalence classes.

---

[8] In [9], we give another definition of streams, permitting more commutations in the calculus, but here we use a simpler one.

[9] We could use all equivalences as directed reduction rules. This would result in a slightly more complicated system (*cf.* [1] for selective λ-calculus)

Definition 1 stream monoid. Let $\mathcal{L}$ be a set of names, $\mathcal{A}$ a set of values. $\mathcal{S}(\mathcal{A})$, the set of $\mathcal{L}$-streams on $\mathcal{A}$, is the set of the functions from $\mathcal{L}$ to the tuples of $\mathcal{A}$, such that only a finite number of tuples are not empty.

$$\mathcal{S}(\mathcal{A}) = \{s : \mathcal{L} \to \mathcal{A}^* \mid \sum_{l \in \mathcal{L}} |s(l)| \in \mathbb{N}\}$$

For a stream s, $\mathcal{D}_s = \{(l, n) \mid 1 \leq n \leq |s(l)|\}$ is the set of its defined positions. We write s as $\{l_1 n_1 \Rightarrow s(l_1)_{n_1}, \ldots, l_m n_m \Rightarrow s(l_m)_{n_m}\}$ by enumerating all its *defined positions*.

Concatenation on streams, the monoidal operation, is the name-wise concatenation of tuples:

$$(r \cdot s)(l) = r(l) \cdot s(l)$$

Notations In the following definitions we will use the abbreviations $A \not\pitchfork B$ for $A \cap B = \emptyset$, $FV(M)$ for the free variables of M, and $V(R)$ for the values contained in the stream R.

Definition 2. Terms of the transformation calculus, or Λ_T, are those generated by M in the following grammar, where variables should be distinct in abstractions. Composition has lower priority than dots.

$M ::=$	x	variable
	$\mid \downarrow$	transformation constructor
	$\mid \lambda\mathcal{S}(x).M$	abstraction
	$\mid \mathcal{S}(M).M$	application
	$\mid M; M$	composition

They are considered modulo $\equiv$, the minimal congruence defined by the closure of the following equalities.

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

$$\begin{array}{rll}
(R.M_1); M_2 \equiv_{.;} & R.(M_1; M_2) & \\
(\lambda R.M_1); M_2 \equiv_{\lambda;} & \lambda R.(M_1; M_2) & V(R) \not\pitchfork FV(M_2) \\
(M_1; M_2); M_3 \equiv_{;} & M_1; (M_2; M_3) &
\end{array}$$

Equalities $\equiv_{.}$ and $\equiv_{\lambda}$ are derived from the monoidal structure. $\equiv_{.;}$, $\equiv_{\lambda;}$ and $\equiv_{;}$ are intuitive. Equality $\equiv_{.\lambda}$ is the "symmetrical" of β-reduction: when they have no common names, application and abstraction may switch, and permit earlier reductions.

Substitutions are done in the same way as for lambda-calculus, composition not interacting with variable binding. Terms will always be considered modulo α-conversion. That is $\lambda\{l \Rightarrow x\}.M \equiv \lambda\{l \Rightarrow y\}.[y/x]M$ when $y \notin FV(M)$.

Definition 3. "$\rightarrow$" is defined on transformation calculus terms by β-reduction and $\downarrow$-elimination[10].

$$\begin{array}{rcl} \{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M & \rightarrow_\beta & [N/x].M \\ \downarrow; M & \rightarrow_\downarrow & M \end{array}$$

$\xrightarrow{*}$ is the reflexive and transitive closure of $\rightarrow$.

Selective λ-terms and β-reduction define the selective λ-calculus.

Theorem 4. *Transformation calculus is confluent.*

The proofs are given in [9]. Confluence of transformation calculus is obtained from selective λ-calculus through a translation into it.

4 Scope-free variable encoding

We cannot expect to give a precise definition of *scope-free variable* in the transformation calculus, where it is only encoded. It appears as an intuitive notion of a variable whose locality is not syntactical but operational. We will define it outside of the calculus.

For this we use a framework in which a program is a sequence of operations. Operations can themselves contain programs, but these are independent, and may not have side-effects on the external sequence.

Definition 5. A *scope-free variable* is some way to create, modify and destroy a value such that:

1. these operations may appear in different syntactic entities, which may be used independently.
2. a *closed use* of this variable is obtained when a creator, some modifiers, and a destructor result in a *modification sequence*.
3. its closed use in a modification sequence has no side-effect outside it.

A consequence of this definition is the hiding property we insisted on. The same variable may have several independent closed uses, with modification sequences included in one another, and there is no problem as long as we do not try to modify the value from one use inside another's modification sequence.

As we introduced in Section 2, elementary creators, modifiers and destructors in transformation calculus are respectively $\{v \Rightarrow M\}.\downarrow$, $\lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}.\downarrow$ and $\lambda\{v \Rightarrow x\}.\downarrow$. But we can think of more complex ones, acting simultaneously on multiple variables, taking arguments, or returning results.

To ensure that we have a correct scope-free variable encoding here, we must verify the third point of the definition, which says that it has no effect outside the sequence it is used in.

Proposition 6. *The scope-free variable encoding into the transformation calculus ensures locality to the modification sequence.*

[10] We chose to make $\downarrow$-elimination a reduction rule rather than a structural equality because it reduces the size of terms, while the structural equalities of Definition 2 do not change it.

As we have seen, thanks to this property, scope-free variables are not only more flexible than classical scoped variables, but can replace them in most of their uses. Particularly, in functional language they can replace "disciplined" references (which do not go out of their scope), without the need of a specific evaluation strategy. Their only limitation is that —in the transformation calculus— one cannot export them like references, since they are linked to an explicit name. However, this is a limitation of the label system we use, and not of scope-free variable in themselves: one can add a syntactical scope to scope-free variables [10]. The real point about them is that the use of a (now scoped) scope-free variable is not restricted by that syntactical scope[11] (which is only a problem of naming), like with the stack discipline, but by its *life area*, or modification sequence (its real operational scope).

5 Simply typed transformation calculus

To obtain a simply typed form of transformation calculus, we annotate variables with some type in abstractions, just the same way it is done in lambda calculus. But first we must define what are these types.

The two most important novelties are that, first, stream types are introduced, and second, that function type are not from any type to any other, but only from stream types to stream or base types. This last particularity "flattens" types, but still contains as a subset all simple types of lambda-calculus.

Definition 7. Simple types in the transformation calculus are generated by t in the following grammar.

$$\begin{array}{lll} u & ::= u_1 \mid \ldots & \text{base types} \\ r & ::= \mathcal{S}(t) & \text{stream types} \\ w & ::= u \mid r & \text{return types} \\ t & ::= r \rightarrow w & \text{types} \end{array}$$

The same label may not appear more than once in the same stream type; stream types are equal up to different orders, and $(\{\} \rightarrow \tau) = \tau$, for short.

Definition 8. A term in the simply typed transformation calculus is constructed according to the following syntax.

$$M \ ::= \ x \mid \downarrow \mid \lambda\mathcal{S}(x{:}t).M \mid \mathcal{S}(M).M \mid M;M$$

Finally the relation between terms and types is given in the following definition.

Definition 9. A *type judgement*, written $\Gamma \vdash M : \tau$, expresses that the term M has type τ in the context Γ. Induction rules for type judgements are given in figure 1.

Rules (I,II,III) are the traditional ones for typed lambda calculus, simply extended to streams. We can go back to it by limiting labels in streams to sequences of integers starting from 1 (that is, in the above rules, having only $l = \epsilon 1$).

[11] This is true with references too, but their operational scope is only defined by garbage collection.

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

$$\frac{\Gamma[x \mapsto \theta] \vdash M : r \to w}{\Gamma \vdash \lambda\{l \Rightarrow x{:}\theta\}.M : (\{l \Rightarrow \theta\} \cdot r) \to w} \tag{II}$$

$$\frac{\Gamma \vdash M : (\{l \Rightarrow \theta\} \cdot r) \to w \qquad \Gamma \vdash N : \theta}{\Gamma \vdash \{l \Rightarrow N\}.M : r \to w} \tag{III}$$

$$\Gamma \vdash \downarrow : \{\} \tag{IV}$$

$$\frac{\Gamma \vdash M : r_1 \to r_2 \qquad \Gamma \vdash N : r_2 \to w}{\Gamma \vdash M; N : r_1 \to w} \tag{V}$$

$$\frac{\Gamma \vdash M : r_1 \to r_2}{\Gamma \vdash M : (r_1 \cdot r) \to (r_2 \cdot r)} \tag{VI}$$

Fig. 1. Typing rules for simply typed transformation calculus

Rule (IV) types the constant $\downarrow$. However it will most often need the cooperation of rule (VI), transformation subtyping, which expresses that any transformation may be applied to labels it is not concerned with: they will simply be rejected to the result. For instance, it gives to $\downarrow$ any symmetrical type ($r \to r$). Rule (V) types composition: M is applied to the result stream of N, and re-abstracted by its abstraction part. Here again, we need the collaboration of rule (VI) to extend the types of either M or N.

Proposition 10 subject reduction. *If $\Gamma \vdash M : \tau$ and $M \to N$ then $\Gamma \vdash N : \tau$.*

Proposition 11 strong normalization. *If $\Gamma \vdash M : \tau$ then there is no infinite reduction sequence starting from M.*

This last property is interesting, since it is general belief that introducing mutables suppresses strong normalization: we keep it here, because all values used by a term appear in its type.

6 Related works

Since transformation calculus only happens to be able to represent state, its origin is not to be found in the field of semantics of stateful languages. It is rather based on two independent threads of work. The first one is the Categorical Combinatory Logic [6], in which composition and currying play a central role. The direction seems opposed: one encodes lambda-calculus into CCL (or its abstract machine version, the CAM [5]), while transformation calculus extends lambda-calculus. But the intuition that algorithmicity can be found in the structures of the lambda-calculus itself is the same.

The second one is process calculi. Their use of names for communication is similar to the principle of the transformation calculus. In [2], Boudol proposes the γ-calculus. The base is lambda-calculus, but applications express emissions of messages and abstractions their reception, while multiple terms can be evaluated simultaneously. Milner's π-calculus [17] proceeds alike, and by labeling with names applications and abstractions, it allows the use of multiple channels. The fundamental difference with our calculus is that non-determinism of the receptor of a message make these calculi divergent, while our terms are syntactically sequenced in order to keep determinism.

After these somewhat different directions, our claims makes necessary to look at the larger literature concerning modeling of mutables in Algol, Lisp, and modern functional programming languages. Algol is the closest to our system, since scope-free variables cannot be used out of their *life area*, like with Algol's stack discipline, where a variable cannot be exported out of its scope.

Landin first proposed an encoding of Algol 60 into the lambda calculus [13]. But the problem was not solved: "The semantics of *applicative expressions* can be specified formally without the recourse to a machine. [...] With *imperative applicative expressions* on the other hand it appears impossible to avoid specifying semantics in terms of a machine".

After a number of attempts, including marked stores [16] and subtler models [15], a denotational semantics was finally obtained with Oles and Reynolds category-theoretic models [19, 21]. The essential idea is to define blocks as functions that can be applied to a range of states with various shapes, but do not change their shapes. However, inside the block, state is temporarily extended with local variables. Thus, they do not appear in its meaning. Our approach shares a lot with this view, since we syntactically "expand" and "shrink" our state when we create and delete a scope-free variable.

If we go out of the Algol tradition, we can forget about the stack discipline. As a result, most systems give a formalization of references. So does the λ_v-S-calculus [7] for Scheme, and a call-by-value reduction strategy. With effect inference [14], restrictions on the reduction strategy can be reduced, and, for instance, parallelism can be introduced.

Still, we feel more concerned by systems going the other way, starting without a specific reduction strategy. There are a number of them, which enforce single-threadedness of variables by various typing disciplines [12, 20, 22, 23, 24]. We can see an intuitive relation between the way scope-free variables are used and linear types, but still we are not relying to typing for single-threadedness.

We actually do it in a syntactical way. In that we are very close to λ_{var} [18, 4]. In fact, even the structures of the calculus have similarities: like us, they use the linear structure of spines to ensure single-threadedness. They have rules to propagate the values of mutable variables along the spine of a term, like does our structural equivalences for labeled arguments, and their **return**-elimination rule ((**return** $N) \triangleright \lambda x.M \rightarrow (\lambda x.M)N$, *cf.* [18]) can be seen as a variant of $\downarrow$-elimination ($\downarrow; M \rightarrow M$) including value-passing. The essential difference is that, since we use the same mechanism for scope-free variables and value-passing, we obtain a more unified calculus. In particular, the fact they are encoding references means that they must do some kind of garbage collection (their **pure** construct) to convert a value obtained using mutable variables into a purely functional one. In the transformation calculus, since we explicitly delete variables, we do not need such an "impure" purifier.

7 Conclusion

We proposed the transformation calculus as an extension of currying in the lambda-calculus permitting both functional and imperative encoding of algorithms.

We think this gives interesting answers to the two sides of the relation function/algorithm: as a demonstration of the relation between lambda-calculus and algorithms, and as a basis for functional languages handling states and sequentiality problems.

Still there are many topics left to explore. Typing is one of them. If we are to write program in this calculus, it is even unavoidable, since we must be able to verify that scope-free variables are correctly used. We presented here a simply typed system. We propose in [8] a polymorphic version of it, extending that for selective λ-calculus [11]. This could be completed by the introduction of linear types [24]: in the transformation calculus, variables are single threaded (operations on them are sequenced), but there is no restriction to their duplication. This is particularly a problem with IO: we can semantically create fictitious worlds, but there is no way to implement them. Moreover, using linear types within the transformations calculus relieves the programmer of most of the grudge of the linear style, since sending back an argument is easy.

Compilation, which is easy with stores, is complex here. The final goal would be to eliminate label information, but the possibility to compose a term with a variable makes impossible to reach it in the general case. There is no problem if we use non-polymorphic typing, since everything can be compiled into tuples, but the polymorphic case is still open.

One strength of the transformation calculus is its system of labels. We may be interested in extending it. For instance, the possibility to generate new label names will give unique identifiers for scope-free variable, and avoid the hiding of a label in those subsequences which create new variables on this label. A more structured label space even enables the use of object-based techniques, and solve the restrictions of dynamic binding.

Last, the similarities between this calculus and process calculi suggest that it might be used to express some forms of parallelism. If one looks at the way data flows in our terms, lots of reminiscences of the dataflow model may be seen. A topic like compilation of the calculus into this model looks interesting.

References

1. Hassan Ait-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. In *Proc. of the Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 24–40, Bombay, India, 1993. Springer-Verlag LNCS 761.
2. Gerard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *Proceedings of TAPSOFT '89*, pages 149–161, Berlin, Germany, 1989. Springer-Verlag, LNCS 351.
3. Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical Report LIENS-90-14, LIENS, July 1990.
4. Kung Chen and Martin Odersky. A type system for a lambda calculus with assignments. In *Proc. of the International Conference on Theoretical Aspects of Computer Software*, pages 347–363, 1994.

5. Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, 1987.
6. Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1993. First edition by Pitman, 1986.
7. M. Felleisen and D.P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.
8. Jacques Garrigue. *Label-Selective Lambda-Calculi and Transformation Calculi*. PhD thesis, University of Tokyo, Department of Information Science, March 1995.
9. Jacques Garrigue. The transformation calculus (revised version). Technical report, Kyoto University Research Institute for Mathematical Sciences, Kyoto 606-01, Japan, 1995. *cf.* `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/`.
10. Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Proc. of the Fuji International Workshop on Functional and Logic Programming*. World Scientific, Singapore, to appear.
11. Jacques Garrigue and Hassan Ait-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994.
12. J.C. Guzman and P. Hudak. Single threaded polymorphic calculus. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.
13. P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8(2-3):89–101 and 158–165, February 1965.
14. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, 1988.
15. Albert R. Meyer and Kurt Sieber. Toward fully abstract semantics for local variables. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.
16. R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.
17. Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, pages 203–246. NATO ASI Series, Springer Verlag, 1992.
18. Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 43–56, 1993.
19. F.J. Oles. Type algebras, functor categories, and block strcutures. In N. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 15, pages 543–573. Cambridge University Press, 1985.
20. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 71–84, 1993.
21. John C. Reynolds. The essence of ALGOL. In de Bakker and van Vliet, editors, *Proc. of the International Symposium on Algorithmic Languages*, pages 345–372. North Holland, 1981.
22. Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hugues, editor, *Proc. ACM Symposium on Functional Programming and Computer Architectures*, pages 192–214. Springer Verlag, 1991. LNCS 523.
23. Philip Wadler. Comprehending monads. In *Proc. ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.
24. Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

This article was processed using the LaTeX macro package with LLNCS style