The Transformation Calculus

Jacques Garrigue

Research Institute for Mathematical Sciences
Kyoto University, Kitashirakawa-Oiwakecho
Sakyo-ku, Kyoto 606-01 JAPAN
E-mail garrigue@kurims.kyoto-u.ac.jp

October 23, 1995

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 prove here confluence, strong-normalization in presence of a typing system, and present a model of the typed calculus.

Topics input/output models and state transformers, lambda calculus (formal aspects), formal semantics.

1 Introduction

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

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

in place of

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

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

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

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

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

The calculus we will present here generalizes currying to these two kinds of symmetries: between arguments, and between input and output. For the first one, we are just taking over the mechanism of *label-selective currying* developed previously [AKG93, GAK94].

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. In this process we are brought to define *streams*, which are formalized in Section 3. Section 4 is devoted to the formal definition of the transformation calculus. Sections 5, 6 and 7 respectively define and give the fundamental properties of scope-free variables, a simply typed transformation calculus, and semantics for this calculus. Related works are presented in Section 8. Finally, Section 9 concludes. Proofs are given in appendix.

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[1]

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

with β-reduction

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

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 $\lambda x.\lambda y.M$.

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

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). Currying can be extended to an arbitrary number of arguments, *i.e.* $\lambda(x_1,\ldots,x_n).M$ is encoded as $\lambda x_1.\cdots.\lambda x_n.M$. As long as we encode similarly applications and abstractions, no problem should appear.

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

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

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

If all x_i's are distinct then

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

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

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

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

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

2.2 Composition

The next step is to introduce a binary *composition* operator (";")[2] 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.

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

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

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

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:

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

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

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

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

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

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

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

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. However, in chosing a reduction system rather than an equational theory to express our calculus, we are interested in giving some meaning to the reductions themselves. We can think of various meanings, like complexity, sequentiality constraints, *etc...* If we do such an analysis, then we see that, with simple implicit currying, we need much more reduction steps than should be necessary. That is,

when we want to access the 5^{th} element of a tuple we have to extract successively all the elements before it, and put them back:

$$\lambda(x_1, x_2, x_3, x_4, x_5).(x_5, x_1, x_2, x_3, x_4).\downarrow$$

Even more than the number of reductions, we should be preoccupied by the fact we have accessed four unrelated values to move only one. And this in an asymmetrical way, values after the fifth element being left untouched.

2.3.1 Indexed streams

The answer to this problem comes from the canonical injection of tuples into records, as it can be found functional languages like Standard ML or LIFE. That is

$$(x_1, \ldots, x_n) \equiv \{1 \Rightarrow x_1, \ldots, n \Rightarrow x_n\}$$

We will just consider tuples as a particular case of *streams* (the name we give to these numerically indexed records). We use streams to access directly the arguments we are interested in. For instance we write the previous transformation

$$\lambda\{5 \Rightarrow x\}.\{1 \Rightarrow x\}.\downarrow$$

We extract the 5^{th} element from the input, and put it in first position.

By symmetry we can use such incomplete streams in applications as well as abstractions. The transformation $\{5 \Rightarrow a\}.\downarrow$ inserts a before the fifth argument of its input, pushing up all its followers by one.

$$(b_1, b_2, b_3, b_4, b_5, b_6).\{5 \Rightarrow a\}.\downarrow \rightarrow_\beta (b_1, b_2, b_3, b_4, a, b_5, b_6).\downarrow$$

The stream we apply our transformation to can be incomplete too, like in the following case.

$$\{2 \Rightarrow a, 5 \Rightarrow b, 7 \Rightarrow c\}.\{5 \Rightarrow d\}.\downarrow \rightarrow_\beta \{2 \Rightarrow a, 5 \Rightarrow d, 6 \Rightarrow b, 8 \Rightarrow c\}.\downarrow$$

Generally, we must define a concatenation operation on streams, compatible with the partial isomorphism between streams and tuple. This is done by shifting indexes in the inserted stream according to those present in the original one. The algorithm doing that will be detailed in the next section. The important point is that we have a reciprocal operation, sub-stream extraction, which we can use to separate a stream into two parts, forming it back by concatenation.

$$\begin{array}{ll} & \{1 \Rightarrow a, 2 \Rightarrow b, 3 \Rightarrow c, 5 \Rightarrow d, 7 \Rightarrow e\}.\lambda\{2 \Rightarrow x, 3 \Rightarrow y\}.M \\ \equiv & \{1 \Rightarrow a, 3 \Rightarrow d, 5 \Rightarrow e\}.\{2 \Rightarrow b, 3 \Rightarrow c\}.\lambda\{2 \Rightarrow x, 3 \Rightarrow y\}.M \\ \overset{*}{\rightarrow} & \{1 \Rightarrow a, 3 \Rightarrow d, 5 \Rightarrow e\}.[b/x, c/y]M \end{array}$$

This operation can be applied to the abstraction part too:

$$\begin{array}{ll} & \{2 \Rightarrow a, 4 \Rightarrow b\}.\lambda\{2 \Rightarrow x, 3 \Rightarrow y\}.M \\ \equiv & \{3 \Rightarrow b\}.\{2 \Rightarrow a\}.\lambda\{2 \Rightarrow x\}.\lambda\{2 \Rightarrow y\}.M \\ \rightarrow_\beta & \{3 \Rightarrow b\}.\lambda\{2 \Rightarrow y\}.[a/x]M \\ \equiv_\lambda & \lambda\{2 \Rightarrow y\}.\{2 \Rightarrow b\}.[a/x]M \end{array}$$

In the second line we decompose both abstraction and application to permit β-reduction in the third one. The $\equiv_\lambda$ equivalence in the last line is there to switch unrelated abstractions and applications, and let the reduction progress smoothly. We will argue later its coherence with the rest of the system.

Thanks to these streams we can now write algorithms using the power of something quite close to a direct-access stack machine. It is slightly different since reading a position in the stream destroys this position, but writing resulting in a symmetrical insertion this is not a problem. Here is a transformation incrementing the fifth position in a stream.

$$\begin{array}{lrl} & (a,b,c,d,e) & .\lambda\{5\Rightarrow x\}.\{5\Rightarrow x+1\}.\downarrow \\ \equiv & (a,b,c,d) & .\{5\Rightarrow e\}.\lambda\{5\Rightarrow x\}.\{5\Rightarrow x+1\}.\downarrow \\ \rightarrow_\beta & (a,b,c,d) & .\{5\Rightarrow e+1\}.\downarrow \\ \equiv & (a,b,c,d,e+1) & .\downarrow \end{array}$$

2.3.2 Naming positions

The problem of such a system is that since the indexes in the stream may change with each transformation we apply to it, we have no uniform way to address a defined position in it. This increments the fifth position by the first (which is destroyed):

$$\lambda\{1\Rightarrow x, 5\Rightarrow y\}.\{4\Rightarrow x+y\}.\downarrow$$

The position we addressed as 5^{th} before the transformation must become the 4^{th} after it, since we do not distinguish arguments (x) from mutable variables (y).

This possibility of mixing is good, since it means that we can see everything with a functional insight. However we would like to have a more uniform way to handle a position. Going on with our analogy between streams and records, we will accept to have named fields in our streams. So that we can write the previous incrementer as

$$\lambda\{1\Rightarrow x, i\Rightarrow y\}.\{i\Rightarrow x+y\}.\downarrow$$

Since i is a named position its index is not modified by the extraction of the first one.

A problem may appear when we concatenate two streams containing the same named position. On records this operation has two definitions. Either we just refuse to do it (symmetrical concatenation), either we accept, take for this field the value in one operand, and just forget the value in the other (asymmetrical concatenation). Since concatenation is already asymmetrical on numerical indexes, there is no point in refusing to do such a thing. However we cannot erase a value since we would lose confluence: $\{i\Rightarrow b\}.\{i\Rightarrow a\}.\lambda\{i\Rightarrow x\}.\lambda\{i\Rightarrow y\}.M \xrightarrow{2}_\beta [a/x, b/y]M$ but $\{i\Rightarrow b\}.\lambda\{i\Rightarrow x\}.\lambda\{i\Rightarrow y\}.M$ doesn't. So we just add a numerical index to our position name. That is, we view both numerical and name positions as their injections into their product, the set of labels. We have a default name ϵ such that the index n is in fact ϵn, and we add 1 to names so that p is $p1$. With that we define easily

$$\{p\Rightarrow b\}.\{p\Rightarrow a\}.M \equiv \{p1\Rightarrow a, p2\Rightarrow b\}.M$$

This works right with the above example, which becomes $\{i1\Rightarrow a, i2\Rightarrow b\}.\lambda\{i1\Rightarrow x, i2\Rightarrow y\}.M$.

Going on with our comparison with stacks, we have now as many stacks as we have names, each of them handled through indexed extraction and insertion.

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

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

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

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

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

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 6 a simply typed transformation calculus that satisfies it (*i.e.* all typable terms are well-behaved).

2.5 Scope-free variables

Up to this point we have progressively enriched the lambda-calculus with new constructs. The transformation calculus is approximately the result of this process. We have insisted

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

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

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

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

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

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

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

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

Still, we must be careful that scope-free variables are not variables in the meaning of lambda-calculus: they appear on a completely different level, that of labels. Nor are they pervasive like would be references. We do not add side-effects to functions, but just provide some implicit way to manipulate a "stream" of arguments. That means that a function that is not called directly on this stream (through composition) will not access the scope-free variables it contains, and as such cannot have any imperative 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 *à 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[3] 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 different. Consider a program with structure

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

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

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
```

[3]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.

The problem is that these temporary variables may be modified by error in A, B or C. So a better solution is to use static variables, only set once:

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

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

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

$$\{con \Rightarrow newc\}.\downarrow; A; \{scr \Rightarrow news\}.\downarrow; B;$$
$$\lambda\{con \Rightarrow c\}.\downarrow; C; \lambda\{scr \Rightarrow s\}.\downarrow$$

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

3 Stream monoid

The transformation calculus is essentially defined in terms of operations on streams. We give one definition for their set here, but transformation calculus can be defined on any "reversible" monoid: the monoid operation, or concatenation, gives us uncurrying, while its inverse, or extraction, gives us currying.

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

Preliminaries

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

Definition 1 (stream) *The set $\mathcal{S}(\mathcal{L}, \mathcal{A})$ of streams on a domain $\mathcal{A}$ is the set of finite partial functions from $\mathcal{L}$ to $\mathcal{A}$.*

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

Notations

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

- $(a_1, \ldots, a_n) = \{1 \Rightarrow a_1, \ldots, n \Rightarrow a_n\}$

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

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

and is reversible.

This example gives our first calculus, omiting selective currying. In the general case, we need a more complex definition, permitting label operations. It is based on a notion of n^{th} free position in a stream, which, while intuitively clear —just imagine that undefined labels point to free positions—, looks a little complex once formalized.

Definition 2 (free position) *1. The n^{th} position on p in a stream r is said to be* occupied *if $pn \in \mathcal{D}_r$. It is* free *otherwise, and $\mathcal{F}_r = \mathcal{L} \setminus \mathcal{D}_r$ is the set of these free positions.*

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

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

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

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

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

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

Proposition 1 (reversibility) *ϕ_r is a bijection from $\mathcal{S}$ to $\{s \in \mathcal{S} \mid \mathcal{D}_r \cap \mathcal{D}_s = \emptyset\}$. $\psi_r \circ \phi_r = id_{\mathcal{S}}$.*

Definition 3 (concatenation and extraction)

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

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

Proposition 2 (monoid) *Concatenation as in Definition 3 is an associative application $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$, accepting $\{\}$ as neutral element.*

$$
\begin{array}{rcll}
l & ::= & pn & p \in \mathcal{L}_s, n \in \mathcal{N} \\
M & ::= & x & \text{variable} \\
 & | & \downarrow & \text{transformation constructor} \\
 & | & \lambda\{l \Rightarrow x, \ldots\}.M & \text{abstraction} \\
 & | & \{l \Rightarrow M, \ldots\}.M & \text{application} \\
 & | & M; M & \text{composition}
\end{array}
$$

Figure 1: Syntax of the transformation calculus

$$
\begin{array}{rcll}
S.R.M & \equiv_{\cdot} & (R \cdot S).M & \\
\lambda R.\lambda S.M & \equiv_{\lambda} & \lambda(S \cdot R).M & V(R) \not\cap V(S) \\
R.\lambda S.M & \equiv_{\cdot\lambda} & \lambda\psi_R(S).\psi_S(R).M & FV(R) \not\cap V(S), \mathcal{D}_R \not\cap \mathcal{D}_S \\
 & & & \\
(R.M_1); M_2 & \equiv_{\cdot;} & R.(M_1; M_2) & \\
(\lambda R.M_1); M_2 & \equiv_{\lambda;} & \lambda R.(M_1; M_2) & V(R) \not\cap FV(M_2) \\
(M_1; M_2); M_3 & \equiv_{;} & M_1; (M_2; M_3) &
\end{array}
$$

Figure 2: Structural equivalences

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

Example 3 (stream concatenation)

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

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

4 Syntax of transformation calculus

In this section we define the untyped transformation calculus, and the selective λ-calculus as a subsystem of it.

The definition is done in two steps. 1) We give a syntactic definition of terms in the transformation calculus, and add a structural equivalence on these terms[4]. 2) Then we define reduction rules for these equivalence classes.

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

[4]We could use all equivalences as directed reduction rules. This would result in a slightly more complicated system (*cf.* [AKG93] for selective λ-calculus)

Definition 4 *Terms of the transformation calculus, or Λ_T, are those generated by M in the grammar of figure 1, where variables should be distinct in abstractions, and labels distinct in streams. Composition has lower priority than dots.*

They are considered modulo $\equiv$, the minimal equivalence relation defined by the closure of the equalities in figure 2.

The selective λ-calculus Λ_S is the subset formed by those terms which do not contain ; nor $\downarrow$.

Equalities $\equiv_.$ and $\equiv_\lambda$ are derived from the monoidal structure. $\equiv_{.;}$, $\equiv_{\lambda;}$ and $\equiv_;$ are intuitive.

Equality $\equiv_{.\lambda}$ is the "symmetrical" of β-reduction. It comes from the need to close the equality

$$(R' \uplus S').\lambda(R \uplus S).N \equiv_. \psi_R(S').R'.\lambda S.\lambda\psi_S(R).N,$$

with $\mathcal{D}_{R'} = \mathcal{D}_R, \mathcal{D}_{S'} = \mathcal{D}_S$ and $V(S) \not\bowtie V(R)$. If we take $M = \lambda\phi_S^{-1}(R).N$, and apply $R'.\lambda S.M$ to $\psi_R(S')$, then $\equiv_{.\lambda}$ preserves confluence: it gives

$$(R' \uplus S').\lambda(R \uplus S).N \equiv \psi_R(S').\lambda\psi_R(S).\psi_S(R').\lambda\psi_S(R).N.$$

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

Definition 5 *"$\rightarrow$" is defined on transformation calculus terms by β-reduction and $\downarrow$-elimination*[5]*.*

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

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

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

Theorem 1 *Selective λ-calculus is confluent.*

Theorem 2 *Transformation calculus is confluent.*

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

Proposition 3 *Normal terms are generated by N in the following grammar, where streams and anti-streams may be empty.*

$$\begin{array}{lll} H & ::= & x \mid \downarrow \\ F & ::= & \{l \Rightarrow N, \ldots\}.x \mid F; \lambda\{l \Rightarrow x, \ldots\}.F \\ N & ::= & \lambda\{l \Rightarrow x, \ldots\}.\{l \Rightarrow N, \ldots\}.H \mid \lambda\{l \Rightarrow x, \ldots\}.(F; N) \end{array}$$

This formalizes our intuition that, in a normal form, a composition only subsists when its left side cannot be reduced to a transformation.

[5]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 4 do not change it.

5 Scope-free variable encoding

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

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

Definition 6 *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. For instance, in

$$\lambda\{1 \Rightarrow x\}.\{a \Rightarrow 2, b \Rightarrow x\}.\downarrow;$$
$$\lambda\{a \Rightarrow x, b \Rightarrow y\}.\{a \Rightarrow x \times y, b \Rightarrow x - y\}.\downarrow;$$
$$\lambda\{a \Rightarrow x, b \Rightarrow y\}.(x + y)$$

a and b are two scope-free variables, but their creator, modifier and destructor are joint.

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

Proposition 4 *The scope-free variable encoding into the transformation calculus ensures locality to the modification sequence.*

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 [Garar]. The real point about them is that the use of a (now scoped) scope-free variable is not restricted by that syntactical scope[6] (which is only a problem of naming), like with the stack discipline, but by its *life area*, or modification sequence (its real operational scope).

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

6 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.*

$$\begin{array}{lcll} u & ::= & u_1 \mid \ldots & \textit{base types} \\ r & ::= & \{l \Rightarrow t, \ldots\} & \textit{stream types} \\ w & ::= & u \mid r & \textit{return types} \\ t & ::= & r \to w & \textit{types} \end{array}$$

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

These last restrictions make a stream type a stream of types as defined in Section 3. This means that we can use stream composition on these types, as we will do for typing rules.

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

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

with the same constraints on labels and variables as before.

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 3.*

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

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

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

Proposition 6 (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.

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

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

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

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

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

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

Figure 3: Typing rules for simply typed transformation calculus

7 Denotational semantics

In this section we give a model of the transformation calculus. However, to avoid technical problems specific to untyped models, and to get simple semantics, we base ourselves on the simply typed version of the calculus.

Definition 10 *A model of the transformation calculus is a pair* $(\mathcal{A}, [\![\_]\!]_\_)$ *with* $\mathcal{A}$ *a set of values, and* $(M, \rho) \mapsto [\![M]\!]_\rho : \Lambda_T \times \mathcal{A}^{\mathcal{V}} \to \mathcal{A}$ *a translation from a simply typed term* M *and an environment* ρ *(*$FV(M) \subset \mathcal{D}_\rho$*) into our model satisfying the axioms*

$$\begin{array}{rcl} M \equiv N & \Rightarrow & [\![M]\!]_\rho = [\![N]\!]_\rho, \\ M \to N & \Rightarrow & [\![M]\!]_\rho = [\![N]\!]_\rho, \\ \rho(x) = a & \Rightarrow & [\![x]\!]_\rho = a. \end{array}$$

We define our model $\mathcal{A} = \bigcup_{\tau \in T} \mathcal{A}^\tau$ by closure of the following procedure.

1. For $u \in T_0$ (base type), $\mathcal{A}^u$ is given. $\mathcal{A}_0 = \bigcup_{u \in T_0} \mathcal{A}^u$.

2. Streams values of level n are in $\mathcal{S}_n = \bigcup_{r \in \mathcal{S}(T_n)} \mathcal{A}^r$, where

$$\mathcal{A}^{\{l_i \Rightarrow \tau_i\}_{i=1}^m} = \bigcup_{a_1 \in \mathcal{A}^{\tau_1}} \cdots \bigcup_{a_m \in \mathcal{A}^{\tau_m}} \{l_1 \Rightarrow a_1, \ldots, l_m \Rightarrow a_m\}$$

3. Types of level $n+1$ are defined by

$$T_{n+1} = T_n \cup \{r \to w \mid r \in \mathcal{S}(T_n), w \in \mathcal{S}(T_n) \cup T_0\}$$

4. Values of level $n+1$ are defined by $\mathcal{A}_{n+1} = \bigcup_{\tau \in T_{n+1}} \mathcal{A}^\tau$ where

$$\mathcal{A}^{r \to w} = \mathcal{A}^r \to \mathcal{A}^w$$

5. $\mathcal{A} = \lim_{n \to \infty} \mathcal{A}_n$

$\mathcal{A}$ is well-defined, since for any τ there exists n such that $\tau \in T_n$ and then $\mathcal{A}^\tau \subset \mathcal{A}_n$.

Note that, like we did with types, we are identifying the streams of $\mathcal{A}^r$ with the functions of $\mathcal{A}^{\{\}\to r} = \mathcal{A}^{\{\}} \to \mathcal{A}^r$.

Once we have defined the values of the model, we must define operations on them. Out of concatenation, already defined on streams, we have two: *extension* and *composition*.

Extension is the operation by which a value in $\mathcal{A}^{r_1\to r_2}$ gets canonically extended into a value of $\mathcal{A}^{(r_1\cdot r)\to(r_2\cdot r)}$. If f is in $\mathcal{A}^{r_1\to r_2}$ then $f * r$, the r-extension of f is defined as:

$$\begin{array}{rrcl} f * r: & \mathcal{A}^{r_1\cdot r} & \to & \mathcal{A}^{r_2\cdot r} \\ \{l \Rightarrow x_l\}_{l\in\mathcal{D}_{r_1\cdot r}} & & \mapsto & (f\{l \Rightarrow x_l\}_{l\in\mathcal{D}_{r_1}}) \cdot (\{l \Rightarrow x_{\phi_{r_1}(l)}\}_{l\in\mathcal{D}_r}) \end{array}$$

Composition is just the mathematical one; if f is in $\mathcal{A}^{r_1\to r_2}$ and g in $\mathcal{A}^{(r_2\cdot r)\to w}$, then $f;g$ is defined as:

$$\begin{array}{rrcl} f;g: & \mathcal{A}^{r_1} & \to & \mathcal{A}^{w} \\ & s & \mapsto & g(f(s)) \end{array}$$

However the above definitions will not work as model: with the subtyping introduced on transformations in the calculus, we expect that the same value may actually be contained in several $\mathcal{A}^\tau$ sets. That is why we will consider the above definition modulo extension.

More precisely, we introduce the following equivalence $f =_* f * r$, closed by symmetry and transitivity (it is already reflexive). Since extension only applies on members of $\mathcal{A}^{r_1\to r_2}$, equivalent terms are transformations. Moreover we remark that, modulo this equivalence, $\mathcal{A}^{s_1\to s_2}$ now includes all $\mathcal{A}^{r_1\to r_2}$ such that for some r, $r_1 \cdot r = s_1$ and $r_2 \cdot r = s_2$.

We define $\mathcal{A}_*$ as $\mathcal{A}_{/=_*}$, and $\mathcal{A}^\tau_*$ as the sets of all classes containing an element of $\mathcal{A}^\tau$.

We show easily that composition is coherent with this equivalence: if $f \in \mathcal{A}^{r_1\to r_2}$ and $g \in \mathcal{A}^{r_2\to r_3}$ (for extension to be possible, both f and g must be transformations), then $f * r \in \mathcal{A}^{(r_1\cdot r)\to(r_2\cdot r)}$, $g * r \in \mathcal{A}^{(r_2\cdot r)\to(r_3\cdot r)}$, and

$$\begin{array}{l} (f * r; g * r)\{l \Rightarrow x_l\}_{l\in\mathcal{D}_{r_1\cdot r}} \\ = (g * r)((f\{l \Rightarrow x_l\}_{l\in\mathcal{D}_{r_1}}) \cdot (\{l \Rightarrow x_{\phi_{r_1}(l)}\}_{l\in\mathcal{D}_r})) \\ = (g(f\{l \Rightarrow x_l\}_{l\in\mathcal{D}_{r_1}})) \cdot (\{l \Rightarrow x_{\phi_{r_1}(l)}\}_{l\in\mathcal{D}_r})) \\ = ((f;g) * r)\{l \Rightarrow x_l\}_{l\in\mathcal{D}_{r_1\cdot r}} \end{array}$$

Finally we define the translation $(M, \rho) \mapsto [\![M]\!]_\rho : \Lambda_T \times \mathcal{A}_*^{\mathcal{V}} \to \mathcal{A}_*$, from a simply typed term M and an environment ρ ($FV(M) \subset \mathcal{D}_\rho$) into our model[7] in figure 4.

Proposition 7 *$(\mathcal{A}_*, [\![\_]\!]\_)$ is a model of the simply typed transformation calculus.*

8 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 [Cur93], in which composition and currying play a central role. The direction seems opposed: one encodes lambda-calculus into CCL (or its abstract machine version, the CAM [CCM87]), while

[7]Since terms are statically typed, we could construct a model based on tuples and type information, which would avoid the extra structure for streams, and compile away labels. Only extension would have to be modified. We based ourselves on streams here for the sake of simplicity of the equivalence relation.

$$
\begin{array}{lcl}
[\![x]\!]_\rho & = & \rho(x) \\
[\![\downarrow]\!]_\rho & = & \{\} \\
[\![\lambda\{l_0 \Rightarrow x : \tau\}.M]\!]_\rho & = & (\text{when } [\![M]\!]_{\rho[a^\tau/x]} \in \mathcal{A}^{r \to w}) \\
\quad \mathcal{A}^{\{l_0 \Rightarrow \tau\} \cdot r} & \to & w \\
\quad \{l \Rightarrow x_l\}_{l \in \{l_0\} \cup \mathcal{D}_r} & \mapsto & [\![M]\!]_{\rho[x_{l_0}/x]} \{l \Rightarrow x_l\}_{l \in \mathcal{D}_r} \\
[\![\{l \Rightarrow N\}.M]\!]_\rho & = & \{l \Rightarrow [\![N]\!]_\rho\}; [\![M]\!]_\rho \\
[\![M; N]\!]_\rho & = & [\![M]\!]_\rho; [\![N]\!]_\rho
\end{array}
$$

Figure 4: Semantic function of the simply typed transformation calculus

transformation calculus extends lambda-calculus. But the intuition that algorithmicity can be found in the structures of the lambda-calculus itself is the same.

The second one is process calculi. Their use of names for communication is similar to the principle of the transformation calculus. In [Bou89], Boudol proposes the γ-calculus. The base is lambda-calculus, but applications express emissions of messages and abstractions their reception, while multiple terms can be evaluated simultaneously. Milner's π-calculus [Mil92] proceeds alike, and by labeling with names applications and abstractions, it allows the use of multiple channels. The fundamental difference with our calculus is that non-determinism of the receptor of a message make these calculi divergent, while our terms are syntactically sequenced in order to keep determinism.

A third might be Lamping's Unified Parameterization System [Lam88], which tries like us to encode state modifications into the parameter passing system. However his systems departs essentially from lambda-calculus, so that this last has to be encoded; and destructive overriding makes impossible to limit the effect of modifying variables like we do.

After these somewhat different directions, our claims makes necessary to look at the larger 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.

This subject starts with Landin's encoding of Algol 60 into the lambda calculus [Lan65]. Or rather, nothing starts, since the problem remains unsolved: "The semantics of *applicative expressions* can be specified formally without the recourse to a machine. [...] With *imperative applicative expressions* on the other hand it appears impossible to avoid specifying semantics in terms of a machine".

Later, to encompass the stack discipline, marked store models were developped [Gor79, MS76] but they had two problems: a lack of abstraction, and the existence of some pathological cases, described in [MS88], where equivalences in Algol are not provable in the model.

A first answer to this was Oles and Reynolds category-theoretic models [Ole85, Rey81]. The essential idea is to define blocks as functions that can be applied to a range of states with various shapes, but do not change their shapes. However, inside the block, state is temporarily extended with local variables. Thus, they do not appear in its meaning. Our approach shares a lot with this view, since we syntactically "expand" and "shrink" our state when we create and delete a scope-free variable.

Another one is the Halpern-Meyer-Trakhtenbrot Store Model [HMT84, THM84], later refined by Meyer and Sieber [MS88], still based on stores, but using locally complete partial orders. It comes at last very close to full-abstraction, but fails in a 7^{th} example. Aknowledging the depth of the problem, Mason and Talcott even proposed an Operational Framework [MT92b, MT92a] to solve it out of denotational semantics.

Apart from this problem, there is interesting remark about the *Orthogonality of assignments and procedures in Algol* [WF93]. A theorem is enunciated, proving that normalization of an Algol program can be done in two phase, one using β and copy rules, and the other a simple stack machine. We do not obtain such a result, since we do not explicitly distinguish between imperative and functional features in the transformation calculus, but we can see the same kind of behaviour, first reducing higher order functions and eliminating composition, and then reducing β-redexes of ground types.

If we go out of the Algol tradition, we can forget about the stack discipline. As a result, most systems give a formalization of references. So does the λ_v-S-calculus [FF87, FF89] for Scheme, and a call-by-value reduction strategy. With effect inference [GL86, LG88, TJ92], restrictions on the reduction strategy can be reduced, and, for instance, parallelism can be introduced.

Still, we feel more concerned by systems going the other way, starting without a specific reduction strategy. There are a number of them, which enforce single-threadedness of variables by various typing disciplines [GH90, PW93, SRI91, Wad90a, Wad90b]. We can see an intuitive relation between the way scope-free variables are used and linear types, but still we are not relying to typing for single-threadedness.

We actually do it in a syntactical way. In that we are very close to λ_{var} [ORH93, CO94]. In fact, even the structures of the calculus have similarities: like us, they use the linear structure of spines to ensure single-threadedness. They have rules to propagate the values of mutable variables along the spine of a term, like does our structural equivalences for labeled arguments, and their **return**-elimination rule $((\mathbf{return}\ N) \triangleright \lambda x.M \to (\lambda x.M)N$, *cf.* [ORH93]) can be seen as a variant of $\downarrow$-elimination ($\downarrow; M \to M$) including value-passing. The essential difference is that, since we use the same mechanism for scope-free variables and value-passing, we obtain a more unified calculus. In particular, the fact they are encoding references means that they must do some kind of garbage collection (their **pure** construct) to convert a value obtained using mutable variables into a purely functional one. In the transformation calculus, since we explicitely delete variables, we do not need such an "impure" purifier.

9 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 [Gar95] a polymorphic version of it, extending that for selective λ-calculus [GAK94]. This could be completed by the introduction of linear types [Wad90b]: 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 fictious 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. As we remarked about the model, there is no problem if we use non-polymorphic typing, 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 would give unique identifiers for scope-free variable, and avoid the hiding of a label in those sub-sequences which create new variables on this label. A more structured label space could even enable 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.

A Applicative translation and confluence

The idea of applicative continuation semantics was developed in [JD88] for a framework of stores and partial continuations. A partial continuation is a function from a state to a new state, but we can translate it into a function prefixing a continuation, that is a function from a continuation to a new continuation doing intended operations before calling the old continuation. In fact transformations look very much like partial continuations on streams, and this idea provides us with a translation from transformation calculus to selective λ-calculus.

Definition 11 *Tr is the applicative translation from $\Lambda_T(\mathcal{L}_s)$ (without the associativity of composition $(M_1; M_2); M_3 \equiv M_1; (M_2; M_3)$) to $\Lambda_S(\mathcal{L}_s \cup \{cont\})$.*

$$\begin{array}{rcl} Tr(\downarrow) & = & \lambda\{cont \Rightarrow x\}.x \\ Tr(M;N) & = & \{cont \Rightarrow Tr(N)\}.M \\ Tr(x) & = & x \\ Tr(\lambda R.M) & = & \lambda R.Tr(M) \\ Tr(R.M) & = & Tr(R).Tr(M) \end{array}$$

The well-definedness of this translation is proved in Lemma 1.
The image of Λ_T is $Tr(\Lambda_T) = \Lambda_S^*$.

"*cont*" in the above definition stands for continuation. We translate composition into an application of its right-hand side to its left-hand side, seen as a continuation. Since *cont* is a new name, the continuation is received by the $\lambda\{cont \Rightarrow x\}.x$ head of the right-hand side, and operational semantics of the calculus are preserved.

We will use this translation to prove the confluence of transformation calculus, based on that of selective λ-calculus.

Theorem 1 *Selective λ-calculus is confluent.*

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

PROOF As one may expect, the proof is complex, and can be found for a variant of this calculus in [AKG93]. We have to modify two points. First, in its original definition selective λ-calculus use sum labels ($\mathcal{L} = \mathcal{N} \cup \mathcal{L}_s$) whereas we use a product system here ($\mathcal{L} = \mathcal{L}_s \times \mathcal{N}$). Names and indexes being independent, the modification of the proof is immediate. The modified proof appears in [AKGar]. Next, structural equivalences were defined in terms of reordering rules included in the calculus. Once proved the confluence of the calculus including them, we can transform these rules into equivalences and keep confluence. □

Lemma 1 *Tr is coherent w.r.t. $\equiv$ (without associativity of composition), and reduction paths coincide. Tr is a bijection from Λ_T to Λ_S^*.*

PROOF For coherence, we just verify that all equivalences in Λ_T are mapped to equivalences in Λ_S. For $\equiv_.$, $\equiv_\lambda$ and $\equiv_{.\lambda}$ this is immediate, since the same equivalence applies in the translation. $\equiv_{.;}$ and $\equiv_{\lambda;}$ are mapped respectively to $\equiv_.$ and $\equiv_{.\lambda}$:

$$\begin{aligned} Tr((R.M_1); M_2) &= \{cont \Rightarrow Tr(M_2)\}.Tr(R).Tr(M_1) \\ &\equiv Tr(R).\{cont \Rightarrow Tr(M_2)\}.Tr(M_1) \\ &= Tr(R.(M_1; M_2)) \end{aligned}$$

$$\begin{aligned} Tr((\lambda R.M_1); M_2) &= \{cont \Rightarrow Tr(M_2)\}.\lambda R.Tr(M_1) \\ &\equiv \lambda R.\{cont \Rightarrow Tr(M_2)\}.Tr(M_1) \\ &= Tr(\lambda R.(M_1; M_2)) \end{aligned}$$

For reduction steps, both $\rightarrow_\beta$ and $\rightarrow_\downarrow$ are mapped to β-reduction:

$$Tr(\downarrow; M) = \{cont \Rightarrow Tr(M)\}.\lambda\{cont \Rightarrow x\}.x$$

We define Tr^{-1} by reversing each case in the definition of T_r. It is well-defined on raw terms of Λ_S^*, and coherent by reversing cases above. This makes Tr a bijection. □

The above lemma lets us translate reduction of the transformation calculus into selective λ-calculus ones. We need another lemma to get reductions starting in Λ_S^* back into Λ_T.

Lemma 2 *If $M \in \Lambda_S^*$ and $M \rightarrow N$ then $N \in \Lambda_S^*$.*

PROOF Terms of Λ_S^* are characterized by the fact all abstractions using *cont* only appear in the form $\lambda\{cont \Rightarrow x\}.x$.

If the reduction step is not on *cont*1, then it do not create any abstraction on *cont*, nor modify $\lambda\{cont \Rightarrow x\}.x$'s.

If this is on *cont*1, then the $\lambda\{cont \Rightarrow x\}.x$ concerned disappears.

In the two cases, the resulting term is still in Λ_S^*. □

Theorem 2 *Transformation calculus is confluent.*

$$\begin{aligned} \forall M, P, Q \in \Lambda_T\ & (M \xrightarrow{*} P \wedge M \xrightarrow{*} Q) \\ & \Rightarrow (\exists T \in \Lambda_T\ P \xrightarrow{*} T \wedge Q \xrightarrow{*} T) \end{aligned}$$

PROOF By Lemma 1, we get $Tr(M) \xrightarrow{*} Tr(P)$ and $Tr(M) \xrightarrow{*} Tr(Q)$. By confluence of selective λ-calculus, we have T' such that $Tr(P) \xrightarrow{*} T'$ and $Tr(Q) \xrightarrow{*} T'$. But by Lemma 2 these reductions are in Λ_S^*, so that $T = Tr^{-1}(T')$ is a solution. □

B Proofs of propositions

Proposition 1 (reversibility) *ϕ_r is a bijection from $\mathcal{S}$ to $\{s \in \mathcal{S} \mid \mathcal{D}_r \cap \mathcal{D}_s = \emptyset\}$. $\psi_r \circ \phi_r = id_{\mathcal{S}}$.*

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

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

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

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

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

Proposition 2 (monoid) *Concatenation as in definition 2 is an associative application $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$, accepting $\{\}$ as neutral element.*

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

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

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

$r \cdot \{\} = r = \{\} \cdot r$ is immediate ($\phi_{\{\}} = id$). □

Proposition 4 *The scope-free variable encoding into the transformation calculus ensures locality to the modification sequence.*

PROOF If our variable is encoded on a label whose name is unique, there is no problem since it does not modify indexes of other arguments and variables, and its label appears only between its creation and destruction.

Otherwise, labels with the same name may appear with a different index in the sequence. But since each time they meet an abstraction or an application on our variable their indexes

go alternately up and down, and the number of abstraction is equal to the number of applications, their index is the same on each side of the sequence. Moreover modifiers do not modify the value they support, since index changes avoid overloading of the same label. □

Proposition 5 (strong normalization) *If $\Gamma \vdash M : \tau$ in the simply typed transformation calculus, then there is no infinite reduction sequence starting from M.*

PROOF We base ourselves on a translation into a simply typed λ-calculus with streams (we use matching rather than projections). This calculus, clearly equivalent to simply typed λ-calculus with pairing, is strongly normalizing.

We translate a proof Π of $\Gamma \vdash M : \tau$ into a proof Π' of $\Gamma \vdash M' : \tau$ in simply typed λ-calculus with streams, where types are identical to those of the simply typed transformation calculus. This translation is very similar to that used for our simply typed model.

Rules get translated into (I does not change):

$$\frac{\Gamma[x \mapsto \theta] \vdash M' : r \to w}{\begin{array}{l}\Gamma \vdash \lambda^*\{k \Rightarrow y_k\}_{k \in \{l\}\cdot\mathcal{D}_r} : \{l \Rightarrow \theta\} \cdot r . (\lambda x : \theta . \\ \quad M' \ \{k \Rightarrow y_{\phi_{\{l\}}(k)}\}_{k \in \mathcal{D}_r}) \ y_l : \{l \Rightarrow \theta\} \cdot r \to w\end{array}} \quad \text{(II')}$$

$$\frac{\Gamma \vdash M' : \{l \Rightarrow \theta\} \cdot r \to w \quad \Gamma \vdash N' : \theta}{\Gamma \vdash \lambda^* y : r . M(\{l \Rightarrow N\} \cdot y) : r \to w} \quad \text{(III')}$$

$$\Gamma \vdash \lambda y : \{\} . y : \{\} \to \{\} \quad \text{(IV')}$$

$$\frac{\Gamma \vdash M' : r_1 \to r_2 \quad \Gamma \vdash N : r_2 \to w}{\Gamma \vdash \lambda^* y : r_1 . N'(M' \ y) : r_1 \to w} \quad \text{(V')}$$

$$\frac{\Gamma \vdash M' : r_1 \to r_2}{\begin{array}{l}\Gamma \vdash \lambda^*\{l \Rightarrow y_l\}_{l \in \mathcal{D}_{r_1 \cdot r}} : r_1 \cdot r . ((M'\{l \Rightarrow y_l\}_{l \in \mathcal{D}_{r_1}}) \\ \qquad \cdot \{l \Rightarrow y_{\phi_{r_1}(l)}\}_{l \in \mathcal{D}_r}) : r_1 \cdot r \to r_2 \cdot r\end{array}} \quad \text{(VI')}$$

We sketch the rest of the proof.

We have marked some abstractions with * in this translation. This means that they have only a structural role. We will just ignore them, and write $\overline{M'}$ for M' where all $*$-marked redexes were reduced (thanks to strong normalization). All other redexes are kept, since $*$-marked abstractions are linear, and their reduction do not change inclusion relation between redexes. This gives us unicity of $\overline{M'}$. Moreover, in $\overline{M'}$ all potential redexes of M (using $\equiv$) appear. Particularly, associativity of transformation composition maps to associativity of λ-function composition.

One can verify that if we have $\Gamma \vdash M \to N : \tau$ in the simply typed transformation calculus, then, for M' and N' translations of M and N (the proof of $\Gamma \vdash N : \tau$ chosen to be similar to that of $\Gamma \vdash M : \tau$), we have $\Gamma \vdash \overline{M'} \to^* \overline{N'} : \tau$ in the simply typed λ-calculus with streams (we may need more steps than in the orginal, because of marked redexes).

As a result, strong normalization extends to simply typed transformation calculus. □

Proposition 6 *$(\mathcal{A}_*, [\![\_]\!]\_)$ is a model of the simply typed transformation calculus.*

PROOF We must prove that our three axioms are verified.

- $\equiv$: the proof is direct for each basic equivalence.

- $\rightarrow_{\downarrow}$: just notice that all extensions of $[\![\downarrow]\!]$ are the identity.
- $\rightarrow_{\beta}$: $[\![\{l \Rightarrow N\}.\lambda\{l \Rightarrow x\}.M]\!]_{\rho} = [\![M]\!]_{\rho[[\![N]\!]_{\rho}/x]}$
  $= [\![[N/x]M]\!]_{\rho}$.

□

References

[AKG93] Hassan Aït-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.

[AKGar] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. *Theoretical Computer Science*, to appear.

[BCL90] Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. Technical Report LIENS-90-14, LIENS, July 1990.

[Bou89] Gérard 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.

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, 1987.

[CO94] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignments. In *Proc. of the International Conference on Theoretical Aspects of Computer Software*, pages 347–363, 1994.

[Cur93] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1993. First edition by Pitman, 1986.

[FF87] Mathias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 314–325, 1987.

[FF89] M. Felleisen and D.P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.

[GAK94] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994.

[Gar95] Jacques Garrigue. *Label-Selective Lambda-Calculi and Transformation Calculi*. PhD thesis, University of Tokyo, Department of Information Science, March 1995.

[Garar] 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.

[GH90] J.C. Guzmán and P. Hudak. Single threaded polymorphic calculus. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

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

[Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[HMT84] J.Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Proc. ACM Symposium on Principles of Programming Languages*, pages 245–257, 1984.

[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*, pages 158–168, 1988.

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

[Lan65] 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.

[LG88] 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.

[Mil92] Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, pages 203–246. NATO ASI Series, Springer Verlag, 1992.

[MS76] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[MS88] Albert R. Meyer and Kurt Sieber. Toward fully abstract semantics for local variables. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.

[MT92a] Ian A. Mason and Carolyn L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2), 1992.

[MT92b] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 186–197, 1992.

[Ole85] F.J. Oles. Type algebras, functor categories, and block strcutures. In N. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 15, pages 543–573. Cambridge University Press, 1985.

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

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[Rey81] 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.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hugues, editor, *Proc. ACM Symposium on Functional Programming and Computer Architectures*, pages 192–214. Springer Verlag, 1991. LNCS 523.

[THM84] B. A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. From denotational to operational and axiomatic semantics for ALGOL-like languages: An overview. In E. CLarke and D. Kozen, editors, *Logic of Programs*, pages 474–500, Berlin, 1984. LNCS 164, Springer-Verlag.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[Wad90a] Philip Wadler. Comprehending monads. In *Proc. ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.

[Wad90b] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

[WF93] S. Weeks and M. Felleisen. On the orthogonality of assignments and procedures in Algol. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 61–78, January 1993.