# DYNAMIC BINDING AND LEXICAL BINDING IN A TRANSFORMATION CALCULUS

(preliminary version)

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

ABSTRACT

While dynamic and lexical binding for variables have both their uses in programming, no language gives an equal access to them.
By encoding them simultaneously in an extension of the $\lambda$-calculus, we show that there is no contradiction between them, and give a better understanding of their respective roles.
Moreover we give a simple type system for the encoding calculus, showing that not only lexical binding, but also dynamic binding, can be correctly typed without abusive restrictions.

## 1. Introduction

When local variables appear in a program, one may think of two strategies to define the binding of names to values. With dynamic binding, it is *dynamically* changed; that is, during the execution of the program, while the local variable exists, all uses of its name are interpreted as references to it, even in subroutines out of the syntactic scope of this local definition.

On the contrary, with lexical binding, only the *syntactic structure* of the program is considered. A variable name is interpreted according to the syntactically closest variable definition in the program, even if the execution has gone through other variable definitions before accessing this name.

It has long been observed that, if we think of a purely functional use of variables, in which variables can be assigned a value only once, lexical binding is to prefer, since it is the only one to preserve referential transparency. However, if we think of mutable variables, this argument disappears, since anyway the value stored in the variable may change over time. And there are cases where dynamic binding offers advantages, for instance when one wants to modify non-destructively the binding context.

The goal of this paper is, by defining a referentially transparent calculus, close to the lambda-calculus, in which we can encode the two binding strategies, to show how they differ but are not incompatible.

Moreover, by giving a simple type system for this calculus, we show that dynamic binding can be typed without restricting it more than lexical binding. The fact this type system ensures strong normalization is also interesting.

Section 3 presents our *transformation calculus*. In Section 4 it is shown how we can encode dynamic binding into it. Section 5 introduces local labels, and encodes lexical binding. Then Section 6 gives a type system for this last calculus, and Section 7 concludes. No proofs are given in this paper, but please refer to other reports [2,3] for more details.

## 2. Previous encodings

One could object that such calculi, close to the lambda-calculus, and in which we can encode mutable variables, already exist.

For instance, dynamic binding is easily encoded in the untyped lambda-calculus (with symbols or strings). We just have to add to each function an argument, the store, represented as a function from symbols to values, and recover it from the output. This can easily be extended to lexical binding by using some form of name binding, like in the $\lambda\nu$-calculus [7]. However, there is a default to this encoding: our store will be always growing, since there is no way to discard a binding. As a result such an encoding does not provide a proper model for mutable variables.

To overcome this discarding problem, one may represent the store as a record. For this we need several operations: extension, to introduce a new variable; modification, to change the value of a variable; and restriction, to suppress a variable once it gets useless[a]. This, combined with local labels, works well with lexical binding. However, dynamic binding becomes heavy: when creating a local variable one has to explicitly save the previous binding of the label, and to restore it afterwards, whereas intuitively the old binding is only hidden. Moreover, in both cases we have to manipulate explicitly the store, which means a low level of integration between the original calculus and mutable variables.

A slight simplification to these manipulations is obtained by translating programs (expressed as $\lambda$-terms) into a *continuation passing style* [10,5]. Since we do not come back from functions, there is no longer any need to get the store in return. The transformation itself is automatic, and we can chose it to use either call-by-value or call-by-name semantics. But this last possibility shifts the problem. In fact we do not need all the framework of CPS just to talk about binding strategies, and do not want to chose *a priori* between call-by-value and call-by-need for the whole calculus.

There also exist various systems dedicated to one of the binding strategies [1,8]. Comparison between these systems may be another way to tackle the problem, but we think that the framework of a unique calculus is instructive.

## 3. Transformation calculus

This is the calculus binding strategies will be encoded in. The definition is done in

[a]This last operation is often absent, being delegated to garbage collection. This makes things look simpler, but introduces some opaque part in the system.

four steps. 1) We present a simplified version of the transformation calculus, without labels. 2) We define streams, a data structure on which we base our calculus. 3) We give a syntactic definition of terms in the transformation calculus, and add a structural equivalence on these terms. 4) Then we define reduction rules for these equivalence classes.

### *3.1. Simple transformation calculus*

The initial idea of the transformation calculus is to add to the lambda-calculus a capacity to pipe the output of one term into another. This is stronger than composition, because we want to be able to return not only values, but also sequences of values.

The calculus includes two new constructs:

$$M ::= x \mid \lambda x.M \mid M.M \mid \downarrow \mid M; M$$

First remark that, to simplify notations, we write application in a reversed way: in $N.M$, $M$ is applied to $N$. As usual $N_1.N_2.M$ is $N_1.(N_2.M)$.

The two new constructs are used to introduce and destroy sequences of values, the only associated rule being (weak) $\downarrow$-elimination:

$$N_1.\cdots.N_n.\downarrow; M \to N_1.\cdots.N_n.M$$

To that, we of course add $\beta$-reduction:

$$N.\lambda x.M \to [N/x]M$$

With this simple transformation calculus, we can easily simulate a stack machine. For instance switching two elements at the top of the stack is written $C = \lambda x.\lambda y.y.x.\downarrow$, and reduces as follows.

$$c.b.a.\downarrow; C \to c.b.a.\lambda x.\lambda y.x.y.\downarrow \to c.a.b.\downarrow$$

This is enough to encode imperative algorithms, but not named variables. That is why we introduce labeled streams.

### *3.2. Streams*

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 $(\mathcal{S}, \cdot)$.

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

Figure 1: Syntax of the transformation calculus

*Preliminaries*

- $\mathcal{L}_s$ is a 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 $\mathcal{A}$.*

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

- *$\mathcal{D}_s$ 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 $\epsilon n$, $p$ denotes $p1$*
  - *if $l = pn$ then $l + m = p(n + m)$*
  - *$(a_1, \ldots, a_n) = \{1 \Rightarrow a_1, \ldots, n \Rightarrow a_n\}$*

For the details of operations on streams, see Appendix A.

### *3.3. Syntax*

*Notations.* In the following definitions we will use the abbreviations $A \not\frown B$ 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 $\Lambda_T$, 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 congruence defined by the closure of the equalities in figure 2.*

$$
\begin{array}{rll}
S.R.M & \equiv_{\cdot} & (R \cdot S).M \\
\lambda R.\lambda S.M & \equiv_{\lambda} & \lambda(S \cdot R).M \qquad V(R) \between V(S) \\
R.\lambda S.M & \equiv_{\cdot\lambda} & \lambda\psi_R(S).\psi_S(R).M \quad FV(R) \between V(S), \mathcal{D}_R \between \mathcal{D}_S \\
 & & \\
(R.M_1); M_2 & \equiv_{;} & R.(M_1; M_2) \\
(\lambda R.M_1); M_2 & \equiv_{\lambda;} & \lambda R.(M_1; M_2) \qquad V(R) \between FV(M_2) \\
(M_1; M_2); M_3 & \equiv_{;} & M_1; (M_2; M_3)
\end{array}
$$

Figure 2: Structural equivalences

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

Equality $\equiv_{\cdot\lambda}$ is the "symmetrical" of $\beta$-reduction. It comes from the need to close the equality [b]

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

with $\mathcal{D}_{R'} = \mathcal{D}_R, \mathcal{D}_{S'} = \mathcal{D}_S$ and $V(S) \between V(R)$. If we take $M = \lambda\psi_S(R).N$, and apply $R'.\lambda S.M$ to $\psi_R(S')$, then $\equiv_{\cdot\lambda}$ preserves confluence: it gives

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

*3.4. Reductions*

Substitutions are done in the same way as for lambda-calculus, composition not interacting with variable binding. Terms will always be considered modulo $\alpha$-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 $\beta$-reduction and $\downarrow$-elimination[c].*

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

**Theorem 1** *The transformation calculus is confluent.*

[b] $\psi$ is the reverse shifting function, such that $R \uplus S = R \cdot \psi_R(S)$, where $\uplus$ is the union of disjoint partial functions.

[c] Remark that, thanks to the structural rules, the new $\downarrow$-elimination is stronger the first one we introduced (it even works when $\downarrow$ is prefixed by abstractions). 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.

## 4. Encoding dynamically bound variables

We call this kind of variable *scope free*. By this we mean that the extent of their use is not determined by a syntactic scope, but simply by their presence as argument in a sequence of transformations.

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:

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

The fundamental property of scope free variables is that, like (lexically bound) 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[d]. 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 dynamic scoping can be more accurate than a syntactic one.

[d] As a result, mutable variables behave like in call-by-name. But this does not mean that we make any hypothesis on the evaluation strategy used for the transformation calculus. The calculus itself being confluent, any strategy will do.

Here is the first 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 $5 + 10 - (3 + 10) = 2$. It goes through the following steps, and gives the expected result.

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

Note here that since we encode dynamic binding, we would get the same result even if the central part was defined as a subprogram: with scope-free variable, even Basic's subprograms would behave correctly, since we can create a scope free variable before the call to pass a parameter, and destroy it after.

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.

Imagine 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 $A$.

The scope free answer is:

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

We didn't define any new variable, but did just temporarily hide the original value by the redirected one. With a syntactic scope, one of the two scopes (that of *con* or *scr*)

$$
\begin{array}{rcll}
\nu p.M & \equiv & M & p \notin FN(M) \\
\nu p.M & \equiv & \nu q.[q/p]M & q \notin FN(M) \\
\nu p.\nu q.M & \equiv & \nu q.\nu p.M & \\
 & & & \\
\lambda\{pm \Rightarrow x\}.\nu q.M & \equiv & \nu q.\lambda\{pm \Rightarrow x\}.M & p \neq q \\
\{pm \Rightarrow N\}.\nu q.M & \equiv & \nu q.\{pm \Rightarrow N\}.M & p \neq q, q \notin FN(N) \\
M; \nu p.N & \equiv & \nu p.(M; N) & p \notin FN(M) \\
\nu p.M; N & \equiv & \nu p.(M; N) & p \notin FN(N)
\end{array}
$$

Figure 3: Structural equivalences for label scoping

would necessarily have included all $A; B; C$, whereas here they are both restricted to their area of concern.

## 5. Local labels, for lexical binding

To encode lexical binding, or *scoped* variables, we need the new concept of *local label*. Once we have it, a scope free variable defined on a local label becomes very naturally a scoped variable.

**Definition 4** *Terms of the* scoped transformation calculus *are those of* $\Lambda_T$ *extended with a label-scoping construct,*

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

*where* $p$ *is a name in* $\mathcal{L}_s$.

$FN(M)$*, the set of free names in* $M$*, is defined in the usual way,* $\nu p.N$ *hiding occurrences of* $p$ *in* $N$.

*The structural equivalences of Figure 2 still hold, but we add to them those of Figure 3.*

We have to extend variable substitution with

$$[N/x](\nu p.M) = \nu p.[N/x]M \quad p \notin FN(N)$$

but reduction rules themselves are not modified.

**Theorem 2** *The scoped transformation calculus is Church-Rosser.*

The key point in the new structural rules is that we don't equate $\{pm \Rightarrow \nu q.N\}.M$ and $\nu q.\{pm \Rightarrow N\}.M$. This would, in this form, make the calculus incoherent, scopes depending of the form on which we apply $\beta$-reduction —if the $\nu q$ is out all residuates of $N$ are in the same scope, otherwise they are in different ones—, and even if we restrict $\beta$ reductions to terms containing no $\nu$, this would lose the name generating power: once all $\nu$ are pushed out we can just forget about them for all internal reductions.

By keeping the $\nu$ in the applications, we create different scopes for each residual of $N$. This amounts to creating news names for each of them, and gives us lexical binding.

For instance, the equivalent of a reference cell can be defined as follows

$$\text{cell} = \nu r. \left\{ \begin{array}{l} mk \Rightarrow \lambda\{x\}.\{r \Rightarrow x\}.\downarrow \\ get \Rightarrow \lambda\{r \Rightarrow x\}.\{x, r \Rightarrow x\}.\downarrow \\ set \Rightarrow \lambda\{x, r \Rightarrow y\}.\{r \Rightarrow x\}.\downarrow \\ del \Rightarrow \lambda\{r \Rightarrow x\}.\downarrow \end{array} \right\}.\downarrow$$

When you want to use it you need to first unpack it (actually create the new name), then to compose the *mk* transformation, some transformations using the cell, and the *del* transformation. For instance,

$$\begin{array}{l} \text{cell}; \lambda\{mk \Rightarrow mk, get \Rightarrow get, del \Rightarrow del\}. \\ \qquad (\{1\}.mk; \{r \Rightarrow 10\}.\downarrow; get; \lambda\{x\}.\{x+1\}.set; get; del) \end{array}$$

Here unpacking is done by composing the definition of the cell with a function that takes its transformations, and uses them locally.

This transformation evaluates to $\{2, r \Rightarrow 10\}.\downarrow$, because of the scoping (otherwise it would have been $\{11, r \Rightarrow 1\}$).

Remark however that this does not mean that the transformation calculus suddenly changed from dynamic to lexical binding. That only means that they are different problems: scope free variables are about where a variable is available in a program, whereas name scopes are about how one can access it (or not).

## 6. Simple type system

To obtain a simply typed form of the scoped 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. Moreover we introduce name abstraction in types[e].

**Definition 5 (simple type)** *Simple types in the transformation calculus are generated by t in the grammar of Figure 4, with a structural congruence[f], and a (reflexive) subtyping relation defined modulo this congruence.*

Note that a stream type is a stream of types, as defined in Appendix A. This means that we can use stream composition on these types.

[e] We note it equivalently $\nu\vec{p}.t$ or $\nu p_1 \dots \nu p_n.t$ if $\vec{p} = (p_1, \dots, p_n)$.

[f] Because of this congruence, the equality on types is not trivial, but we can test it by 1) taking a normal form, where all abstracted labels appear at least once in their scopes; 2) matching combinatorially the abstracted labels for every type.

*Types*

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

*Congruence*

$$\begin{array}{lcll} \nu p.\tau & = & \tau & p \notin FN(\tau) \\ \nu q.\tau & = & \nu p.[p/q]\tau & p \notin FN(\tau) \\ \nu p.\nu q.\tau & = & \nu q.\nu p.\tau & \end{array}$$

*Subtyping*

$$\frac{r_1' \prec r_1 \qquad r_2 \prec r_2'}{\nu\vec{p}.(r_1 \to r_2) \prec \nu\vec{p}.(r_1' \cdot r \to r_2' \cdot r)}$$

$$\frac{r' \prec r}{\nu\vec{p}.(r \to u) \prec \nu\vec{p}.(r' \to u)}$$

$$\frac{\tau_1 \prec \tau_1' \ldots \tau_n \prec \tau_n'}{\{l_1 \Rightarrow \tau_1, \ldots, l_n \Rightarrow \tau_n\} \prec \{l_1 \Rightarrow \tau_1', \ldots, l_n \Rightarrow \tau_n'\}}$$

Figure 4: Simple types

**Definition 6 (simply typed term)** *A term in the simply typed scoped transformation calculus is constructed according to the following syntax.*

$$\begin{array}{rcl} M & ::= & x \mid \downarrow \mid \lambda\{l \Rightarrow x{:}t, \ldots\}.M \mid \{l \Rightarrow M, \ldots\}.M \mid M; M \mid \nu p.M \\ & \mid & \mathit{let}\ \{l \Rightarrow x, \ldots\} = M\ \mathit{in}\ M \end{array}$$

*with the same constraints on labels and variables as before.*

The new *let* construct is present here to permit the propagation of bound labels. In an untyped context, you can see it as $\mathit{let}\ \{l \Rightarrow x, \ldots\} = M \mathit{in}\ N \simeq M; \lambda\{l \Rightarrow x, \ldots\}.N$, and as such we add the rules

$$\begin{array}{lcll} \text{let } P = \nu p.M \text{ in } N & \equiv & \nu p.\text{let } P = M \text{ in } N & p \notin FN(P.N) \\ \text{let } P = M \text{ in } \nu p.N & \equiv & \nu p.\text{let } P = M \text{ in } N & p \notin FN(P.M) \\ \text{let } \{\} = M \text{ in } N & \to & M; N & \\ \text{let } \{l \Rightarrow x\} \cdot P = Q.\{l \Rightarrow M\}.\downarrow \text{ in } N & \to & \text{let } P = Q.\downarrow \text{ in } [M/x]N & \end{array}$$

We must slightly modify scoping/abstraction structural equivalence to cope with the presence of (possibly scoped) types in abstractions.

$$\lambda\{pm \Rightarrow x{:}\tau\}.\nu q.M \equiv \nu q.\lambda\{pm \Rightarrow x{:}\tau\}.M \quad p \neq q, q \notin FN(\tau)$$

**Definition 7 (type judgement)** *A* type judgement*, written $\Gamma \vdash M : \tau$, expresses that the term $M$ has type $\tau$ in the context $\Gamma$. Induction rules for type judgements are given in figure 5.*

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

$$\frac{\Gamma[x \mapsto \theta] \vdash M : \nu\vec{p}.(r \to w)}{\Gamma \vdash \lambda\{l \Rightarrow x{:}\theta\}.M : \nu\vec{p}.(\{l \Rightarrow \theta\} \cdot r \to w)}\ \vec{p} \not\sharp FN\{l \Rightarrow \theta\} \tag{II}$$

$$\frac{\Gamma \vdash M : \nu\vec{p}.(\{l \Rightarrow \theta\} \cdot r \to w) \quad \Gamma \vdash N \prec \theta}{\Gamma \vdash \{l \Rightarrow N\}.M : \nu\vec{p}.(r \to w)}\ \vec{p} \not\sharp FN\{l \Rightarrow \theta\} \tag{III}$$

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

$$\frac{\Gamma \vdash M \prec \nu\vec{p}.(r_1 \to r_2) \quad \Gamma \vdash N \prec \nu\vec{q}.(r_2 \to w)}{\Gamma \vdash M; N : \nu\vec{p}.\nu\vec{q}.(r_1 \to w)} \tag{V}$$

where $\vec{p} \not\sharp FN(r_2 \to w), \vec{q} \not\sharp FN(r_1 \to r_2)$, $(r_1 \to w)$ minimal.

$$\frac{\Gamma \vdash M : \nu\vec{p}.\{l_i \Rightarrow \theta_i\}_1^n \cdot r_1 \quad \Gamma[x_i \mapsto \theta_i]_1^n \vdash N \prec \nu\vec{q}.(r_1 \cdot r_2 \to w)}{\Gamma \vdash \text{let } \{l_i \Rightarrow x_i\}_1^n = M \text{ in } N : \nu\vec{p}.\nu\vec{q}.(r_2 \to w)} \tag{VI}$$

where $\vec{p} \not\sharp FN(N, \Gamma, \vec{l_i}), \vec{q} \not\sharp FN(r_1)$, $(r_2 \to w)$ minimal.

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \nu p.M : \nu p.\tau}\ p \notin FN(\Gamma) \tag{VII}$$

Figure 5: Typing rules for simply typed scoped transformation calculus

Rules (I,II,III) are the traditional ones for typed lambda calculus, simply extended to streams and subtyping, and preserving the scoping of names. 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 $l = \epsilon 1$), and suppressing abstractions on names.

Rule (IV) types the constant $\downarrow$. Note that any $r \to r$ is a subtype of $\{\} \to \{\}$.

Rule (V) types composition: $N$ is applied to the result stream of $M$, and re-abstracted by its abstraction part. The use of the subtyping rule permits to get a common $r_2$ for $M$ and $N$. The minimality constraint on $r_1 \to w$ ensures the uniqueness of typing.

Rule (VI) types name propagation. It is needed, since we cannot make explicit the private labels of $M$ in $N$'s type with the previous rule. Fresh names are first created for these, which are directly introduced in $N$'s environment. Again minimality ensures uniqueness.

Finally rule (VII) types name abstraction.

**Proposition 1 (subject reduction)** *Types decrease monotonously: if* $\Gamma \vdash M : \tau$, *and* $M \to N$ *or* $M \equiv N$, *then* $\Gamma \vdash N \prec \tau$.

**Proposition 2 (strong normalization)** *If* $\Gamma \vdash M : \tau$ *in the simply typed scoped transformation calculus, then there is no infinite reduction sequence starting from* $M$.

This last proposition lets us see that we didn't introduce anything "impure" in the lambda-calculus. We can manipulate both dynamic and lexical variables, but this does not introduce any possibility of non-termination in the calculus, since they are independent from syntactic variables, and explicitly appear in types.

## 7. Conclusion

We presented here a calculus powerful enough to directly encode both dynamically and lexically bound (mutable) variables.

A first remark is that the encoding of scoped (lexically bound) variables is done in addition to the encoding of scope free (dynamically bound) variables. In our calculus the natural binding strategy is dynamic, and lexical binding is obtained by scoping the name of such a dynamic variable.

In some way this was to be expected, since the modeling of dynamic binding by stores has been known for long, but a satisfactory model of lexical binding cannot be obtained in a straightforward way [9,11,4,12,6].

A second remark is that the notions behind these two binding strategy are orthogonal. Dynamic binding is about the presence of a variable in the environment, and lexical binding is about its accessibility through a name. One could argue that creating a dynamic variable potentially hides another variable with the same name as a side effect — and as such changes its accessibility —, but in the transformation calculus we can still use indexes to retrieve it.

This is an indication that these two aspects should be considered independently. Usually the first one is in fact left over to the garbage collector, but this works only for scoped variables: if a variable is scope free, the GC never knows whether we will still use it or not.

Last, by giving types to scope free variable in a way that does not unify types for all occurrences of the same name, but only when they represent the same variable, we may make possible a wider use of these variables, which are, in our theory, "lighter" —easier to model— than scoped ones.

## Appendix A. Stream Monoid

The definition of stream concatenation is based on a notion of $n^{\text{th}}$ 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 8 (free position)** *1. The $n^{\text{th}}$ 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 $n^{\text{th}}$* free position *for $p$ in $r$ is the $n^{\text{th}}$ 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.

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

We extend both $\phi$ and $\psi$ 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 $\psi$).

**Example 1 (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 3 (reversibility)** *$\phi_r$ 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 9 (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 4 (monoid)** *Concatenation as in Definition 9 is an associative application $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$, accepting $\{\}$ as neutral element.*

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 $n^{\text{th}}$ free position for $p$ in $r$. $\phi_r$ is the function which does this shifting. Reciprocally, extraction uses $\psi_r$ to shift back positions in the rest to their relative indexes w.r.t $r$.

**Example 2 (stream concatenation)**

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

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

## References


1. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.
2. J. Garrigue. The transformation calculus. Technical Report 94-09, University of Tokyo, Department of Information Science, Apr. 1994.
3. J. Garrigue. *Label-Selective Lambda-Calculi and Transformation Calculi*. PhD thesis, University of Tokyo, Department of Information Science, Mar. 1995.
4. J. 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.
5. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 458–471, 1994.
6. A. R. Meyer and K. Sieber. Toward fully abstract semantics for local variables. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.
7. M. Odersky. A functional theory of local names. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 48–59, 1994.
8. M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment, and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 43–56, 1993.
9. F. Oles. Type algebras, functor categories, and block strcutures. In N. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 15, pages 543–573. Cambridge University Press, 1985.
10. G. D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1, 1975.
11. J. 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.
12. B. A. Trakhtenbrot, J. Y. Halpern, and A. 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.