数理解析研究所講究録
第851巻 1993年 34-45

The Transformation Calculus and its Typing

Jacques Garrigue

The University of Tokyo, Faculty of Science,
Dept. of Information Science, Yonezawa Lab.
Hongo 7-3-1, Bunkyo-ku, TOKYO 113
garrigue@is.s.u-tokyo.ac.jp

July 19, 1993

Abstract

Many calculi supporting a notion of state have been proposed. However this notion is nearly always based on the intuition of a store, that is a binding from name to values. The exception, monads, recently focused on for I/Os, suffers from its rigidity.

The transformation calculus, an extension of lambda calculus, shows another, more general way to do that. It is different from others in that no orthogonal reduction rule is added to β-reduction, but only structural ones.

We introduce here the transformation calculus, and give our approach to its typing. Fundamental properties, like confluence, have been shown, and two type systems, simple and polymorphic, are proposed.

1 Introduction

What we call state is a complex notion. It can be as well the external state of a device (like for I/O), the internal value of a variable, which changes during the evaluation, or even, why not, a local value. What makes it a state is fundamentally the way we look at it, the meaning we give to it.

The answer to the two different demands of external and internal mutables, has been recently given in two different ways. For externals, the use of an I/O monad has been suggested [PW93]. It introduces an idea of composition, very relevant to our calculus too. For internals, the concept of store is still largely predominant, with different formalizations. Lamping [Lam88] proposes a generic system, where abstractions can be made on names, and stores can be manipulated freely, whose power is comparable to lambda calculus. Johnson and Duggan [JD88] introduce the notion of partial continuation to capture the modifications of a store by a piece of program. Odersky, Rabin, Hudak [ORH93] define a calculus combining β-reduction with operations on references, and prove that it can be executed by a store model. These are untyped systems, but effects [TJ92] give an abstraction of accesses done to the store, and can be used jointly with types.

These two approaches are not completely opposed, and we can use monads to represent internal states, or references for I/O. For this last remark, one may remember that on many computers, access to ports is done by writing the memory, which is the only medium for side-effects. Monads offer many other possibilities, as pointed out in [Mog91, Wad90], like non-determinism. But combining different monads together presents some difficulties. Effects are easy to combine together, but present little abstraction from the operational model.

If we must compare transformation calculus to these, we can say that it integrates the notion of semantic composition from monads, with the use of names (or labels) which makes so intuitive stores. However it differs deeply from them by the the presence of a unique reduction rule, an extension of β-reduction. This unified vision makes it quite close to CCL [Cur86, Har89], but with quite a different goal — CCL is intended to give a basis for compiling λ-calculus without name bindings — and a more practical representation of environments, since we reintroduce names.

What the transformation calculus manipulates as an environment is no longer a store, but something we will call a *stream*. The particularity of streams is that *labels* are composite, with a name and a numerical index. Two values with same name but different indexes are completely independent semantically. In a typed framework, they do not need to have the same type. They are similar to two values of a variable in different scopes. The presence of this numerical index permits to define composition of streams. When we compose two streams, label-value pairs in the first stream are left unchanged, but pairs from the second one are added with modified indexes. This way, we avoid overlapping of labels.

Using this structure, we can first define a selective λ-calculus [AKG93, GAK93]: functions are applied to such streams and *select* their arguments through labels. Unused arguments are not discarded, but kept for the result to be applied to them. All this can be translated into the addition of labels to applications and abstractions in the classical lambda-calculus.

If we accept to have streams as result too, then we have transformation calculus. We introduce a notion of composition, where output of a *transformation* (a function whose result is a stream) is "connected" to input of another one, according to labels. This is the central concept of transformation calculus, and it makes it similar to the use of monads, in a more flexible form. If we abstract names of labels, transformation calculus may be better understood as dataflow graphs: a transformation is a unit of calculation which gets its input from some units, and sends its result to other ones, labels only denoting these connections.

In the second section we define formally streams, and give their essential properties. The third section is devoted to the presentation of untyped transformation calculus. It is developed in a simply-typed version, where we explicit type inference, in section 4. Section 5 presents shortly how this system can be extended in a polymorphic one. Finally section 6 concludes, and tries to analyze what is the real expressive power of transformation calculus, and its type systems.

2 Stream monoid

Streams differ from records by the existence of an associative concatenation operation which is invertible. We will note the concatenation on streams by a simple dot "$\cdot$", and the monoid of streams is $(\mathcal{S}, \cdot)$. It has a neutral element $\{\}$.

Definition 1 (stream) *1. The set of streams on a domain $\mathcal{A}$ is the set of finite partial functions from $\mathcal{L} = \mathcal{L}_s \times \mathcal{N}$, the set of labels, to $\mathcal{A}$. $\mathcal{L}_s$ is an ordered set of symbols, $\mathcal{N}$ natural integers without 0, and $\mathcal{L}$ is lexicographically ordered.*

$$\mathcal{S} = \{s \in \mathcal{P}(\mathcal{L} \times \mathcal{A}) \mid s \textit{ finite and } \forall (l,a) \in s, (\nexists a' \neq a)\ (l,a') \in s\}$$

$\mathcal{D}_s$ is the definition domain (first projection) of a stream s.

2. *We note defining pairs as $(pn \Rightarrow a)$ with $p \in \mathcal{L}_s, n \in \mathcal{N}, a \in \mathcal{A}$ and $\{\}$ is the function defined nowhere ($\mathcal{D}_{\{\}} = \emptyset$).*

l will always represent an element of $\mathcal{L}$, p and q elements of $\mathcal{L}_s$, m and n elements of $\mathcal{N}$.

Definition 2 (concatenation) 1. *$d_r(pn)$ is the number of occupied positions in r on symbol p and whose index less or equal to n. That is, $d_r(pn) = |(\{p\} \times [1,n]) \cap \mathcal{D}_r|$.*

2. *We define the n^{th} free position for p in r as*

$$\phi_{r,p}(n) = \min\{i \in \mathcal{N} \mid i - d_r(pi) = n\}.$$

Its inverse is $\phi_{r,p}^{-1}(i) = i - d_r(pi)$. We extend these functions to streams by $\phi_r(\{p_1n_1 \Rightarrow a_1, \ldots, p_kn_k \Rightarrow a_k\}) = \{p_1\phi_{r,p_1}(n_1) \Rightarrow a_1, \ldots, p_k\phi_{r,p_k}(n_k) \Rightarrow a_k\}$ and similarly for $\phi_r^{-1}(s)$.

3. *Stream concatenation "$\cdot$", and its converse matching, can now be defined by the following operations. "$\uplus$" is union of functions on disjoint domains.*

$$\begin{aligned} r \cdot s &= r \uplus \phi_r(s) \\ r \uplus s &= r \cdot \phi_r^{-1}(s) \end{aligned}$$

One more definition may help understanding left insertion in a stream: we can shift labels up or down by the following two functions ($\Downarrow_{pn} \circ \Uparrow_{pn} = Id$).

$$(\Uparrow_{pn} r)(qm) = \begin{cases} r(pm-1) & \text{if } q = p \text{ and } m > n \\ r(qm) & \text{if } q \neq p \text{ or } m < n \end{cases}$$
$$(\Downarrow_{pn} r)(qm) = \begin{cases} r(pm+1) & \text{if } q = p \text{ and } m \geq n \\ r(qm) & \text{otherwise.} \end{cases}$$

We have $\Uparrow_{pn}(r) = \phi_{\{pn \Rightarrow a\}}(r)$ and $\Downarrow_{pn}(r) = \phi_{\{pn \Rightarrow a\}}^{-1}(r)$.

Special cases of the definitions define commutation relations for concatenation of singletons.

$$\begin{aligned} \{pm \Rightarrow a\} \cdot \{qn \Rightarrow b\} &= \{qn \Rightarrow b\} \cdot \{pm \Rightarrow a\} & p \neq q \\ \{pm \Rightarrow a\} \cdot \{pn \Rightarrow b\} &= \{pn \Rightarrow b\} \cdot \{pm-1 \Rightarrow a\} & m > n \end{aligned}$$

Proposition 1 *Concatenation as in definition 2 is an associative application $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$, accepting $\{\}$ as neutral element. Its partial applications $\mathcal{S} \to \mathcal{S}$ ($s \cdot \_$ and $\_ \cdot s$) are injective.*

PROOF Associativity comes from the equality $\phi_r \circ \phi_s = \phi_{r \uplus \phi_r(s)}$. For this we reason on inverses: $\phi_{s,p}^{-1}(\phi_{r,p}^{-1}(i)) = i - d_r(pi) - d_s(\phi_{r,p}^{-1}(i)) = i - d_{r \uplus \phi_r(s)}(pi) = \phi_{r \uplus \phi_r(s),p}^{-1}(i)$. 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 ($\phi_{\{\}} = id$). Injectiveness is a consequence of reversibility. □

Since using always conjointly the symbolic and numerical part of a label would be cumbersome, we choose a default symbol ϵ in $\mathcal{L}$, and we will admit the following abbreviations:

- no label at all (a value of the domain alone): $a \sim \{\epsilon 1 \Rightarrow a\}$
- only a numerical label: $\{n \Rightarrow a\} \sim \{\epsilon n \Rightarrow a\}$
- only a symbolic label: $\{p \Rightarrow a\} \sim \{p1 \Rightarrow a\}$

For instance,

$$\begin{aligned} &\{2 \Rightarrow a\} \cdot b \cdot c \cdot \{p \Rightarrow d\} \cdot \{q \Rightarrow e\} \cdot \{p \Rightarrow f\} \\ &= \{\epsilon 1 \Rightarrow b, \epsilon 2 \Rightarrow a, \epsilon 3 \Rightarrow c, p1 \Rightarrow d, p2 \Rightarrow f, q1 \Rightarrow e\}. \end{aligned}$$

$$
\begin{array}{lcll}
l & ::= & pn & p \in \mathcal{L}_s, n \in \mathcal{N} \\
M & ::= & x & \text{variable} \\
 & \mid & \{\} & \text{empty stream} \\
 & \mid & M \cdot \overline{\{l \Rightarrow x, \ldots\}} & \text{abstraction} \\
 & \mid & M \cdot \{l \Rightarrow M, \ldots\} & \text{application} \\
 & \mid & M \circ M & \text{composition}
\end{array}
$$

Figure 1: Syntax of the transformation calculus

3 Syntax

In this section we define the untyped transformation calculus, and its simplified form the selective λ-calculus, and give their fundamental properties.

The definition is done in five steps. First we give a syntactic definition of terms in the transformation calculus, and add a structural equivalence on these terms. Then we define substitutions on our terms, taking care of the constructs that did not appear in λ-calculus. After that we need syntactical notions of *open transformation* and *juxtaposition*, to manipulate the structure of our terms, and finally we can define reduction rules for the transformation calculus.

Definition 3 *A term of the transformation calculus is any proposition written according to the syntax in figure 1, and appropriate parentheses. The set of such terms is Λ_T.*

(Since abstractions and applications are streams the same label may not appear more than once in them; and the same variable may not be bound more than once in the same abstraction.)

To avoid parentheses, "$\circ$" is left-associative, and with same priority as "$\cdot$".

This identity of priority between "$\circ$" and "$\cdot$" means that, without parentheses, an abstraction binds *everything* on its left. For application we will see that it doesn't change the meaning of a term.

The structural equivalence we define on these terms include the structural laws of the stream monoid into transformation calculus' abstractions and applications.

Definition 4 (structural equivalence) *The stream monoid structure is used according to the following two rules, defining "$\equiv$", together with α-conversion. We structurally close $\equiv$ as usual to apply it on subterms.*

$$
\begin{array}{rl}
\forall R, S \in \mathcal{S}(\Lambda_T), & M \cdot R \cdot S \equiv M \cdot (R \cdot S) \\
\forall R, S \in \mathcal{S}(\mathcal{V}), V(R) \cap V(S) = \emptyset \Rightarrow & M \cdot \overline{R} \cdot \overline{S} \equiv M \cdot \overline{(S \cdot R)}
\end{array}
$$

For a stream of variables, V is its image (second projection).

This equivalence means that any term in this calculus can equivalently be written using only unitary application or abstraction streams, by decomposing streams in the monoid. Particularly, when $l_i < l_{i+1}$, we have

$$M \cdot \{l_1 \Rightarrow N_1, \ldots, l_n \Rightarrow N_n\} \equiv M \cdot \{l_1 \Rightarrow N_1\} \cdot \ldots \cdot \{l_n - n + 1 \Rightarrow N_n\}$$

and its symmetrical for abstractions.

Definition 5 *Substitutions are done in the same way as for lambda calculus, but we must pay attention to the new composition (cf. figure 2). For a term or a stream of terms, FV is the set of its free variables (substitutable variables).*

$$\begin{array}{lcll}
x[x\backslash N] & \equiv & N & \\
y[x\backslash N] & \equiv & y & x \neq y \\
\{\}[x\backslash N] & \equiv & \{\} & \\
(M \cdot \overline{R})[x\backslash N] & \equiv & (M \cdot \overline{R}) & x \in V(R) \\
(M \cdot \overline{R})[x\backslash N] & \equiv & (M[x\backslash N]) \cdot \overline{R} & x \notin V(R), V(R) \cap FV(N) = \emptyset \\
(M \cdot R)[x\backslash N] & \equiv & (M[x\backslash N]) \cdot (R[x\backslash N]) & \\
(M \circ M')[x\backslash N] & \equiv & (M[x\backslash N]) \circ (M'[x\backslash N]) &
\end{array}$$

Figure 2: Definition of substitutions

$$\begin{array}{llcll}
 & \text{Reduction} & & & \\
(\beta) & M \cdot \overline{\{l_1 \Rightarrow x_1, \ldots, l_n \Rightarrow x_n\}} \cdot \{l_1 \Rightarrow N_1, \ldots, l_n \Rightarrow N_n\} & \rightarrow & M[x_1\backslash N_1, \ldots, x_n\backslash N_n] & \\
(\circ) & M \circ (\{\} \odot P) & \rightarrow & M \odot P & BV(P) \cap FV(M) = \emptyset \\
 & \text{Reordering} & & & \\
(\cdot\cdot) & M \cdot \overline{R} \cdot S & \rightarrow & M \cdot \phi_R^{-1}(S) \cdot \overline{\phi_S^{-1}(R)} & \mathcal{D}_R \cap \mathcal{D}_S = \emptyset, V(R) \cap FV(S) = \emptyset \\
(\cdot) & (M \circ N) \cdot R & \rightarrow & M \circ (N \cdot R) & \\
(\bar{\cdot}) & M \circ (N \cdot \overline{R}) & \rightarrow & (M \circ N) \cdot \overline{R} & FV(M) \cap V(R) = \emptyset
\end{array}$$

Figure 3: Reduction system for the transformation calculus.

Definition 6 *An* open transformation *is a syntactic entity representing a list of streams of terms, anti-streams of variables, and terms. It can be linked to terms by the juxtaposition meta-operator "$\odot$".*

$$M \odot (e_1, \cdots, e_n) \overset{\text{def}}{=} (\ldots(M * e_1)\ldots) * e_n.$$

where $N * e = N \cdot e$ if e is a stream or an anti-stream, and $N * e = N \circ e$ if e is a term.

The set of variables bound by anti-streams in the body of an open transformation P is $BV(P)$.

This definition introduces a distinction between functional terms (which can be written $x \odot P$) and transformational terms (or closed transformations, since variable bound in their body are local, written $\{\} \odot P$).

Now we are ready to define rewriting rules for the transformation calculus.

Definition 7 *"$\rightarrow$" is defined on transformation calculus terms by the two reduction rules in figure 3, in parallel with three reordering rules. "$\overset{*}{\rightarrow}$" is the combination in any order of $\rightarrow$ and $\equiv$.*

Purely functional terms (not containing $\circ$ nor $\{\}$) and rules $(\beta, \cdot\cdot)$ define the selective λ-calculus.

Rule β is the classical β-reduction extended to streams. $\circ$ "connects" a function and a transformation. The condition, that all variables bound in P should be protected in M, means that we "close" our transformation, hiding its variables from M, before syntactically juxtaposing them.

Reordering rules "$(\cdot)$" and "$(\bar{\cdot})$"are easily understandable. They respectively send the input of a composition to its first element (the right one), and externalize abstractions on its first element. Only one of them would be enough, but we give the two for symmetricity.

Rule $(\cdot\cdot)$ is based on the equality $N\cdot\overline{(R \uplus S_V)}\cdot(R_{\Lambda_T} \uplus S) \equiv N\cdot\overline{\phi_R^{-1}(S_V)}\cdot\overline{R}\cdot S\cdot\phi_S^{-1}(R_{\Lambda_T})$ (with $\mathcal{D}_{R_{\Lambda_T}} = \mathcal{D}_R, \mathcal{D}_{S_V} = \mathcal{D}_S$ and $V(S_V) \cap V(R) = \emptyset$). If we take $M = N\cdot\overline{\phi_R^{-1}(S_V)}$, and apply $M\cdot\overline{R}\cdot S$ to $\phi_S^{-1}(R_{\Lambda_T})$, then rule $(\cdot\cdot)$ preserves confluence: it gives $N\cdot\overline{(R_V \uplus S_V)}\cdot(R_{\Lambda_T} \uplus S_{\Lambda_T}) \rightarrow N\cdot\overline{\phi_R^{-1}(S_V)}\cdot\phi_R^{-1}(S)\cdot\overline{\phi_S^{-1}(R)}\cdot\phi_S^{-1}(R_V)$.

Thanks to that we have the Church-Rosser property, which makes our calculus meaningful.

Theorem 1 *Transformation 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 We can adapt from [AKG93] that selective λ-calculus in the definition we give here is confluent.

For proving the confluence of transformation calculus we use the following translation suppressing transformational constructs by using a new symbolic label, *cont*.

$$\begin{array}{rcl}
Tr(\{\}) & = & x\cdot\overline{\{cont \Rightarrow x\}} \\
Tr(M \circ N) & = & Tr(N)\cdot\{cont \Rightarrow Tr(M)\} \\
Tr(x) & = & x \\
Tr(M\cdot\overline{R}) & = & Tr(M)\cdot\overline{R} \\
Tr(M\cdot R) & = & Tr(M)\cdot Tr(R)
\end{array}$$

Then, there is a direct correspondence between reductions in the two calculi, composition being replaced by β-reduction.

Interestingly, this translation gives simultaneously partial continuation semantics (*cf.* [JD88]) to the transformation calculus: to compose two transformations is to apply the second one to the first, viewed as a continuation we will call last. □

4 State handling

Since it includes the lambda-calculus, transformation calculus is already Turing-complete. But the interesting point is to see how we can use labels to encode notions that are not naturally encoded in the lambda-calculus.

We propose here a notion of *scope-free* variable, which is able to capture the various definitions of state we gave in introduction. By *scope-free* we mean that the scope is not limited to a syntactically determined part of the program. However we do not mean global either: we may have states that are only *locally* meaningful, but this locality is not necessarily restricted to a syntactic scope. For such a variable we have three operations: creation, update, and destruction. They are all represented by transformations.

Creation is just putting a binding on the stream. The stream we are talking about here is an execution notion: we can see a program in the transformation calculus as a succession of transformations modifying a stream, and finally either returning the final state of the stream or a value. To create a scope-free variable with label l and value v, we execute the transformation $\{\}\cdot\{l \Rightarrow v\}$.

To represent a state modification in transformation calculus we change the binding of its label, like this

$$\{\}\cdot\{l \Rightarrow M\}\cdot\overline{\{l \Rightarrow x\}}.$$

It gets the value on l, and puts a new value in its place, possibly depending on x. If we think of an applied calculus for I/O, x may be a representation for an external state, and M that of the resulting state.

Last we must delete a variable from the stream if we want to keep it local to a certain part of the execution. We can of course purely suppress it by $\{\}\cdot\overline{\{l\Rightarrow x\}}$ which reads x but do not use it, but more generally we apply a transformation which takes l in input and do not put a new value for it.

Example We will just give here the basic example of how to define a point and move it.

- We define it by $defpoint = \{\}\cdot\{mypoint\Rightarrow\{\}\cdot\{x\Rightarrow 0, u\Rightarrow 0\}\}$. Here we are using the trivial encoding of records as transformations for $\{\}\cdot\{x\Rightarrow 0, y\Rightarrow 0\}$.
- We can inspect its contents by $lookpoint = p\cdot\{mypoint\Rightarrow p\}\cdot\overline{\{mypoint\Rightarrow p\}}$. For instance $lookpoint \circ defpoint = \{\}\cdot\{mypoint\Rightarrow\{\}\cdot\{x\Rightarrow 0, y\Rightarrow 0\}, x\Rightarrow 0, y\Rightarrow 0\}$.
- This last transformation moves the point according to the values passed on x and y: $move = \{\}\cdot\{mypoint\Rightarrow\{\}\cdot\{x\Rightarrow x+x', y\Rightarrow y+y'\}\cdot\overline{\{x\Rightarrow x, y\Rightarrow y\}}\circ p\}\cdot\overline{\{mypoint\Rightarrow p, x\Rightarrow x', y\Rightarrow y'\}}$. We can calculate $move\cdot\{x\Rightarrow 10, y\Rightarrow 15\}\circ defpoint = \{\}\cdot\{mypoint\Rightarrow\{\}\cdot\{x\Rightarrow 10, y\Rightarrow 15\}\}$.

This example shows that scope-free variables let us practice some sort of object-based programming. In fact this encoding permits much stranger things. For instance, even in a typed calculus, we can change the type of a variable when we modify it, since the new definition is totally independent of the previous. Modification is a destruction followed by a creation.

More interesting, the "scope" of the scope-free variable, that is from its creation to its destruction, enjoys the same "masking" property as syntactical ones. If we create successively two variables on the same label, the second one will be apparent and the first masked until we destroy the second, and then the first one will reappear. This possibility makes scope-free variables an alternative to scoped variables. We can even compile this calculus without any notion of environment, putting all variables "on the stream".

5 Simple types

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, which look very much like record types, 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 8 *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 $(\{\}\to\tau)=\tau$, to shorten.

These last restrictions make a stream type a stream of types as defined in Section 2. This means that we can use stream composition on these types, as we will do for typing rules.

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

$$\Gamma \vdash \{\} : \{\} \tag{II}$$

$$\frac{\Gamma[x_1 \mapsto \theta_1] \ldots [x_n \mapsto \theta_n] \vdash M : r \to \tau}{\Gamma \vdash M \cdot \overline{\{l_1 \Rightarrow x_1 : \theta_1, \ldots, l_n \Rightarrow x_n : \theta_n\}} : (\{l_1 \Rightarrow \theta_1, \ldots, l_n \Rightarrow \theta_n\} \cdot r) \to \tau} \tag{III}$$

$$\frac{\Gamma \vdash M : (\{l_1 \Rightarrow \theta_1, \ldots, l_n \Rightarrow \theta_n\} \cdot s) \to \tau \qquad \Gamma \vdash N_i : \theta_i}{\Gamma \vdash M \cdot \{l_1 \Rightarrow N_1, \ldots, l_n \Rightarrow N_n\} : s \to \tau} \tag{IV}$$

$$\frac{\Gamma \vdash M : (r_2 \cdot r) \to \tau \qquad \Gamma \vdash N : r_1 \to r_2}{\Gamma \vdash M \circ N : (r_1 \cdot r) \to \tau} \tag{V}$$

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

Figure 4: Typing rules for simply typed transformation calculus

Definition 9 *A term in the simply typed transformation calculus is constructed according to the following syntax.*

$$M \; ::= \; x \mid \{\} \mid M \cdot \overline{\{l \Rightarrow x : t, \ldots\}} \mid M \cdot \{l \Rightarrow M, \ldots\} \mid M \circ M$$

with the same constraints on labels and variables as before.

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

Definition 10 *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 4.*

Rules (I,III,IV) 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 $l_n = \epsilon n$).

Rule (II) types the constant $\{\}$. 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 $\{\}$ 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.

Traditional results about simply typed lambda-calculus stand.

Theorem 2 (object reduction) *If $\Gamma \vdash M : \tau$ and $M \xrightarrow{*} N$ in the transformation calculus, then $\Gamma \vdash N : \tau$.*

Theorem 3 (strong normalization) *If for some Γ and some τ, $\Gamma \vdash M : \tau$, then there is no infinite reduction starting from M in the transformation calculus.*

PROOF by constructing a directed acyclic graph calculating an upper bound of the longest reduction. □

Such a type system is not polymorphic, but it is more generic than what we would have obtained by the translation towards selective λ-calculus. In this translation, composition $M \circ T$ is interpreted as the passing of a continuation to the transformation, $T \cdot \{cont \Rightarrow M\}$, which means that when typing T we fix the type of the eventual continuation. Thanks to the rule (VI), this is not the case here: the continuation M must only be able to accept all the output of T, but its result has no link with T's type.

For instance, suppose we have a language modelling the transformation calculus, with basic arithmetic operation. We define a transformation on a pair of values *add_sub*:

$$\{x + y, x - y\} \cdot \overline{\{x : int, y : int\}} : \{1 \Rightarrow int, 2 \Rightarrow int\} \rightarrow \{1 \Rightarrow int, 2 \Rightarrow int\}$$

where we abbreviate $\{\} \cdot \{P\}$ in $\{P\}$.
Now we can compose it with $mult = (x * y) \cdot \overline{\{x : int, y : int\}}$ and obtain

$$((x + y) * (x - y)) \cdot \overline{\{x : int, y : int\}} : \{1 \Rightarrow int, 2 \Rightarrow int\} \rightarrow int$$

or with itself for

$$\{x + x, y + y\} \cdot \overline{\{x : int, y : int\}} : \{1 \Rightarrow int, 2 \Rightarrow int\} \rightarrow \{1 \Rightarrow int, 2 \Rightarrow int\}$$

Application can be done on related labels (with the good type), or unrelated ones, the type being free then:

$$\begin{aligned} add\_sub \cdot \{2 \Rightarrow 5, ok \Rightarrow true\} = \{1 \Rightarrow x + 5, 2 \Rightarrow x - 5, ok \Rightarrow true\} \cdot \overline{\{1 \Rightarrow x : int\}} \\ : \{1 \Rightarrow int\} \rightarrow \{1 \Rightarrow int, 2 \Rightarrow int, ok1 \Rightarrow bool\} \end{aligned}$$

6 A glimpse of polymorphic types

This polymorphism which is not becomes a problem when we try to polymorphically type transformation calculus expressions. Suppose for instance that we want to define composition as a function $comp \cdot \{f \Rightarrow f, g \Rightarrow g\} = f \circ g$. If we have only generic type variables, we cannot express the relation which is between the output stream of g and the input stream of f.

The intuitive solution to this problem is the introduction of stream variables, for stream types. It is indeed quite expressive. Suppose *comp* defined above to be typed $\{f \Rightarrow (\rho \rightarrow \alpha), g \Rightarrow (\rho' \rightarrow \rho)\} \cdot \rho' \rightarrow \alpha$. It correctly expresses the constraint, even permitting to receive arguments on labels other than $f1$ and $g1$, linking their types with types in g's input.

However it will appear not to be enough. The second problem comes from rule (VI) of typing, which expresses than any transformation has an infinity of types. But, when we write $\rho' \rightarrow \rho$ we are interested in only one of these types, whose ρ is the same as in $\rho \rightarrow \alpha$, which may be a simple function. In this case the maximal possible ρ is the total input of f. If we let α be any type, it may contain part of the input and reduce the genericness of g. This is why we need a last sort of type variables, we will call them return variables, and they are restricted to represent return types.

s	$::=$	$\bullet \mid \diamond$	return/stream sort
v	$::=$	$\alpha \mid \beta \mid \ldots$	variables
r	$::=$	$\{l \Rightarrow t, \ldots\} \mid \{l \Rightarrow t, \ldots\} \cdot v^{\diamond}$	stream types
w	$::=$	$u \mid v^{\bullet} \mid r$	return types
t	$::=$	$\{l \Rightarrow t, \ldots\} \rightarrow v \mid r \rightarrow w$	monotypes
σ	$::=$	$t \mid \forall \alpha . \sigma$	polytypes

During type instantiation an unsorted variable may be substituted with any type, but a return ($\bullet$) variable is restricted to return types, and a stream ($\diamond$) one to stream types.

We do not give here a type inference algorithm. Such inference is possible, but complex. Done in a direct way it is incomplete, peculiarities of transformations introducing non-monotonicity. Here, we can suppose that types are declared, like in the simply typed calculus, and that type checking is a problem of matching, and not unification, which becomes easily decidable.

In this system composition will be typed:

$$comp : \{f \Rightarrow (\alpha^\circ \rightarrow \beta^\bullet), g \Rightarrow (\gamma^\circ \rightarrow \alpha^\circ)\} \cdot \gamma^\circ \rightarrow \beta^\bullet.$$

Most functions can be typed in this formalism. There are still exceptions. The simplest one is auto-composition: $(x \circ x) \cdot \overline{\{x\}}$. The only polymorphic type we can give it is $\{1 \Rightarrow (\alpha^\circ \rightarrow \alpha^\circ)\} \cdot \alpha^\circ \rightarrow \alpha^\circ$. If our intention was to use it as stream duplication $(\{p \Rightarrow a, q \Rightarrow b\} \mapsto \{p1 \Rightarrow a, p2 \Rightarrow a, q1 \Rightarrow b, q2 \Rightarrow b\})$, this will not work.

Example We go on with our point example, and give types for our different transformations.

- $defpoint : \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\}\}$.
- $lookpoint : \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\circ\} \rightarrow \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\circ, x \Rightarrow int, y \Rightarrow int\}$.
- $move : \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\circ, x \Rightarrow int, y \Rightarrow int\} \rightarrow \{mypoint \Rightarrow \{x \Rightarrow int, y \Rightarrow int\} \cdot \alpha^\circ\}$.

We see here that polymorphism gives us "inheritance" for streams. For instance, if we apply *move* on a *mypoint* with more fields than x and y, it will work and leave other fields unchanged.

7 A polymorphic record calculus

An extension suggested by this polymorphism on streams is the use of an extended pattern matching on them. Extracting a field from a stream can then be done polymorphically, without extending the typing system. This is all we need to get a complete polymorphic record calculus, since we already had modification. Here is the example of a field selector for label l. $\{l \Rightarrow x\} \cdot y$ is an extended pattern, matching the value on l with x and the rest of the stream with y.

$$\#l = x \cdot \overline{\{1 \Rightarrow \{l \Rightarrow x\} \cdot y\}} : \{1 \Rightarrow \{l \Rightarrow \alpha\} \cdot \beta^\circ\} \rightarrow \alpha$$

However such types are fragile, in that the same polymorphic stream should not be used as a transformation, since constraints would interfere. This is not surprising: transformation composition gives us record concatenation, and such an operation has no most generic type (see [Wan91] in a slightly different system: generally label repetition is handled by retaining only one of the two occurrences, and we keep the two by our label-shifting). If this rule against mixing is respected, we can enjoy record operations as an "extra".

8 Conclusion

The transformation calculus proposed here is a practical way of representing state modifications, while staying in a λ-calculus structure. If the name part of labels keeps the intuition of identity like it can be in stores, the numerical part permits to have a completely relative representation of applications and abstractions, which is necessary to express properly commutation. A calculation can then be written in an entangled way: $(\{p \Rightarrow x+1\} \cdot \overline{\{p \Rightarrow x\}}) \circ (\{q \Rightarrow x\} \cdot \overline{\{p \Rightarrow q\}})$ is equivalent to $(\{q \Rightarrow x\} \cdot \overline{\{p \Rightarrow x\}}) \circ (\{p2 \Rightarrow x+1\} \cdot \overline{\{p2 \Rightarrow x\}})$ since the two p's here expect independent values.

These properties makes it a good formalism to analyze order dependency in computations using states. Moreover, the type systems we proposed express directly these dependencies in the type itself. One criticism could be that we do not explicitly handle the case in which we only read a value, and give it back without modification on the same label, which is an important one for commutation. But this can be done by a simple program transformation, like $\{p \Rightarrow x, q \Rightarrow M\} \cdot \overline{\{p \Rightarrow x\}} \Longrightarrow (\{q \Rightarrow M\} \cdot \overline{\{dup \Rightarrow x\}}) \circ (\{p \Rightarrow x, dup \Rightarrow x\} \cdot \overline{\{p \Rightarrow x\}})$, which desynchronizes reading and use of the value, and makes possible commutation with the operational part of another transformation using p without modifying it.

This gives a possible use of such a calculus. We can extract order dependencies in sequentially written programs, to be able to compile them in a concurrent way, and do it only with typing information. A way to obtain such a result is analyzing each operation into a sequence of fine-grain transformation, whose type express these dependencies. In this perspective transformation calculus is a quite applied one.

However, the possibility suggested last of introducing record field selection by an extended pattern matching, and this without modifying the type system, shows the generality of transformation calculus. We are using here the similarity between records and environments to apply methods working on one to the other. This is for us a confirmation that the notion of state should not be limited to internal or external values changing with time, but can well be applied to unchanging things, like the quality of something as expressed in a record; or to data transmitted between different parts of a program, with only a transient existence.

To conclude, the essence of transformation calculus is certainly this possibility of communicating data outside of a syntactical scope in a structured way. The tradition of structured programming is to limit the scope of each variable to its block, blocks being structured in a hierarchy. Same may be said for lambda-calculus, without the possibility of changing bindings. Transformation calculus suppresses this limit, by making data flow between transformations. This is dangerous, since this identity of syntactical and execution scopes was a protection against errors, but we hope to overcome this danger by the strength of a more expressive typing.

References

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

[Cur86] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, 1986.

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

[Har89] Thérèse Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[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.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[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.

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

[Wad90] Philip Wadler. Comprehending monads. In *Proc. ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.