Proving and applying program transformations expressed with second-order patterns

Gerard Huet, Bernard Lang

IRIA
laboria

Institut de Recherche
d'Informatique
et d'Automatique

Domaine de Voluceau
Rocquencourt
B. P. 105 78150 - Le Chesnay
France
Tél.: 954 90 20

laboratoire de recherche
en informatique
et automatique

PROVING AND APPLYING PROGRAM TRANSFORMATIONS EXPRESSED WITH SECOND-ORDER PATTERNS

Gérard HUET
Bernard LANG

Rapport de Recherche N° 266

Novembre 1977

39p

PROVING AND APPLYING PROGRAM TRANSFORMATIONS EXPRESSED WITH SECOND-ORDER PATTERNS

Gérard Huet, Bernard Lang
IRIA/LABORIA

Résumé :

Nous proposons une technique de transformations de programmes basée sur la réécriture de schémas de second ordre. Un algorithme de reconnaissance complète est défini. On montre comment valider formellement les transformations, en utilisant des méthodes de preuve basées sur une sémantique dénotationnelle. La méthode est illustrée sur des exemples concernant la transformation de récursions en itérations.

Abstract :

We propose a program transformation technique based on rewriting second-order schemas. A complete matching algorithm is given, and the correctness of the method is proved, using denotational semantics. The method is illustrated with recursion removal examples.

Introduction

There is a huge gap between practical software certification techniques and the theoretical tools defined for formal proofs of programs. One way to close this gap is to write interactive systems that will help the programmer to design, debug, run and ultimately validate his programs.

A desirable feature of such a system is the ability to manipulate programs into various forms, while preserving their meaning.

Such a "program massaging" facility has been advocated by many people [3], [4], [7], [15], [24].

As a systematic approach to high-level optimization for programs, it is the natural complement to structured programming development techniques, which tend to sacrifice program efficiency to clarity and good organization.

One of the most promising techniques is due to Darlington and Burstall. Program transformations are defined as schematic rewriting systems, together with constraints on the instances of the schemas that must be met in order for the transformation to be valid.

We shall here formalize this method, using a second order term language to represent program schemas. We give a uniform matching algorithm to apply the transformations to parts of a program. We show how one can formally validate the transformations, using denotational semantics. Finally we discuss the problems related to organizing a system of schematic transformations.

1. Inducing transformation templates from program transformations

Let us first give an example of the schematic transformation method on an example in recursion removal.

Let us suppose we are interested in replacing recursions by iterations. Over the integers, the problem has a well-known but trivial solution : every partial recursive function may be computed with a flowchart using only three registers. However we are not interested in such "Turing machine monster", and we want to favor transformations based on control structure equivalences, as opposed to data encodings. This is why we are more interested in transformations expressed by schemes (programs computing over arbitrary interpretations).

On program schemes it is possible to express the fact that recursion is more powerful than iteration : Paterson and Hewitt [22] showed that the following program scheme may not be computed by a flowchart :

$$f(x) \Leftarrow \mathit{if}\ p(x)\ \mathit{then}\ a(x)\ \mathit{else}\ h(f(b(x)), f(c(x))).$$

(of course we do not allow storage stacks !).

Actually, few recursion schemes are translatable into iterations at the schematic level [25], [28].

Darlington and Burstall proposed in [3] to consider program transformations in the class of all the interpretations that satisfy some axioms concerning the data manipulated by the program.

For instance, consider the factorial program :

$$P : \mathrm{fact}(x) \Leftarrow \mathit{if}\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ x * \mathrm{fact}(x-1)$$

which computes fact(n) as : $n * ((n-1) * (\cdots * (2 * (1*1))\cdots))$; Rearranging this expression as :

$$(((\cdots(n * (n-1)) * \cdots) * 2) * 1) * 1$$

suggests using a different computation sequence, which corresponds to the following iterative program, written in ALGOL- like style :

```
P' : function fact(x) ;
        if x=0 then fact := 1 else
           begin
           fact := x ;
           x := x-1 ;
           while x ≠ 0 do
                 begin
                 fact := fact * x ;
                 x := x-1 ;
                 end ;
           [fact := fact * 1]
           end
```

The equality of the two computation expressions above depends solely on the associativity of the operator *. We keep the useless statement at the end of P', which corresponds to the last "1" in the second sequence, because we do not want to use the additional property that 1 is a right identity for *.

Let us now "abstract" the corresponding schemas. We get :

Σ : f(x) <= *if* a(x) *then* b(x) *else* h(d(x), f(e(x))).

and :

```
Σ' : function f'(x) ;
        if a(x) then f' := b(x) else
           begin
           f' := d(x) ;
           x := e(x) ;
           while not a(x) do
                 begin
                 f' := h(f',d(x)) ;
                 x := e(x) ;
                 end ;
           f' := h(f',h(x)) ;
           end
```

Our conjecture at this point is that the two schemas compute the same

function, in every interpretation that satisfies the axiom :

$$X_1 : \forall xyz\ h(h(x,y),z) = h(x,h(y,z)).$$

If the statement between brackets in P' had been suppressed, we would get a more restrictive condition, i.e. we would have to add an axiom $\forall xy\ a(y) \supset h(x,b(y)) = x$, which is not relevant to the recursion removal and would reduce the applicability of the transformation.

Let us call *transformation template* a triple such as $<\Sigma,\Sigma',\{X_1\}>$. There are now four problems to be solved :

1) How do we discover such templates ?
2) How do we validate them ?
3) How do we recognize a template is applicable to a given program ?
4) How do we organize a system applying automatically such templates ?

Problem 1 may be considered as one of the basic problems of communicating knowledge about programming. Although it is the focus of much research at this time, surprisingly few different program transformations are known. For instance, on the topic of recursion removal, Darlington's thesis [2] contains only 10 templates, that we were actually able to generalize to only 6. In the appendix, we list these and a few others. We have not attempted to list all known recursion removal techniques ; in particular we have not considered transformations requiring counters or stacks. Other recursion removal transformations that could be represented by templates may be found in [28,36,37,...]. Templates pertaining to more general transformations may be found in Loveman [15], Gerhart [6] and Standish [23].

Our paper is mainly concerned with problems 2 and 3. We shall see in the next section a uniform matching algorithm, complete over a wide class of schemas, which answers problem 3. Then section 3 shows how to formally validate transformations templates, using program proving techniques based on denotational semantics. It is our thesis that formal proof techniques, although rightly criticized as being difficult to apply to non-trivial programs, can be put to practical use for the rigorous validation of program transformation systems. We have completely proved all the templates presented in the appendix and in the process discovered missing assumptions in transformations given in [3]. For instance, we shall see in section 3 that in order to validate the template above, we need a supplementary axiom X_2.

2. Matching programs to schemes

2.1. Example

Our problem here is to consider our program schemes as *second-order patterns*. For instance, P above may be considered as an instance of the schema Σ, where a,b,h,d and e are now treated as second-order variables. We have $P = \sigma\Sigma$, with σ being the second-order substitution :

$$\begin{cases} a \leftarrow \lambda x \cdot x=0 \\ b \leftarrow \lambda x \cdot 1 \\ h \leftarrow \lambda xy \cdot x*y \\ d \leftarrow \lambda x \cdot x \\ e \leftarrow \lambda x \cdot x-1 \end{cases}$$

Such second-order patterns are a powerful way of expressing a wide class of programs. Let us give a few programs that are instances of Σ :

- program reversing a list

$$\text{reverse}(x) \Leftarrow \textit{if}\ \text{Null}(x)\ \textit{then}\ x\ \textit{else}\ \text{Append}(\text{reverse}(\text{Cdr}(x)), \text{Cons}(\text{Car}(x),\text{Nil}))$$

  using the substitution

$$\begin{cases} a \leftarrow \lambda x \cdot \text{Null}(x) \\ b \leftarrow \lambda x \cdot \text{Nil} \\ e \leftarrow \lambda x \cdot \text{Cdr}(x) \\ h \leftarrow \lambda xy \cdot \text{Append}(y,x) \\ d \leftarrow \lambda x \cdot \text{Cons}(\text{Car}(x),\text{Nil}) \end{cases}$$

- program computing if a number is perfect :

$$\text{perfect}(y) \Leftarrow y = f(y-1)\ \textit{where}\ f(y) \Leftarrow \textit{if}\ x=1\ \textit{then}\ 1\ \textit{else}\ f(x-1) + \textit{if}\ \text{Div}(x,y)\ \textit{then}\ x\ \textit{else}\ 0.$$

Here the inner recursive function is an instance of Σ, using the substitution :

$$\begin{cases} a \leftarrow \lambda x \cdot x=1 \\ b \leftarrow \lambda x \cdot 1 \\ h \leftarrow \lambda xy \cdot y+x \\ e \leftarrow \lambda x \cdot x-1 \\ d \leftarrow \lambda x \cdot \textit{if}\ \text{Div}(x,y)\ \textit{then}\ x\ \textit{else}\ 0 \text{ (note : y is free here)} \end{cases}$$

For these examples, it is easy to check the associativity of the function substituted to h. Note that, for the last example, the last statement of the iterative translation $\sigma\Sigma'$ is necessary.

2.2. A second-order term language.

Since programs denote functions, it is natural to treat them as second-order objects. Our formalism will therefore represent programs, and schemas denoting families of programs, by terms in a second-order logic.

We assume first that our programming language is defined by an abstract syntax. With every operator or construct of the language is associated a typed operator of fixed arity. For instance, if we have the types "variable", "expression" and "instruction", the assignment operator would have type (variable x expression → instruction).

a) Types

We assume there is a finite set T_0 of elementary types, and the set T of types is the smallest superset of T_0 closed by the operation :

$$\alpha_1,\alpha_2,\cdots,\alpha_n\in T \quad \& \quad \beta\in T_0 \implies (\alpha_1 x\alpha_2 x\cdots x\alpha_n\to\beta)\in T.$$

We define the *order* of type τ recursively by :

- if $\tau\in T_0$ $\quad O(\tau) = 1$
- if $\tau = (\alpha_1 x\alpha_2 x\cdots x\alpha_n\to\beta)$ $\quad O(\tau) = \max\{O(\alpha_i)\,|\,1\le i\le n\}+1$.

Notation if $\tau = (\alpha_1 x\cdots x\alpha_n\to\beta)$, $(\alpha\to\tau)$ stands for $(\alpha x\alpha_1 x\cdots x\alpha_n\to\beta)$.

b) Terms .

The set T of terms is built as follows.

We first define the set of *atoms* $A = C\cup V$, where :

- C is a set of *constant* symbols
- V is a set of *variable* symbols
- $C\cap V = \emptyset$
- every ϕ in A is given a type $\tau(\phi)\in T$
- for every α in T, there is a denumerable subset V_α of V of variables of type α.

The set of terms is built from :

. Atomic terms

If $t\in A$ and $\tau(t)\in T_0$, then $t\in T$. (+)

(+) Atoms of non-elementary type are ruled out as terms for technical convenience ; equivalent terms may be built through explicit abstraction. This permits us to avoid η-conversion problems.

∘ Applications

If $\phi \in A$, and $t_1, t_2, \cdots, t_n \in T$, with $\tau(\phi) = (\alpha_1 x \alpha_2 x \cdots x \alpha_n \to \beta)$ and $\tau(t_i) = \alpha_i$, then $t = \phi(t_1, \cdots, t_n) \in T$, with $\tau(t) = \beta$.

By convention, $\phi(t_1, \cdots, t_i)$, with $i<n$, stands for

$$\lambda u_{i+1} \cdots u_n \cdot \phi(t_1, \cdots, t_i, u_{i+1}, \cdots, u_n).$$

∘ Abstractions

If $t \in T$, with $\tau(t) \in T_0$, and $u_1, u_2, \cdots, u_n \in V$, then $t' = \lambda u_1 u_2 \cdots u_n \cdot t \in T$, with $\tau(t') = (\tau(u_1) x \cdots x \tau(u_n) \to \tau(t))$.

We note $t = t'$ iff t and t' are identical up to a one-one renaming of their bound variables (α-conversion).

Example :

Let $T_0 = \{B, I\}$. The program schema Σ given in section 1 could be represented as the term :

$$\lambda u \cdot Y(\lambda f\ x \cdot C(a(x), b(x), h(d(x), f(e(x)))), u)$$

with $C = \{C, Y\}$ (conditional and fixpoint operators)

$V = \{x, f, a, b, h, d, e\}$

and $\tau(x) = I$, $\tau(a) = (I \to B)$, $\tau(b) = \tau(d) = \tau(e) = \tau(f) = (I \to I)$,

$\tau(h) = (I x I \to I)$, $\tau(C) = (B x I x I \to I)$, $\tau(Y) = (((I \to I) x I \to I) x I \to I)$.

(I stands for integers, B for booleans).

c) Type constraints

In order to restrict ourselves to second order terms, we assume that the variable symbols in the set V have their types restricted to be :

- either in T_0 (first-order variables)
- or of the form $(\alpha_1 x \cdots x \alpha_n \to \beta)$ with $\tau(\alpha_i) \in T_0$ (function variables).

d) Substitutions

Let us first introduce a notation. If $t \in T$, $x_1, x_2, \cdots, x_n$ are first-order variables, and $t_1, \cdots, t_n$ are terms in T such that $\tau(t_i) = \tau(x_i) \in T_0$, we denote by $t[x_i/t_i, 1 \le i \le n]$ the term t, in which every free occurrence of x_i is replaced by t_i. (As usual, a free occurrence of x is an occurrence which is not inside the scope of some $\lambda \cdots x \cdots$). It should be clear that $t[\cdots]$ is a term of the same type as t.

We define a *substitution pair* as a pair $\langle \phi, t \rangle$ where $\phi \in V$ and $t \in T$, in which $\tau(t) = \tau(\phi)$.

A *substitution* σ is now defined as a finite set of substitution pairs, pertaining to a set $\mathcal{D}(\sigma)$ of distinct variables. For every $t \in T$, we define inductively $\sigma t \in T$ by :

. if $t \in A$: if $\exists <t,t'> \in \sigma$ then $\sigma t = t'$, otherwise $\sigma t = t$.

. if $t = \phi(t_1,\cdots,t_n)$:

- if $\exists <\phi, \lambda x_1 \cdots x_n \cdot t'> \in \sigma$ then $\sigma t = t' \lceil x_i \backslash \sigma t_i, 1 \le i \le n \rceil$

- otherwise $\sigma t = \phi(\sigma t_1,\cdots,\sigma t_n)$.

. if $t = \lambda u_1 \cdots u_n \cdot t'$ then $\sigma t = \lambda u_1 \cdots u_n \cdot \sigma t'$

(We ignore here α-conversion problems, and assume u_i does not appear in σ).

Example :

If $t = f(f(x))$

and $\sigma = \{<x,A>,<f,\lambda x \cdot G(x,x)>\}$,

then we compute σt inside out :

$\sigma x = A$

$\sigma f(x) = G(x,x) \lceil x/A \rceil = G(A,A)$

$\sigma t = \sigma f(f(x)) = G(x,x) \lceil x/G(A,A) \rceil = G(G(A,A),G(A,A))$.

2.3. Program constructs expressed by second-order patterns

A program pattern is going to be a second-order term of the language T defined in 2.2. Let us see some specific uses of this notation.

a) program composition

We can readily express "black boxes" of a flowchart as functions variables of the corresponding schema, and composition of these boxes corresponds to application. For instance, the pattern f(A(g(B)),C) where A,B and C are constants, f and g are function variables, matches the program *while* C *do* A(B) according to (among others) the substitution

$\{<f,\lambda uv \cdot$ *while* v *do* $u>,<g,\lambda u \cdot u>\}$.

b) contexts

If we want to express as a pattern a condition such as : " a while statement containing in its body an assignment to variable x", then the term *while* p *do* f(x:=e) will not do. We want to restrict the terms substituted to

variable f to be of the form $\lambda u \cdot t$, where t contains one and only one occurrence of u. We shall introduce a new kind of variable, called a *context variable*, and use square brackets rather than parentheses : *while* p *do* f[x:=e]. This expresses that variable f is restricted as indicated above. The notation generalizes immediately to contexts with several arguments.

c) scope restrictions

Block structure and scope restrictions can be expressed with the corresponding variable abstraction. For instance, we shall represent a system of recursive equations of the form :

$$\begin{cases} f_1(x_1^1,\cdots,x_{n_1}^1) \Leftarrow t_1 \\ \cdots \\ f_p(x_1^p,\cdots,x_{n_p}^p) \Leftarrow t_p \end{cases}$$

by the term :

$Y(\lambda f_1 \cdots f_p \cdot <\lambda x_1^1 \cdots x_n^1 \cdot t_1,\cdots,\lambda x_1^p \cdots x_{n_p}^p \cdot t_p>)$, where Y and < > are two constants of the appropriate type, denoting respectively the fixpoint operator and cartesian product Y and < > are generic constants, whose types are determined by the context. In the following, we shall use also $\perp$ and = as generic constants.

d) segment variables for lists and associative operators

Let us indicate briefly how the associativity of the composition of monadic functions may be used to model associativity of certain operators, or to build in lists.

Given a type τ, we can add a new elementary type $\bar{\tau}$, and a new constant C^τ of type $(\tau x \bar{\tau} \to \bar{\tau})$. We now represent a list $a_1,\cdots,a_n$ of objects of type τ as the term of type $(\bar{\tau} \to \bar{\tau})$: $\lambda u \cdot C^\tau(a_1,C^\tau(a_2,\cdots,C^\tau(a_n,u)\cdots))$.

Note that appending lists is denoted by functional composition. We can now represent the set of lists containing the constant A as the pattern $<\mathcal{l}_1 \cdots A \cdots \mathcal{l}_2>$, where $\mathcal{l}_1$ and $\mathcal{l}_2$ are (segment) variables of type $(\bar{\tau} \to \bar{\tau})$. This pattern is an abbreviation for the term :

$$\lambda u \cdot \mathcal{l}_1(C^\tau(A,\mathcal{l}_2(u))).$$

This pattern matches the list <A,A,B> in two ways :

$$\sigma_1 \begin{cases} <\mathcal{l}_1, \lambda u \cdot u> & \text{(the empty list< >)} \\ <\mathcal{l}_2, \lambda u \cdot C^\tau(A,C^\tau(B,u))> & \text{(the list <A,B>)} \end{cases}$$

$$\sigma_2 \begin{cases} <\mathcal{l}_1, \lambda u \cdot C^\tau(A,u)> & \text{(the singleton list <A>)} \\ <\mathcal{l}_2, \lambda u \cdot C^\tau(B,u)> & \text{(the singleton list <B>).} \end{cases}$$

e) Finally, program features such asarrays and data structures are basically functional in nature, and indexing (respectively field accessing) may be represented by function application. But the authors have not used so far this idea for representing transformations of programs manipulating structured data.

2.4. A complete matching algorithm for second-order terms

a) The match preorder ≤

Let t,t' be two terms of the same type. We say that t *matches* t', and we note t≤t', iff there exists a substitution σ such that t' = σt.

Let us define in the usual way the composition of two substitutions σ and σ' :

$$\sigma'\circ\sigma = \{\langle x,\sigma' t\rangle \mid \langle x,t\rangle\in\sigma\} \cup \{\langle x,t\rangle\in\sigma' \mid x\notin\mathcal{D}(\sigma)\}.$$

We can extend the preorder ≤ to substitutions by defining :

$$\sigma\leq\sigma' \quad \text{iff} \quad \exists\rho \quad \sigma' = \rho\circ\sigma.$$

Properties of this preorder for various logical languages are given in [8].

For t,t' terms of the same type, we define :

S is a *complete set of minimal matches* of <t,t'> iff :

a)	$\forall\sigma\in S \quad t' = \sigma t$	consistence
b)	$\forall\sigma' \quad t' = \sigma' t \Longrightarrow \exists\,\sigma\in S \quad \sigma\leq\sigma'$	completeness
c)	$\forall\sigma,\sigma'\in S \quad \sigma\neq\sigma' \Longrightarrow \sigma\not\leq\sigma'$	minimality

Example : Let t = f(x), t' = A. Then $\{\sigma_1,\sigma_2\}$ is a complete set of minimal matches of <t,t'>, with :

$$\begin{cases}\sigma_1 = \{\langle f,\lambda u\cdot A\rangle\}\\ \sigma_2 = \{\langle f,\lambda u\cdot u\rangle,\langle x,A\rangle\}\end{cases}$$

Some other matches :

$$\sigma_3 = \{\langle f,\lambda u\cdot A\rangle,\langle x,B\rangle\}$$
$$\sigma_4 = \{\langle f,\lambda u\cdot A\rangle,\langle y,B\rangle\}.$$

b) Matching trees

We shall now show how to construct a complete set S of minimal matches of terms t and t'. The method is an adaptation to second-order of a unification

algorithm for type theory described in [9].

We define trees, called matching trees, whose nodes are finite sets of pairs of terms $<t,t'>$ of the same type, or the special tokens Ⓢ for success and Ⓕ for failure.

We first define a procedure SIMPL(N)that simplifies a node N, by recognizing common constant initial subterms :

SIMPL(N)

Loop

If $N = \emptyset$ then replace N by Ⓢ and exit.

If $N = \bar{N} \cup \{<\lambda x_1 \cdots x_n \cdot \phi(t_1,\cdots,t_p), \lambda x'_1 \cdots x'_n \cdot \phi'(t'_1,\cdots,t'_p,)>\}$

with $\phi \in C \cup \{x_1,\cdots,x_n\}$ then :

i) if $\phi' = \begin{cases} \phi & \text{if } \phi \in C \\ x'_i & \text{if } \phi = x_i \end{cases}$ then

$N \leftarrow \bar{N} \cup \{<\lambda x_1 \cdots x_n \cdot t_i, \lambda x'_1 \cdots x'_n \cdot t'_i> \mid 1 \le i \le p\}$

ii) otherwise replace N by Ⓕ and exit.

Repeat

End of SIMPL.

Matching trees for $<t,t'>$ are grown as follows.

1) The root node is N_0 = SIMPL($\{<t,t'>\}$). We shall assume that N_0 is such that all constants appearing in it are of a type of order at most 3. That means that we authorize constants of order higher than 4 in t and t' only in a common outer context. This will be the case for our schemas, where the fourth-order constant Y appears only at the top-level.

2) Nodes Ⓢ and Ⓕ are leaves.

3) To grow the successors of node N, select in N an arbitrary pair : $<t_1,t_2>$. The function MATCH$<t_1,t_2>$ given below returns a finite set of substitution pairs $\sigma_1,\cdots,\sigma_k$. For each of these pairs σ_i, grow an arc labeled with σ_i from N to its successor SIMPL(σ_iN). (If k=0, replace N by Ⓕ).

Let us now define the algorithm MATCH :

MATCH$<t_1,t_2>$.

There are two cases, according to the order of the head variable of t_1 :

1. Order 1 : $t_1 = \lambda u_1 \cdots u_n \cdot x \quad \tau(x) \in T_0$.

Let $t_2 = \lambda v_1 \cdots v_n \cdot t_2'$. Then :

- if one of the v's appear free in t_2', then k=0, no solutions
- otherwise k=1, return the unique solution $<x,t_2'>$.

2. Order 2 : $t_1 = \lambda u_1 \cdots u_n \cdot f(t_1^1,\cdots,t_1^p)$

$t_2 = \lambda v_1 \cdots v_n \cdot \phi(t_2^1,\cdots,t_2^q)$.

Remark that $\phi \in C \cup \{v_1,\cdots,v_n\}$.

We generate $k \leq p+1$ solutions as follows :

a) in the case $\phi \in C$, one *imitation* : $<f,\lambda x_1 \cdots x_p \cdot \phi(\hat{t}_1,\cdots,\hat{t}_q)>$ where $\hat{t}_i$ is constructed as :

- if $\tau(t_2^i) \in T_0$ then $\hat{t}_i = h_i(x_1,\cdots,x_p)$
- if $\tau(t_2^i) = (\alpha_1,\cdots,\alpha_s \to \beta)$ then $\hat{t}_i = \lambda w_1 \cdots w_s \cdot h_i(x_1,\cdots,x_p,w_1,\cdots,w_s)$ with $\tau(w_j) = \alpha_j \quad 1 \leq j \leq s$

where the h_j's are new distinct variables of the appropriate type. Note that since ϕ is of order at most 3, t_2^i is of order at most 2, whence $\alpha_j \in T_0$, and therefore the h_i's are second-order variables.

b) all the *projections* $<f,\lambda x_1 \cdots x_p \cdot x_j>$ that are type compatible, i.e. such that $\tau(f) = (\gamma_1 \times \cdots \times \gamma_p \to \delta)$ with $\delta = \gamma_j$.

end of MATCH.

Lemma : For all t and t', the construction of a matching tree always terminates.

Proof : By double induction on $\lambda + \omega\lambda'$, where

$$\lambda = \Sigma\{|t_1| \mid <t_1,t_2> \in N\}$$

and $$\lambda' = \Sigma\{|t_2| \mid <t_1,t_2> \in N\},$$

with $\begin{cases} |\lambda u_1 \cdots u_n \cdot \phi(t_1,\cdots,t_p)| = 1 + \sum_{i=1} |t_i| \\ |x| = 1. \end{cases}$

c) *Correctness of the method*

Definition :

Let MT be a matching tree constructed for t and t' as defined above. We call *set of solutions* of MT the set S(MT), of all substitutions $\sigma = \sigma_n o \sigma_{n-1} o \cdots o \sigma_1$ such that $\sigma_1, \sigma_2, \cdots, \sigma_n$ label a branch of MT terminated in(S). The domain of σ is restricted to those variables appearing free in t.

Theorem : For all t and t' terms of the same type built on variables of order at most 2 and constants of order at most 3 (except possibly in a common outer context), for all MT matching trees for t and t', S(MT) is a complete set of minimal matches of t and t'.

Proof : Given in Huet [8], where various examples of matching trees are given.

Note that this proves the existence of such complete sets, which can be shown to be unique up to an isomorphism.

Example : Let us go back to our example template, where Σ is represented by the term :

$$t = \lambda u \cdot Y(\lambda f\, x \cdot C(a(x), b(x), h(d(x), f(e(x)))), u).$$

Suppose we are trying to recognize as an instance of Σ the "reverse" program, represented by the term :

$$t' = \lambda u \cdot Y(\lambda\ rev\ x \cdot C(Null(x), Nil, Append(rev(Cdr(x)), Cons(Car(x), Nil)), u).$$

Any matching tree constructed for t and t' will give us three solutions :

$\rho \cup \sigma_1$, $\rho \cup \sigma_2$ and $\rho \cup \sigma_3$ where

$$\rho = \{<a, \lambda u \cdot Null(u)>, <h, \lambda u \cdot Nil>, <e, \lambda u \cdot Cdr(u)>\}$$

and

$$\sigma_1 = \{<h, \lambda xy \cdot Append(y, x)>, <d, \lambda u \cdot Cons(Car(u), Nil)>\}$$

$$\sigma_2 = \{<h, \lambda xy \cdot Append(y, Cons(x, Nil))>, <d, \lambda u \cdot Car(u)>\}$$

$$\sigma_3 = \{<h, \lambda xy \cdot Append(y, Cons(Car(x), Nil))>, <d, \lambda u \cdot u>\}$$

Of these, only $\rho \cup \sigma_1$ satisfies X_1 ; this is precisely the match we considered in section 2.1.

Remarks :

In the algorithm above we have supposed that the second order variables were unrestricted. For instance if we suppose f to be a *context* variable,

then we accept σ as a solution only if $f = \lambda x \cdot t$, where x appears free in t in just one occurrence.

The algorithm given here is general and rather efficient. However one should formulate the schemas with care to as not to get redundant solutions, for instance by using unnecessary compositions of free variables.

In certain special cases the algorithm can be speeded up. For instance, in the case of monadic functions, it may be a good idea to first check length conditions.

2.5. Applying transformation templates

Let $\langle \Sigma, \Sigma', X \rangle$ by a transformation template, and let $P[t]$ be a program which contains t as a subterm. We say the template is *applicable* in P at t iff there exists asubstitution σ for the free variables of Σ and Σ' such that :

1) $t = \sigma\Sigma$

2) $M \models \sigma X$ i.e. axioms X instantiated by σ are valid in the programming language semantics M(we shall make this more precise in the next section). When these two conditions are met, we say we apply the template in P at t by replacing in P the subterm t by the term $\sigma\Sigma'$: $P[\sigma\Sigma']$.

Let $V(\Sigma)$ be the set of free variables appearing in Σ, same for $V(\Sigma')$ and $V(X)$. We have of course $V(X) \subset V(\Sigma) \cup V(\Sigma')$.

The general method for applying a template to a program part proceeds as follows :

a) Apply the matching algorithm given in the previous section to terms Σ and t. This gives a finite set of candidates of σ. However such σ's only fix the values of the variables in Σ.

b) Complete σ to the variables in $V(X)-V(\Sigma)$

c) Prove σX as explained in the next section.

d) If this succeeds, replace in the program t by $\sigma\Sigma'$.

In simple examples of transformation templates such as our standard example, step b is missing since $V(X) \subset V(\Sigma)$. However in more complicated examples step b can be a bit tricky. This problem is in fact very similar to program synthesis. Let us give an example of the kind of difficulty involved.

One of the templates given in the apperdix is the following :

$$\Sigma_2 = f(x) \Longleftarrow \textit{if } a(x) \textit{ then } b \textit{ else } h(x,f(e(x)))$$

$$\Sigma_2' = \begin{cases} f'(x) \Longleftarrow g'(x,b) \\ g'(x,y) \Longleftarrow \textit{if } a(x) \textit{ then } y \textit{ else } g'(e(x),h'(y,x)) \end{cases}$$

$$X_2 \begin{cases} \forall x\ h(x,b) = h'(b,x) \\ \forall xyz\ h(x,h'(y,z)) = h'(h(x,y),z) \\ \forall x\ h(x,\perp) = \perp \end{cases}$$

It is possible for instance to recognize as an instance of Σ_2 the reverse program :

$rev(x) \Longleftarrow \textit{if } Null(x) \textit{ then } Nil \textit{ else } Append(rev(Cdr(x)),Cons(Car(x),Nil))$,

by using the match :

$$\sigma \begin{cases} h' \leftarrow \lambda xy \cdot Append(y,Cons(Car(x),Nil)) \\ a \leftarrow \lambda x \cdot Null(x) \\ b \leftarrow Nil \\ e \leftarrow \lambda x \cdot Cdr(x) \end{cases}$$

But now one must find the match for h' (which does not appear in Σ_2) that will validate X_2. One way of doing that is to use σX_2 as match equations, using our algorithm above. For instance, using the first axiom, one wants to match $\lambda x \cdot h'(Nil,x)$ with $\lambda x \cdot Append(Nil,Cons(Car(x),Nil))$. One of the matches is

$$h' \leftarrow \lambda xy \cdot Append(Nil,Cons(Car(y),x)).$$

It is now easy to prove the other axioms, knowing the recursive definition of Append :

a) $Append(Nil,x) = x$

b) $Append(Cons(x,y),z) = Cons(x,Append(y,z))$

and the fact that Append is strict in its second argument.

A better way would be first to apply a) as a simplification on $X_{2.1}$, then match h' in $X_{2.1}$ by: $h' \leftarrow \lambda uv \cdot Cons(Car(y),x)$, then validate $X_{2.2}$ as an instance of b).

One can then rewrite Σ' in an iterative reverse program :

$rev(x) \Longleftarrow revl(x,Nil)$

where $revl(x,y) \Longleftarrow \textit{if } Null(x) \textit{ then } y \textit{ else } revl(Cdr(x),Cons(Car(x),y))$.

3. Proving transformation templates

3.1. Semantics of program schemes

In this section, we shall now regard a program pattern as a program *schema*, denoting the family of programs obtained by fixing the interpretation of the atoms appearing in it.

Our general formalism to express the semantics of such program schemas is usually known as denotional semantics (as opposed to operational semantics, which describes the deductive aspects of the programming language considered as a formal system). The foundations of this method were first investigated by the Scott-Strachey group. It is now generally recognized that this type of semantic definition is the most rigorous framework in which one can prove formally properties of programs (including termination). The formal system known as LCF [19] (Logic for Computable Functions) is an implementation of this method for typed languages, which is directly suitable for our purpose. We shall here assume an underlying proof system similar to LCF.

Let T be the second-order term language over V and C, as defined in section 2, in which we represent programs and schemas. An *interpretation* domain $\mathcal{D}$ of T is determined by a set of complete partial orders $\mathcal{D}_\alpha$, one for each elementary type α in T_0. For $\alpha = (\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \beta)$, we define $\mathcal{D}_\alpha$ as the set of continuous functions in $\mathcal{D}_{\alpha_1} \times \mathcal{D}_{\alpha_2} \times \cdots \times \mathcal{D}_{\alpha_n} \to \mathcal{D}_\beta$.

We then define an *interpretation* over $\mathcal{D}$ as a mapping $I : A \to \bigcup_{\alpha \in T} \mathcal{D}_\alpha$ preserving types : $I(\phi) \in \mathcal{D}_{\tau(\phi)}$. The interpretation I is then extended to the set T of terms as a morphism. For more details, see Milner [20].

A *programming language* over T is defined by a set M, (the models of the language) of interpretations. We shall here assume that M is the set of interpretations that satisfy a set of LCF axioms.

In practice, these axioms express the meaning of the control structure part of the programming language, and things such as "$\perp_\alpha$ is the least element of $\mathcal{D}_\alpha$", and so on. If p is an LCF statement, then we use $M \models p$ to express the fact that p is derivable in LCF from the programming language axioms. For instance, $M \models P_1 = P_2$ means that program schemes P_1 and P_2 are strongly equivalent

in any interpretation in the class M. For more information about constructing programming language axioms, see the excellent survey by Tennent [26].

Let $<\Sigma,\Sigma',X>$ be a transformation template. We say this template is *valid* if $M,X \models \Sigma=\Sigma'$ i.e. iff in every interpretation in M that satisfies all the axioms in X, Σ and Σ' denote the same function.

If a template is valid, then the correctness of program transformations obtained with it is easy to show :

Let $P[t]$ be a program containing as a subterm a term t maching Σ : $t = \sigma\Sigma$, with σ such that $M \models \sigma X$. Let $M' = \{I \circ \sigma \mid I \in M\}$. We have $M' \subset M$, since the programming language axioms are supposed to be closed, i.e. do not pertain to the free variables in the templates. Thus $M' \models X$, and by validity of the template :

$$M' \models \Sigma = \Sigma', \text{ i.e. } M \models \sigma\Sigma = \sigma\Sigma'.$$

If we now assume that our semantic equivalence = is a congruence relatively to the programming construct, i.e. if

$$M \models t = t' \supset P[t] = P[t'], \quad (*)$$

which is usually the case, then we may infer

$$M = P[t] = P[t'], \text{ with } t' = \sigma\Sigma' ;$$

i.e. our schematic transformation has preserved strong equivalence.

This finishes the formal justification of the method. Our plan now is to prove valid every transformation template before adding it to our library of schematic transformations. These proofs can be carried out mechanically, for instance on the LCF system implemented in Edinburgh [33]. However, it should be stressed that this is not a trivial task : these proofs can be fairly tedious, depending on the peculiarities of the programming language. In the next section, we shall show the complete proof of the validity of the template given in section 1.

For the simplicity of the exposition, we shall not prove directly the equivalence of Σ and Σ', since in order to do that we should axiomatize the iterative constructs used in Σ'. Rather, we shall prove the equivalence of Σ and the recursive schema Σ'' obtainded by applying McCarthy's transform on Σ' [17].

3.2. A worked example

We now attempt to prove the validity of the template $\langle \Sigma, \Sigma'', X \rangle$ where :

$$\Sigma : f(x) \Leftarrow \textit{if } a(x) \textit{ then } h(x) \textit{ else } h(d(x), f(e(x))).$$

and Σ'' is the recursive schema obtained from the iterative Σ' given in section 1 by application of McCarthy's transformation [17] :

$$\Sigma'' : \begin{cases} f''(x) \Leftarrow \textit{if } a(x) \textit{ then } b(x) \textit{ else } g''(e(x), d(x)) \\ g''(x,y) \Leftarrow \textit{if } a(x) \textit{ then } h(y, b(x)) \textit{ else } g''(e(x), h(y, d(x))) \end{cases}$$

The set of axioms X contains the associativity axiom for h, as expected, but also an axiom stating that h is strict in its second argument :

$$X \begin{cases} X_a : \forall x,y,z \; h(x,h(y,z)) = h(h(x,y),z) \\ X_b : \forall x \; h(x,\perp) = \perp \end{cases}$$

(As noted before, $\perp$ is a constant always interpreted as the least element of the appropriate domain D).

We now prove that $M, X \models \Sigma = \Sigma''$, assuming for M the usual fixpoint semantics of recursive equations [16]. Let us recall two properties that will be needed in the proof :

(1) (parallel) computation induction rule

Let P be a proposition, ϕ_i be a variable of type τ_i, $F_i = \lambda f_i \cdot t_i$ be a term of type $(\tau_i \to \tau_i)$, $\perp_i$ be the least element of type τ_i, and Y_i be the fixpoint operator of type $((\tau_i \to \tau_i) \to \tau_i)$, for $i \leq n$.

We let : $\sigma_i = \{\langle f_i, \phi_i \rangle\}$, $\sigma_\perp = \{\langle f_i, \perp_i \rangle \mid i \leq n\}$,
$\sigma_F = \{\langle f_i, \sigma_i t_i \rangle \mid i \leq n\}$ and $\sigma_Y = \{\langle f_i, Y_i(F_i) \rangle \mid i \leq n\}$.

Our induction rule is :

$$\frac{\sigma_\perp P \;,\; P \supset \sigma_F P}{\sigma_Y P}$$

For the conditions of validity of this rule (P must be an admissible proposition) see [16]. In the following we use the rule with n=2. Recall that we use $\perp$ and Y as generic constants, whose types are determined by the context.

(2) The "if axiom" $\forall \phi \in A$, $\forall n$, $\forall i \leq n$ $\forall t, t_1, \cdots, t_n, t_i'$ of the appropriate types $t = \perp \supset \phi(t_1, \cdots, t_{i-1}, \perp, t_{i+1}, \cdots, t_n) = \perp \quad \vdash$

$$\phi(t_1, \cdots, t_{i-1}, \textit{if } t \textit{ then } t_i \textit{ else } t_i', t_{i+1}, \cdots, t_n) =$$
$$= \textit{if } t \textit{ then } \phi(\cdots, t_i, \cdots) \textit{ else } \phi(\cdots, t_i', \cdots).$$

Let $f = Y(F)$, where

$F = \lambda f\, x \cdot$ *if* $a(x)$ *then* $b(x)$ *else* $h(d(x), f(e(x)))$,

confusing the variable f with the name of the smallest fixpoint $Y(F)$.
Similarly $g'' = Y(G'')$, with

$G'' = \lambda g''\, x\, y \cdot$ *if* $a(x)$ *then* $h(y, b(x))$ *else* $g''(e(x), h(y, d(x)))$

and we introduce a new function $g = Y(G)$, with

$G = \lambda g\, x\, y \cdot$ *if* $a(x)$ *then* $h(y, b(x))$ *else* $h(y, g(e(x), d(x)))$.

Lemma 1 : $\forall x,y \quad h(x, f(y)) = g(y,x)$

Proof : We use computation induction in parallel on f and g.

- base step : $h(x, \perp) = \perp$ using X_b
- induction step :

  Assuming the lemma true for ϕ and Ψ, we prove it for $F(\phi)$ and $G(\Psi)$.

  $h(x, F(\phi)(y)) = h(x,$ *if* $a(y)$ *then* $b(y)$ *else* $h(d(y), \phi(e(y))))$

  $=$ *if* $a(y)$ *then* $h(x, b(y))$ *else* $h(x, h(d(y), \phi(e(y))))$
  using the "if axiom" and X_b

  $=$ *if* $a(y)$ *then* $h(x, b(y))$ *else* $h(x, \Psi(e(y), d(y)))$
  using the induction hypothesis

  $= G(\Psi)(y,x)$. □

Lemma 2 : $\forall x,y,z \quad h(x, g(y,z)) = g''(y, h(x,z))$

Proof : We use again computation induction on g and g''

- base step : $h(x, \perp) = \perp$ using X_b.
- induction step : we assume the lemma true for ϕ and ϕ''.

  $h(x, G(\phi)(y,z)) = h(x,$ *if* $a(y)$ *then* $h(z, b(y))$ *else* $h(z, \phi(e(y), d(y))))$

  $=$ *if* $a(y)$ *then* $h(x, h(z, b(y)))$ *else* $h(x, h(z, \phi(e(y), d(y))))$
  using the "if axiom" and X_b.

  $=$ *if* $a(y)$ *then* $h(h(x,z), b(y))$ *else* $h(h(x,z), \phi(e(y), d(y)))$
  using X_a twice.

  $=$ *if* $a(y)$ *then* $h(h(x,z), b(y))$ *else* $\phi''(e(y), h(h(x,z), d(y)))$
  using the induction hypothesis.

  $= G''(\phi'')(y, h(x,z))$. □

Lemma 3 : $\forall x,y \quad g(x,y) = g''(x,y)$

Proof : $g(x,y) = \mathit{if}\ a(x)\ \mathit{then}\ h(y,b(x))\ \mathit{else}\ h(y,g(e(x),d(x))$

$= \mathit{if}\ a(x)\ \mathit{then}\ h(y,b(x))\ \mathit{else}\ g''(e(x),h(y,d(x))$

using lemma 2.

$= g''(x,y)$. □

Lemma 4 : $\forall x \quad f(x) = f''(x)$

Proof : $f(x) = \mathit{if}\ a(x)\ \mathit{then}\ b(x)\ \mathit{else}\ h(d(x),f(e(x)))$

$= \mathit{if}\ a(x)\ \mathit{then}\ b(x)\ \mathit{else}\ g(e(x),d(x))$

using lemma 1.

$= \mathit{if}\ a(x)\ \mathit{then}\ b(x)\ \mathit{else}\ g''(e(x),d(x))$

using lemma 3.

$= f''(x)$. □

Remark : validity of Mc Carthy's transformation

We have now proven that $\Sigma=\Sigma''$. The validity of the template $<\Sigma,\Sigma',X>$ will thus be established, provided Mc Carthy's transformation preserves strong correctness. Unfortunately this is not quite the case, since we assumed for our recursive equations a least fixpoint semantics, whereas the usual proof of the validity of Mc Carthy's transform [25] uses an operational semantics (innermost evaluation of recursive call) which may lead to functions less defined than by using the least-fixpoint semantics.

For instance, consider the following interpretation :

$$\begin{cases} h = \lambda x\ y \cdot y \\ a(X) = \text{false} \\ a(e(X)) = \text{true} \\ d(X) = \bot \end{cases} \qquad \text{for some value X}$$

then $f(X) = b(e(X)) = f''(X)$

whereas $f'(X) = \bot$ with f' computed by the iterative Σ' in section 1.

Thus, with the above method, we would only prove that $\Sigma' \sqsubseteq \Sigma'' = \Sigma$, and therefore $\Sigma' \sqsubseteq \Sigma$. (Where $\sqsubseteq$ is the "less defined than" partial ordering). One solution to this problem is to add new validity conditions, such as $\forall x\ d(x) \neq \bot$ in the above example, in order to preserve termination. Another solution would be to define the semantics of iterative and recursive schemes so that McCarthy's

transformation is strongly correct. We shall not do it here for lack of space. Some conditions on those semantics that guarantee strong correctness of the transformation are investigated in [14].

This discussion emphasizes the necessity of presenting the semantic definition of the language used, and the validation proof, together with any proposed transformation template.

3.3. Finding and checking transformation templates

Even for the simple transformation template validated in the previous section, the proof is rather long. Furthermore, although simple enough to prove, the intermediate lemmas are not always easy to find. Thus we have found convenient to have some other techniques to check ("debug") transformation templates before attempting to prove formally their validity. It appears from our experience that these techniques might be developped to help discovering new templates.

The first is the "fold-unfold" method described in [1]. Let us apply it to check the template $\langle \Sigma, \Sigma'', X \rangle$.

We have :

$f(x) \Leftarrow$ *if* a(x) *then* b(x) *else* h(d(x), f(e(x)))

and we define ("eureka"!) :

$g''(x,y) \Leftarrow h(y,f(x))$

So, by unfolding :

$g''(x,y) \Leftarrow h(y,$ *if* a(x) *then* b(x) *else* h(d(x),f(e(x))))

$\Leftarrow$ *if* a(x) *then* h(y,h(x)) *else* h(y,h(d(x),f(e(x))))

by the "if axiom" and X_h.

$\Leftarrow$ *if* a(x) *then* h(y,h(x)) *else* h(h(y,d(x)),f(e(x)))

by axiom X_a

$\Leftarrow$ *if* a(x) *then* h(y,h(x)) *else* g''(e(x),h(y,d(x)))

by folding.

We define :

$f''(x) \Leftarrow$ *if* a(x) *then* b(x) *else* g''(e(x),d(x))

$\Leftarrow$ *if* a(x) *then* b(x) *else* h(d(x),f(e(x)))

by unfolding of g''

$\Leftarrow$ f(x) by folding of f.

If this checking is carried out with computer assistance, the matching algorithm used to apply transformations to programs may be used here to perform the fold operation.

Note that this type of program manipulation, possibly guided by an analysis of the program [30], can be used to discover new transformations.

Similar program manipulations (e.g. loop unraveling) are possible for iterative languages.

The second method, best used with computer assistance, is to symbolically evaluate the two program patterns of the transformation template. Of course, equivalent arbitrary (but consistent) assumptions on the value of boolean expressions must be made for both patterns when conditionals are encountered. Then we can use the axioms of the transformation template to check the equivalence of the two symbolic expressions computed.

Applying this to our example, let us assume that for some integer n (to be chosen) we have

$$\forall i<n \quad ae^i(x) = \text{false} \quad \text{and} \quad ae^n(x) = \text{true}$$

The term computed by Σ is then

$$E = h(d(x), h(de(x), \cdots h(de^{n-1}(x), be^n(x))\cdots))$$

while Σ' or Σ'' computes :

$$E' = h(h(\cdots h(d(x), de(x))\cdots, de^{n-1}(x)), be^n(x))$$

It is easy to see that E and E' are equivalent, provided that h is associative.

Again it should be possible to use this approach to guide the discovery of new transformations, for example along the line outlined for factorial in §1.

A close examination of the terms computed symbolically by two program schemes may give clues about the kind of induction necessary to prove their equivalence. This approach is close to the theorem proving technique used by Boyer and Moore in [0].

We have presented here 3 various methods to check and ultimately formally prove the validity of a transformation template $<\Sigma,\Sigma',X>$. Applying such a template in a program context $P[\ ]$ has been explained in 2.5. However we stress that the validity of the transformation requires our semantics to be congruent with respect to program contexts (condition (*) in 3.1). When this condition is not met, context preconditions must be added to the templates. We shall not further develop this idea in the present paper.

Let us now turn to the problem of organizing together a set of transformation templates.

4. Organizing transformation templates

4.1 Matching several templates simultaneously

Let us suppose that two templates $<\Sigma_1,\Sigma_1',X_1>$ and $<\Sigma_2,\Sigma_2',X_2>$ are such that Σ_1 and Σ_2 are similar. It may be a good idea to try to factorize the common part of Σ_1 and Σ_2 into a term Σ, and to search first for occurrences of Σ.

This problem has a well-known solution in first-order logic : Σ is the least generalisation (i.e. glb for the preorder $\leq$) of the terms Σ_1 and Σ_2. Unfortunately, such a glb does not always exist in second-order logic, even for monadic terms, as shown in [8]. We would have therefore to content ourselves with some lower bound of Σ_1 and Σ_2 (in general of a set of Σ_i's).

Another problem occurs when in a certain context two distinct transformations are available, leading to two different programs. Then either the user of the system is asked to validate transformations one by one, and he chooses the one is he prefers, or some preference ordering is used.

More generally, the application of a transformation at some point may prevent some other transformation in an inside or outside context. The effect of a set of transformation templates may therefore differ whether one adopts an outside-in or an inside-out search. These are well known problems for rewriting systems. In some cases it will be possible to prove nice properties of our sets of templates such as confluence (or Church-Rosser property). Non-confluent systems may be made confluent by trying to apply closure algorithms, along the lines of the Knuth-Bendix superposition algorithm. Although

these problems have been well studied for first order logic [11,34], more research is needed for second-order logic.

Finally, let us indicate that the application of transformation templates may be considerably speeded up if one supplies heuristics to the system concerning plausible rules to try when one rule has succeeded. This has been suggested in particular by Loveman [15].

We shall in the next section deal with the problems of adding a new template to a set of templates, and checking whether a template subsumes another one.

4.2. Template subsumption

a) Extension of the match preorder to templates

Let $T_i = \langle \Sigma_i, \Sigma_i', X_i \rangle$ with i=1,2 be two transformation templates.

We say that T_1 *subsumes* T_2, and note $T_1 \leq T_2$ iff there exists a substitution σ such that :

i) $\Sigma_2 = \sigma\Sigma_1$

ii) $\Sigma_2' = \sigma\Sigma_1'$

iii) $M \models X_2 \supset \sigma X_1$

We will note $T_1 \leqq_{\sigma} T_2$, when we desire to specify σ.

The match relation "$\leq$" is now also a preorder on the set of templates.

Let t be a term and T_i a transformation template we note $T_i(t)$ the set of terms that may be obtained by applying T_i to t. There may be more than one translation when the first pattern of T_i matches t in more than one way.

b) Proposition

Let $T_i = \langle \Sigma_i, \Sigma_i', X_i \rangle$ with i=1,2 be two templates and t be a term :

$$T_1 \leq T_2 \text{ implies } T_2(t) \subseteq T_1(t)$$

Proof :

Let τ be a substitution such that $T_1 \leqq_{\tau} T_2$ (there may be more than one).

Let t' be in $T_2(t)$. There is a substitution σ_2 such that :

$$\begin{cases} t = \sigma_2\Sigma_2 \\ t' = \sigma_2\Sigma_2' \\ M \models \sigma_2 X_2 \end{cases}$$

Define $\sigma_1 = \sigma_2 o \tau$; since $T_1 \leqslant_\tau T_2$ we have also $\begin{cases} t = \sigma_1 \Sigma_1 \\ t' = \sigma_1 \Sigma'_1 \end{cases}$

Let $M' = \{ I o \sigma_2 \mid I \in M \}$

$M' \models X_2$, but since $M' \subset M$ (see §3.1)

$M' \models X_2 \supset \tau X_1$

so $M' \models \tau X_1$, i.e. $M \models (\sigma_2 o \tau) X_1$ or $M \models \sigma_1 X_1$

Therefore $t' \in T_1(t)$. □

c) Application to template library organization

We can apply the proposition above to the cleaning-up of a template library. If the library contains T_1 and T_2 such that $T_1 \leq T_2$, we can eliminate T_2 because it does not add any extra transformational power.

Such redundancy is easily detected in practice by applying to T_2, and with the library minus T_2, the same matching algorithm that we use to transform programs.

In fact, since transformations can often be composed (i.e. applied in succession), we can detect with this method the subsumption of a transformation by the composition of some others.

As an example, we will establish that the template $T_{3.3}$ is redundant with another one $T_{1.2}$ (the indexes refers to the template organization in [2]).

$T_{1.2} = \langle \Sigma_{1.2}, \Sigma'_{1.2}, X_{1.2} \rangle$ where

$\Sigma_{1.2}$: f(x) $\Leftarrow$ *if* a(x) *then* b *else* h(d(x),f(e(x)))

$\Sigma'_{1.2}$:

```
function f(x) ;
        begin
        f:=b ;
        while not a(x) do
            begin
            f:=h(d(x),f) ;
            x:=e(x) ;
            end ;
        end ;
```

$$X_{1.2} : \begin{cases} X_{1.2,a} : & \forall x,y,z \quad h(x,h(y,z)) = h(y,h(x,z)) \\ X_{1.2,b} : & \forall x \quad h(x,\perp) = \perp \end{cases}$$

and $T_{3.3} = \langle \Sigma_{3.3}, \Sigma'_{3.3}, X_{3.3} \rangle$ where

$\Sigma_{3.3}$: $f(x) \Leftarrow$ *if* $a(x)$ *then* b *else* $h(f(d(x)),f(d(x)))$

$\Sigma'_{3.3}$:
```
function f(x) ;
    begin
    f:=b ;
    while not a(x) do
        begin
        f:= h(f,f) ;
        x:= d(x) ;
        end ;
    end ;
```

$$X_{3.3} : \{ X_{3.3.a} : \quad h(\perp,\perp) = \perp$$

We leave it to the reader to show that the substitution σ below is such that $T_{1.2} \overset{\leq}{\sigma} T_{3.3}$:

$$\sigma = \{\langle h, \lambda uv \cdot h(v,v) \rangle, \langle e, \lambda u \cdot d(u) \rangle\}$$

Therefore, $T_{3.3}$ can be removed from the template library. In fact, four templates out of the ten presented in [2] can be removed in the same way.

We should however emphasize that it is only because our matching is complete that we can eliminate redundant templates. Going back to the proof of the underlying proposition b), we see that it is only the completeness of the matching algorithm that guarantees that, if it finds the match $t = \sigma_2 \Sigma_2$, it can also find the match $t = (\sigma_2 \circ \tau)\Sigma_1$.

d) Application to template simplification

Let us consider again the template $T_2 = \langle \Sigma_2, \Sigma'_2, X_2 \rangle$ mentioned in 2.5. Actually, the template that was originally discovered was the slightly more

complicated $\hat{T}_2 = \langle \hat{\Sigma}_2, \hat{\Sigma}'_2, \hat{X}_2 \rangle$ where :

$$\hat{\Sigma}_2 : f(x) \Leftarrow \textit{if } a(x) \textit{ then } b \textit{ else } h(d(x), f(e(x)))$$

$$\hat{\Sigma}'_2 : \begin{cases} f'(x) \Leftarrow g'(x,b) \\ g'(x,y) \Leftarrow \textit{if } a(x) \textit{ then } b \textit{ else } g'(e(x), h'(y,d(x))) \end{cases}$$

$$\hat{X}_2 : \begin{cases} \hat{X}_{2,a} : \forall x \; h(x,b) = h'(b,x) \\ \hat{X}_{2,b} : \forall x,y,z \; h(x,h'(y,z)) = h'(h(x,y),z) \\ \hat{X}_{2,c} : \forall x \; h(x,\perp) = \perp \end{cases}$$

The reader can easily check that $T_2 \overset{\sigma}{\leq} \hat{T}_2$ and $\hat{T}_2 \overset{\tau}{\leq} T_2$ where $\sigma = \{\langle h, \lambda xy \cdot h(d(x),y) \rangle , \langle h', \lambda xy \cdot h'(x,d(y)) \rangle\}$

and $\tau = \{\langle d, \lambda x \cdot x \rangle\}$.

The two templates T_2 and $\hat{T}_2$ being equivalent as to their effect, we should choose on some other criterion the one we will keep. This criterion will be simplicity of application.

Let us try to apply T_2 and $\hat{T}_2$ to the same program defining the list reversal function :

rev(x)$\Leftarrow$*if* Null(x) *then* Nil *else* Append(rev(Cdr(x)),Cons(Car(x),Nil)).

A match with Σ_2 gives the unique substitution

$\sigma = \{\langle a, \lambda x \cdot Null(x) \rangle, \langle b, Nil \rangle, \langle e, \lambda x \cdot Cdr(x) \rangle,$
$\langle h, \lambda xy \cdot Append(y, Cons(Car(x), Nil)) \rangle\}$

which will eventually lead to a transformation (see 2.5).

A match with $\hat{\Sigma}_2$ gives three solutions (see 2.4) :

$\tau \cup \tau_1$, $\tau \cup \tau_2$ and $\tau \cup \tau_3$ where

$\tau = \{\langle a, \lambda x \cdot Null(x) \rangle, \langle b, Nil \rangle, \langle e, \lambda x \cdot Cdr(x) \rangle\}$

$\tau_1 = \{\langle h, \lambda xy \cdot Append(y,x) \rangle, \langle d, \lambda x \cdot Cons(Car(x), Nil) \rangle\}$

$\tau_2 = \{\langle h, \lambda xy \cdot Append(y, Cons(x, Nil)) \rangle, \langle d, \lambda x \cdot Car(x) \rangle\}$

$\tau_3 = \{\langle h, \lambda xy \cdot Append(y, Cons(Car(x), Nil)) \rangle, \langle d, \lambda x \cdot x \rangle\}$

All three matches satisfy $\hat{X}_2$, with a proper choice of h', and thus lead to a solution.

Since $T_2 \leq \hat{T}_2$, the solution has to be the same unique one given by T_2 for all three matches of $\hat{T}_2$.

Without going into details, we can note that the matching algorithm will have more computation to perform to find 3 substitutions instead of one. Furthermore, if our transformation algorithm tries all possible transformations rather than stopping after the first success, $\hat{T}_2$ will cause it to do three times as much work as T_2, for the same result.

In short, the (apparently) increased degree of freedom brought in the template by the extra free variable d will cause greater non determinism (more choices), thus more inefficiency, in the computations without giving better results.

Given two equivalent templates, it seems to be a good rule to keep in the template library the one having the smallest number of free variables, e.g. T_2 in our example.

It is worth noting that this situation arises fairly often since one way of generalizing an already validated template is by trying to introduce new free variables without increasing (too much) the axiomatic constraints.

4.3. Focusing on interesting parts of a program

In general, one will want to use program transformations in order to achieve specific goals, such as optimization of computing time.

A system for applying program transformations should therefore associate with the templates some indication on their effects on various cost parameters. It is also desirable to analyse the program under consideration to identify the critical parameters associated with a given program context. The system will then try to apply the templates that improve those parameters. For instance, it will try time-saving transformations to innermost loops, and space saving ones to the least used program segments.

We shall not deal with this kind of problems here, and we refer the interested reader to [12], [21], [29].

5. More general program transformations

It is possible to generalize the program transformation method presented above in several ways. First, it may be possible to enrich the schema language, and still have a complete match algorithm. However, it is known for instance, that it is not possible to find a complete finite set of matches. for arbitrary third-order terms [8].

Another interesting extension would be to use information given in the axioms to help the match algorithm. For instance, if we know that some operator is associative, we know how to do the match modulo associativity of this operator. Some specialized match algorithms exist already in pattern-matching oriented programming languages for artificial intelligence [31], [32].

More generally, one may imagine specialized match algorithms that recognize specific families of program constructs. This is the approach we are currently investigating in the MENTOR project at IRIA. The idea is to use as a programming language the command language of a structured editor [5], and to write specialized programs to match complex schemes, and in case of success to effect the corresponding transformation. For instance, we implemented with this technique the recursion removal technique that consists in removing terminal recursions (i.e. doing the inverse of McCarthy's transformation). As an illustration, we show the effect of this transformation on a (real) Pascal program :

```
procedure RECSAVE(P:SPTR);
    begin
    if P#nil then
        begin
        with P@ do
            begin
            SAUVEFILE0.FLD1:=5;
            case [illegible] of
                0:  begin
                    [illegible]
                       #;#;#;#
                    end %CASE S%;
                    SAVECOM(P);
                    end %CAS 0%;
                1:  begin
                    if P11=nil then # else #;
                    PUT(SAUVEFILE);
                    SAVECOM(P);
                    RECSAVE(P11);
                    end %CAS 1%;
```

```
                    2:    begin
                          if F21=nil then # else #;
                          if F22#nil then #;
                          PUT(SAUVEFILE);
                          SAVECOM(P);
                          RECSAVE(F21);
                          RECSAVE(F22);
                          end %CAS 2%;
                    3:    begin
                          if F31=nil then # else #;
                          if F32#nil then #;
                          if F33#nil then #;
                          PUT(SAUVEFILE);
                          SAVECOM(P);
                          RECSAVE(F31);
                          RECSAVE(F32);
                          RECSAVE(F33);
                          end %CAS 3%
               end %ARITY%
               end %WITH%
          end %P#NIL%;
     end %RECSAVE%
```

The symbols '#' are standard abbreviation signs in our system (the full procedure would be two pages long). After the transformation is performed here is the resulting program :

```
procedure RECSAVE(P:SPTR);
     label
          1;
     begin
     1: if P#nil then
          begin
          with P@ do
               begin
               SAUVEFILE@.FLD1:=S;
               case ARITY of
                    0:    begin
                            case S of
                               #;#;#;#
                          end %CASE S%;
                           SAVECOM(P);
                          end %CAS 0%;
                    1:    begin
                           if F11=nil then # else #;
                          PUT(SAUVEFILE);
                          SAVECOM(P);
                          P:=F11;
                          goto 1;
                          end %CAS 1%;
```

```
                2:    begin
                      if F21=nil then # else #;
                      if F22#nil then #;
                      PUT(SAUVEFILE);
                      SAVECOM(P);
                      RECSAVE(F21);
                      P:=F22;
                      goto 1;
                      end %CAS 2%;
                3:    begin
                      if F31=nil then # else #;
                      if F32#nil then #;
                      if F33#nil then #;
                      PUT(SAUVEFILE);
                      SAVECOM(P);
                      RECSAVE(F31);
                      RECSAVE(F32);
                      P:=F33;
                      goto 1;
                      end %CAS 3%
          end %ARITY%
          end %WITH%
     end %P#NIL%;
end %RECSAVE%
```

This technique is very general and more powerful than schematic rewriting. We believe that it may be used for instance to implement Cohen's recursion removal transformations [39]. However the formal proof of correctness of such transformations is a much harder problem than correctness of schematic rewritings.

Conclusion

We have presented in this paper a program transformation technique based on rewriting program schemas written in a 2^{nd} order term language. We have given a complete matching algorithm for this language and have shown how to formally validate such transformations, using denotational semantics.

This technique can be completely automated, and made part of the "program massagers" or "assistant programmers" that are now under development in various places [3], [15], [24]. The MENTOR project at IRIA [10] is part of this effort. These systems, in which knowledge about programming is organized in various ways, seem to be among the most promising applications of artificial intelligence techniques.

Appendix

A library of recursion removal templates

We now present a small library of recursion removal templates. However, to keep them short we have expressed the second patterns of those templates as recursive equations with only "tail-recursion", i.e. readily transformable into iterative patterns by a straightforward application of the reverse Mc Carthy transformation (see [17]).

All these templates have been proved valid, but because of space limitations the proofs will be published later [13]. Proof of weak correctness for some of these templates may be found in [35].

These patterns have varied origins. Some were taken in [2], and sometimes generalized. Some were obtained by abstracting patterns from examples of known equivalent programs. Some were useful intermediary steps appearing in the validation proof of other templates.

- $T_1 = \langle \Sigma_1, \Sigma_1', X_1 \rangle$ where

Σ_1 : $f(x) \Leftarrow$ *if* $a(x)$ *then* $b(x)$ *else* $h(d(x), f(e(x)))$

Σ_1' : $\begin{cases} f'(x) \Leftarrow \textit{if } a(x) \textit{ then } b(x) \textit{ else } g'(e(x), d(x)) \\ g'(x,y) \Leftarrow \textit{if } a(x) \textit{ then } h(y, b(x)) \textit{ else } g'(e(x), h'(y, d(x))) \end{cases}$

X_1 : $\begin{cases} \forall x,y,z & h(x, h(y,z)) = h(h'(x,y), z) \\ \forall x & h(x, \perp) = \perp \end{cases}$

- $T_2 = \langle \Sigma_2, \Sigma_2', X_2 \rangle$ where

Σ_2 : $f(x) \Leftarrow$ *if* $a(x)$ *then* b *else* $h(x, f(e(x)))$

Σ_2' : $\begin{cases} f'(x) \Leftarrow g'(x, b) \\ g'(x,y) \Leftarrow \textit{if } a(x) \textit{ then } y \textit{ else } g'(e(x), h'(y,x)) \end{cases}$

X_2 : $\begin{cases} \forall x & h(x,b) = h'(b,x) \\ \forall x,y,z & h(x, h'(y,z)) = h'(h(x,y), z) \\ \forall x & h(x, \perp) = \perp \end{cases}$

- $T_3 = \langle \Sigma_3, \Sigma_3', X_3 \rangle$ where

Σ_3 : $f(x) \Leftarrow$ *if* $x=a$ *then* b *else* $h(x,f(e(x)))$

Σ_3' : $\begin{cases} f'(x) \Leftarrow g'(a,b,x) \\ g'(x,y,z) \Leftarrow$ *if* $x=z$ *then* y *else* $g'(e'(x),h(e'(x),y)z) \end{cases}$

X_3 : $\begin{cases} \forall x \quad e'(e(x)) = x \wedge e(e'(x)) = x \\ \forall x \quad h(x,\perp) = \perp \end{cases}$

- $T_4 = \langle \Sigma_4, \Sigma_4', X_4 \rangle$ where

Σ_4 : $f(x) \Leftarrow$ *if* $a(x)$ *then* b *else* $h(f(d(d(x))),f(d(x)))$.

(the "Fibonacci scheme")

Σ_4' : $\begin{cases} f'(x) \Leftarrow g'(x,b,b) \\ g(x,y,z) \Leftarrow$ *if* $a(x)$ *then* y *else* $g(d(x),h(z,y),y) \end{cases}$

X_4 : $\begin{cases} \forall x(\neg a(x) \wedge a(d(x))) \supset a(d(d(x))) \\ \forall x \quad h(x,\perp) = \perp \end{cases}$

- $T_5 = \langle \Sigma_5, \Sigma_5', X_5 \rangle$ where

Σ_5 : $f(x,y) \Leftarrow$ *if* $a(x)$ *then* $h(x,y)$ *else* $h(y,f(c(x),d(x)))$

Σ_5' : $f'(x,y) \Leftarrow$ *if* $a(x)$ *then* $h(x,y)$ *else* $f'(c(x),h'(y,d(x)))$

X_5 : $\begin{cases} \forall x,y,z \quad h(x,h(y,z)) = h(h'(x,y),z) \\ \forall x,y,z \quad h(x,h(y,z)) = h(y,h'(x,z)) \\ \forall x \quad\quad h(x,\perp) = \perp \end{cases}$

- $T_6 = \langle \Sigma_6, \Sigma_6', X_6 \rangle$ where

Σ_6 : $f(x,y) \Leftarrow$ *if* $a(x)$ *then* $h(x,y)$ *else* $h(x,f(e(x),y))$

Σ_6' : $f'(x,y) \Leftarrow$ *if* $a(x)$ *then* $h(x,y)$ *else* $f'(e(x),h'(x,y))$

X_6 : $\begin{cases} \forall x,y,z \quad h(x,h(y,z)) = h(y,h(x,z)) \quad (+) \\ \forall x,y,z \quad h(x,h(y,z)) = h(y,h'(x,z)) \\ \forall x \quad\quad h(x,\perp) = \perp \end{cases}$

(+) It is also possible to replace this axiom with the corresponding one for h'.

References :

[0] B. Boyer, J. Moore *"Proving theorems about LISP functions"* JACM 22,1 Jan 75, pp 129-144.

[1] R. Burstall, J. Darlington *"Some Transformations for Developping Recursive Programs"*. Proceedings International Conference on Reliable Software. Los Angeles, California, 1975. Also JACM 24,1 Jan. 77.

[2] J. Darlington, *A Semantic Approach to Automatic Program Improvement"*, Ph.D. Thesis, Dept. of Machine Intelligence, University of Edinburgh, (1972).

[3] J. Darlington & R. Burstall, *"A System which Automatically Improves Programs"*, Proceeding of the third International Joint Conference on Artificial Intelligence, (1973), Stanford, Cal. pp. 479-484. Also Acta Informatica, 1976.

[4] N. Dershowitz & Z. Manna *"The evolution of programs : a system for automatic program modification"* . Proceedings of 4th POPL Conference, Los Angeles 1977.

[5] V. Donzeau-Gouge et al. *"A structure-oriented program editor : a first step towards computer assisted programming"*. Proceedings of ICS 75, Antibes, France.

[6] S. Gerhart. *"Correctness preserving program transformations"*. Proceedings of 2nd POPL Conference, Palo Alto 1975.

[7] S. Gerhart. *"Knowledge about programs : a model and case study"*. Proc of Intern. Conf. on Reliable Software, Los Angeles 75.

[8] G. Huet, *"Résolution d'Equations dans des Langages d'ordre 1,2,•••,ω"* Thèse d'Etat, 1976.

[9] G. Huet *"A unification algorithm for typed λ-calculus"* Theoretical Computer Science 1,1 June 1975 pp.27-58.

[10] G. Huet, G. Kahn et al. *"The MENTOR program transformation system"* Rapport Laboria, in preparation.

[11] D. Knuth & P. Bendix *"Simple word problems in universal algebras"* in Computational problems in abstract algebra. Ed. by J. Leech, Pergamon Press 1969.

[12] D. Knuth & P. Stevenson *"Optimal measurement points for program frequency counts"*. BIT 13(1973) pp.313-322.

[13] B. Lang, *"Quelques transformations éliminant la récursion et leur preuve"* Rapport Laboria, in preparation.

[14] B. Lang, *"ThresholdEvaluation and the Semantics of Call by Name, Assignment and Generic Procedures"*, Proceeding of Fourth Conf. on Principles of Prog. Lang., Los Angeles, 77.

[15] D.B. Loveman, *"Program Improvement by source-to-source transformation"*, JACM, 24,1 (Jan. 77), pp.121-145.

[16] Z. Manna, S. Ness, J. Vuillemin, *"Inductive Methods for Proving Properties of Programs"*. Comm. ACM, Vol 15,n°7, 1972.

[17] J. McCarthy, *"Recursive functions of symbolic expressions and their computation by machine"*
Part I, CACM Vol3 (1960) pp.184-204.

[18] R.E. Milne & C. Strachey, *"A theory of Programming language semantics"*. Chapman & Hall, London,; Wiley, New-York, 1976.

[19] R. Milner, *"Implementation and Applications of Scott's Logic for Computable Functions"*, Proc. ACM Conference on Proving Assertions about Programs, Las Cruces, New-Mexico, 1972.

[20] R. Milner, *"Models of LCF"*. Stanford Artificial Intelligence Laboratory Memo AIM-186, January 73.

[21] B.Mont-Reynault: Forthcoming Ph.D. thesis, Stanford University.

[22] M. Paterson, C. Hewitt, *"Comparative Schematology"*, Record of Project MAC Conference on Concurrent Systems and Parallel Computations, ACM, New Jersey, 1970, p.119-128.

[23] T.A. Standish, D.C. Harriman, D.F. Kibler and J.M. Neighbors. *"The Irvine program transformation catalogue"*. Dept. Inform. and Computer Science, U. Of California at Irvine, Irvine, Calif., Jan. 1976.

[24] T.A. Standish, D.C. Harriman, D.F. Kibler, and J.M. Neighbors. *"Improving and refining programs by program manipulation"*. Proc. 1976 ACM Annual Conf., Oct. 20-22 1976, pp-509-516.

[25] R. Strong, *"Translating Recursion Equations into Flow Charts"*, Journal of Computer and System Sciences, vol. 5,n°3, (June 1971).

[26] R.D. Tennent, *"The denotational semantics of programming languages"*, CACM 19,8 Aug 76, pp. 437-453.

[27] J. Vuillemin, *"Syntaxe, Sémantique et Axiomatique d'un langage de Programmation simple"*. Thèse de Doctorat ès-sciences Mathematiques Université de Paris VI, Paris, (1974).

[28] S.A. Walker and H.P. Strong, *"Characterizations of Flowchartable Recursions."*, Journal of Computer and System Sciences, Vol 7, pp.404-447, (1973).

[29] B. Wegbreit, *"Mechanical program analysis"*, CACM 18,9 (Sept. 75) pp. 528-539.

[30] B. Wegbreit. *"Goal-directed program transformation"*; IEEE, Trans on Software Eng. 2,2 (June 76) pp 69-80.

[31] R. Roboh & E. Sacerdoti. Q.LISP Reference manual. Technical Note 81, SRI, Aug 73.

[32] M. Stickel , Forthcoming Ph.D. thesis, Carnegie Mellon University.

[33] M. Gordon, R. Milner & C. Wadsworth
Edinburgh LCF
private communication.

[34] G. Huet Confluent reductions : abstract properties and application to term rewriting systems
FOCS conference 1977.

[35] D.C. Cooper, *"The equivalence of certain computations"*, Computer Journal 9(1966) pp.45-52.

[36] R. Steinbrüggen, *"Equivalent recursive definitions of certain number theoretical functions"*. Technical Report INFO-7714 (June 77), Technische Universität Munchen.

[37] A.K. Chandra, *"Efficient compilation of linear recursive programs"*. Res.Rep CS-72-282, Stanford University, 1972.

[38] R.S. Bird, *"Notes of recursion elimination"*, CACM 20 (1977) pp. 434-439.

[39] N.H. Cohen, *"Mechanical analysis and restructuring of recursive programs"*, Internal report, Harvard University (May 1976).