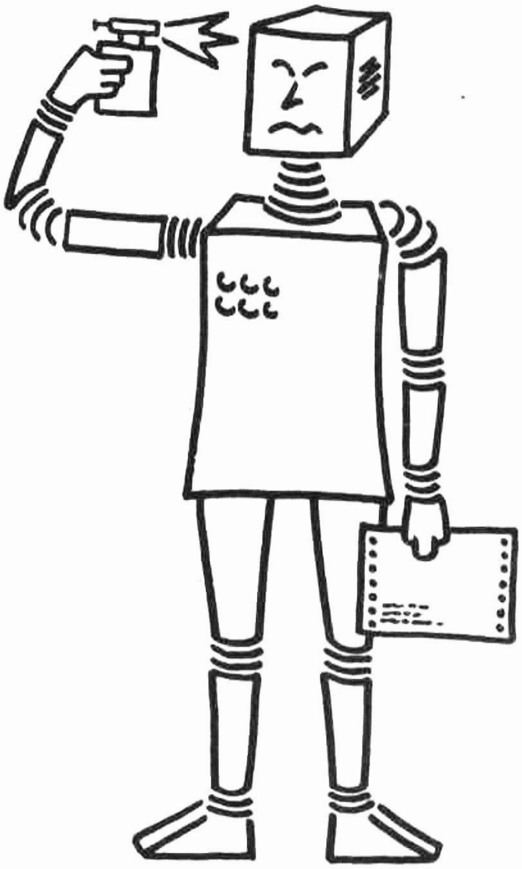
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI MEMO

SEKI-PROJEKT



A Second-Order Matching Procedure
for the Practical Use
in a Program Transformation System

Michael Gerlach

Memo SEKI-83-13

A SECOND-ORDER MATCHING PROCEDURE FOR THE PRACTICAL USE IN A PROGRAM TRANSFORMATION SYSTEM

M. Gerlach

Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
West Germany

November 1983

Abstract

One way of transforming functions towards greater efficiency is to apply a transformation rule given in form of two program schemes. The first great step in applying such a rule is to recognize that a given function is an instance of such a program scheme. We describe a procedure for this task working on a second-order term language. Using this language it is possible to comprise the essential features of a wide class of programs into one scheme, independently of their arity as well as the number and arity of auxiliary functions used in their definitions.

This research was supported by the Bundesministerium für Forschung und Technologie under contract IT.8302363.

Contents

I. Introduction

The specification system SPESY, currently developed at the University of Kaiserslautern, provides an environment for the construction of highly reliable software. The whole system is based on the paradigm of stepwise refinement with the following main tasks:

1. Specification of the requirements by using signatures and logical formulas (axiomatic specification).
2. Construction of algorithms fulfilling these requirements but still presented as abstract data types (algorithmic specification).
3. Optimizing these algorithms without loosing their correctness.
4. Implementing the abstract data types by Pascal programs and proving this implementation correct with respect to the abstract types.

This paper is concerned with step 3.: How can programs be optimized without loosing their correctness? An important method for this kind of program development is the use of transformation rules [Broy, Pepper, 1981]. A transformation rule may be regarded as a tripel $\langle \Sigma, X, \Sigma' \rangle$ [Huet, Lang, 1978] with the following components:

- A scheme Σ denoting the class of programs the rule is applicable to.
- A condition X which must be true to make the transformation semantics preserving.
- A scheme Σ' denoting the result of the transformation.

The scheme Σ always contains variables. The set of all legal substitutions for these variables constitutes the set of all programs matching the scheme Σ. Given a program P and a scheme Σ we must decide whether there is a substitution σ such that

$\sigma\Sigma = P$, i.e.: "Is P an instance of the scheme Σ"? That is the question this paper is dealing with.

To make a single transformation rule as powerful as possible we try to make the set of legal variable bindings as large as possible. Look at the scheme for linear recursive functions:

f(x) := <u>if</u> B(x) <u>then</u> ϕ(f(K(x)),E(x)) <u>else</u> H(x)

The class of functions described by that scheme will comprise functions

- that are of arbitrary arity, i.e. x may denote several parameters $x_1,\ldots,x_n$, and K several functions $K_1,\ldots,K_n$.
- where ϕ is of arbitrary arity, i.e. E may denote several functions $E_1,\ldots,E_m$ computing arguments for ϕ.
- where the variables ϕ, K, E, H are any cascade of function calls, from identity to complex conditional expressions.

Here are two linear recursive functions matching the given scheme:

```
insert(x,l) :=
   if    if   empty?(l)
         then false
         else le (first(l),x)
   then  put(first(l),insert(x,rest(l)))
   else  if    empty?(l)
         then  put(x,empty)
         else  put(x,l)

    with the variable bindings
    f    insert
    x    x,l
    B    λuv. if    empty?(v)
              then  false
              else  le (first(v),u)
```

```
    ϕ     λuv. put (v,u)
    K     λuv. u, λuv. rest(v)
    E     λuv. first(v)
    H     λuv. if     empty? (v)
               then   put (u,empty)
               else   put (u,v)


sub(n,m) :=
   if     gt(m,0)
   then   sub(subl(n),subl(m))
   else   n

    with the variable bindings

    F    sub
    x    n,m
    B    λuv. gt(v,0)
    ϕ    λu.  u
    K    λuv. subl(u), λuv. subl(v)
    E
    H    λuv. u
```

To gain this wide class of functions matching one scheme, we use second-order variables and multivariables. Second-order variables are well-known from higher-order logic and the typed λ-calculus [Church, 1940]. The value of a second-order variable must always denote a function, given in form of a λ-abstraction. The concept of multivariables is introduced in this paper. A multivariable may have several values, and the substitution rule substitutes a multivariable by its values (and not by the list consisting of these values). In the scheme for the linear recursion B, ϕ, K, E, H are second-order variables, and x, K, E are multivariables. Hence, E and K are second-order multivariables.

In the following chapter we give a formal definition of the term

language. It was designed for the practical use in a program transformation system, and is therefore a modification and extension of the ordinary typed λ-calculus. The task of matching two terms is specified, and in the succeeding chapter an algorithmic solution of that task based on the work of [Huet, Lang, 1978] is described in detail. The transformation example given afterwards shows how to use matching for recursion removal which is an important issue in making programs more efficient.

II. The Term Language

There is a set C of constants and a set V of variables. C and V must be disjoint. Furthermore, V is divided up into the following subsets:

- V_1 simple first-order variables
- V_{1*} first-order multivariables
- V_2 simple second-order variables
- V_{2*} second-order multivariables
- V_B bound variables

The explicit definition of a set of bound variables that is disjoint to all others is for avoiding collision problems when substituting variables. $V_S := V - V_B$ is called the set of scheme variables. (Remark: The difference and union of sets is denoted by ´-´ and ´+´.)

Terms are atomic terms, abstractions, applications and lists.

Atomic terms are constants, first-order and bound variables.

If $u_1,\ldots,u_n$ are bound variables and t is an atomic term or an application, $(\lambda u_1\ldots u_n \,.\, t)$ is an abstraction.

If $t_1,t_2,\ldots,t_n$ are atomic terms or applications or lists, and ϕ

is a second-order variable or an abstraction, then $\phi(t_1, t_2, \ldots, t_n)$ is an <u>application</u>. $(t_1, t_2, \ldots, t_n)$ is a <u>list</u>, which may also be written as $t_1(t_2, \ldots, t_n)$.

The classical λ-calculus does not define lists as terms. The extension just established is introduced for the following reason: The definition of the sub operation

(sub(x,y):= (<u>if</u> ... <u>then</u> sub(...)...))

is an instance of the list

(f(u):= (<u>if</u> ... <u>then</u> f(...)...)),

f and u being first order variables. Equivalent terms not being lists will contain the fixpoint operator Y, the conditional operator C, and f as a second-order variable, c.f.

Y(λfu . C(...f(...)...))

The former notation seems to be more natural and allows to simplify the algorithm that matches two terms.

The <u>evaluation</u> rule for our terms is called β-reduction. It specifies how to apply an abstraction to a sequence of arguments. It is defined by the following function β:

$$\beta(f,(a_1,\ldots,a_k)) = \begin{cases} s[u_i/a_i, 1 \leq i \leq k] & \text{if } f = (\lambda u_1 \ldots u_n \,.\, s) \;\&\; n=k \\ \text{undefined} & \text{if } f = (\lambda u_1 \ldots u_n \,.\, s) \;\&\; n \neq k \\ f(a_1,\ldots,a_k) & \text{otherwise} \end{cases}$$

$s[u_i/a_i, 1 \leq i \leq k]$ denotes the term s in which every occurrence of u_i is replaced by a_i. Collision problems between free and bound variables do not occur due to our strict classification of variables.

A <u>substitution</u> <u>tupel</u> is a tupel $\langle v, t_1, \ldots, t_n \rangle$ meeting the following properties:

1. $v \in V_s$
2. If $v \in V_1 + V_2$ then n=1. Hence, $\langle v, t_1 \rangle$ is a well-known substitution pair.
   If $v \in V_{1*} + V_{2*}$ then $n > 0$.

3. If $v \varepsilon V_2 + V_{2*}$ then all t_i must be abstractions.
4. None of the t_i may contain a free occurrence of a variable $u \varepsilon V_B$

A substitution is a set of substitution tupels pertaining to distinct variables.

For every substitution σ and every term t the term σt denotes the result of the application of σ to t. In some cases, however, σt is not defined at all: What should σt be if $\sigma = \{\langle t,a,b\rangle\}$?

1. t is atomic:

$$\sigma t = \begin{cases} \text{undefined} & \text{if } t \varepsilon V_{1*} + V_{2*} \\ t' & \text{if } \exists \langle t,t'\rangle \varepsilon \sigma \\ t & \text{otherwise} \end{cases}$$

2. t is an application $\phi(t_1,\ldots,t_n)$:

$$\sigma t = \begin{cases} \text{undefined} & \text{if } \phi \varepsilon V_{2*} \\ \beta(\sigma\phi,\ \sigma(t_1,\ldots,t_n)) & \text{otherwise} \end{cases}$$

3. t is an abstraction $(\lambda u_1 \ldots u_n . t')$:

$\sigma t = (\lambda u_1 \ldots u_n \ . \ \sigma t')$

(For the conditions 1 and 4 variable bindings may be ignored)

4. t is a list $(t_1,\ldots,t_n)$:

$$\sigma t = (x'_{11}, x'_{12}, \ldots, x'_{1m1},$$
$$x'_{21}, x'_{22}, \ldots, x'_{2m2},$$
$$\ldots..$$
$$x'_{n1}, x'_{n2}, \ldots, x'_{nmn})$$

where

4.1 if $t_i \varepsilon V_{1*}$ then:

either $\langle t_i, x'_{i1}, x'_{i2}, \ldots, x'_{imi}\rangle \ \varepsilon \ \sigma$

or $x'_{i1} = t_1$ & $mi=1$

4.2 if $t_i = \phi(a_1,\ldots,a_k)$ & $\phi \varepsilon V_{2*}$ then:

either $\exists$ $\langle \phi, f_1, f_2, \ldots, f_{mi} \rangle \varepsilon \sigma$ &

$x'_{ij} = \beta(f_j, \sigma(a_1,\ldots,a_k))$, $j \leq mi$

or $x_{i1} = \beta(\phi, \sigma(a_1,\ldots,a_k))$ & $mi=1$

4.3 otherwise:

$x'_{i1} = \sigma t_i$ & $mi=1$

The composition of two substitutions σ and σ' is

$$\sigma\sigma' = \{\langle v, q_1,\ldots,q_k \rangle \mid (q_1,\ldots,q_k) = \sigma'[\sigma(v)] \text{ \& } \exists\, t_1,\ldots,t_n: \langle v, t_1,\ldots,t_n \rangle \varepsilon \sigma + \sigma'\}$$

Let t, t' be two terms. We say that t' matches t if there exists a substitution σ with $\sigma t = t'$ (σ is called a "matcher"). Two substitutions σ, σ' are said to be dependent if there is a substitution σ'' such that $\sigma''\sigma = \sigma'$ or $\sigma''\sigma' = \sigma$. In a first-order language the following proposition holds:

$\sigma t = t'$ & $\sigma' t = t'$ $\rightarrow$ σ and σ' are dependent.

I.e. if t' matches t, then there is a unique (up to renaming of bound variables) most general matcher. In a second-order term language this statement is no longer true. I.e. if t is a program scheme and t' some procedure then there may be several independent ways to interpret the procedure as an instance of the scheme. Fortunately the set of all independent matches is finite, and there is an algorithm to evaluate this set. This is not self-evident since the unification problem is undecidable in second-order logic [Goldfarb, 1981].

Example

The variables: $V_1 = \{f, u, w\}$

$V_{1*} = \{v\}$

$V_2 = \{\phi, \psi\}$

$V_{2*} = \{K\}$

The substitution: $\sigma = \{\langle f,F\rangle, \langle u,X\rangle,$
$\langle v,Y,Z\rangle, \langle w,(Y,Z)\rangle,$
$\langle \psi, \lambda x.x\rangle, \langle \phi, \lambda x_1 x_2.\ G(x_2,H(x_1))\rangle,$
$\langle K, \lambda x_1 x_2.\ x_2, \lambda x_1 x_2.\ x_1\rangle\}$

t	t´=σt	substitution rule
(1) f(u,v)	F(X,Y,Z)	4.3, 1., 4.1
(2) f(u,w)	F(X,(Y,Z))	4.3, 1.
(3) $\phi(S,\psi(T))$	G(T,H(S))	2., 4.3, 1.
(4) $\phi(K(v))$	G(Y,H(Z))	2., 4.2, 4.1

- The examples (1) and (2) are showing the difference between simple variables and multivariables and how the latter can be used to express the idea that a function may have an arbitrary number of parameters.
- σ is a matcher for each pair of terms t and t´.
- There is another matcher σ´ for t and t´ in example (3) such that σ and σ´ are independent:

$$\sigma' = \{\langle \phi, \lambda x_1 x_2.\ G(x_2,x_1)\rangle, \langle \psi, \lambda x.\ H(x)\rangle\}$$

III. The Matching Procedure

An algorithm matching two second-order terms is described in [Huet, Lang, 1978]. That algorithm has been implemented with the following modifications:

1. The concept of multivariables introduced in the previous chapter caused some extensions of the algorithm.
2. Introducing lists as a kind of terms allows us to exclude that a matched scheme contains abstractions. So function definitions may be expressed as lists instead of using the fixpoint operator.
3. The procedure does not look for all independent matchers at

once. With each call it produces <u>one</u> substitution together with some information that can be used by the procedure in the next call. Given this information it will find the next matcher (if there is any). It is assumed that a second-order variable comprising a recursion should have a value being as simple as possible, because many transformation rules require for a algebraic property of the value of such a variable. Therefore the complexity of that value determines the order of the produced solutions [→ appendix].

4. A recursive operation scheme like
   f(x):= <u>if</u> b(x) <u>then</u> f(k(x)) <u>else</u> h(x)
   normally implies that there is no occurrence of f in the values of b, k, and h. That´s why there is an option telling the matching procedure that f must not occur inside the value of any second-order variable.
5. A matcher for the terms
   f(x):= g(x) and F (X):= X
   may contain the substitution tupel $\langle g,\lambda u.X\rangle$. Since X is a parameter, such a substitution is not desired (we expect $\langle g,\lambda u.u\rangle$). That´s why there is an option telling the matching procedure that no function parameter may occur in the value of a second-order variable.
6. We are not using the term "matching tree" as Huet and Lang do. However, we describe the <u>state</u> of the matching procedure by a set of items still called nodes corresponding to the terminal nodes in the matching tree. A node is a tupel (P,σ) where P is a set of pairs of terms and σ is a substitution. A node (P,σ) represents an alternative in the search for independent matchers. σ contains the substitution tupels already found, and P the pairs of terms not yet matched.

There are some <u>restrictions</u> to the applicability of our matching procedure pertaining to multivariables. These restrictions do not effect our applications but make the matching procedure simpler and more efficient. (The following notion of the ´first occurence´ is assuming the usual prefix notation of terms.)

1. The first occurence of a first-order multivariable must not be an argument of an application.
2. A list may contain a first-order multivariable on top level several times but none of these occurences must be the first one inside the term comprising the list.
3. Different applications that contain applications of the same second-order multivariable as arguments may cause the matching procedure to be incomplete: It will not find all independent matchers.
4. The scheme to be matched must not be a multivariable application or a first-order multivariable (applying a substitution to such terms is not defined at all).

This is the top-loop of the matching procedure given two terms T1, T2:

```
Initialization: RESULT:= Ø
                N:=({<T1, T2>},Ø
                S:={N}

LOOP:           if S = Ø then ready : return RESULT
                N:= any node in S
                S:= S - N
                N:= SIMPLIFY (N)
                if N = F  then mismatch: goto LOOP
                if N = (Ø,σ)
                   then a matcher is found:
                        RESULT := RESULT + {REDUCE(σ,T1)}
                        goto LOOP
                   else
                        S:= S + {σN | σ ε MATCHAPPLICATION (N)}
                        goto LOOP
```

The procedure SIMPLIFY is essentially a first-order matcher keeping second-order terms unmatched.

```
SIMPLIFY (N):
P:= pairs (N)
pairs(N):= Ø
for <t1,t2> ε P do
    σ´:= SIMPLIFY1 (t1,t2,N)
    if σ´= F then return F
    N:= σN
    P:= σP
```

The procedure SIMPLIFY1 is matching two terms t_1, t_2 . It is adding pairs of corresponding second-order subterms of t_1 and t_2 to the node N and yields a substitution resulting from the first-order match of t_1 and t_2. If t_1 and t_2 do not match the value is F. The second-order variables h_i used in the following algorithm are created by the program and must not occur anywhere else in the nodes. Such variables are created in the procedure MATCHAPPLICATION, too.

```
SIMPLIFY1 (t1, t2, N):
if       t1=t2=ε then ε
else if t1=ε v t2=ε then  F
else if t1 εVs then <t1,t2>
else if t1εC then
        if t1=t2 then ε else  F
else if t1 is an application then
        add <t1,t2> to N; ε
else    let (t11,...,t1n) = t1, (t21,...,t2m) = t2 in
     if t11 ε V1* then
        if n=1 then <t11,t21,t22,...,t2m>
               else <t11,t21,t22,...,t2(m-n+1)> o
                    SIMPLIFY1((t12,...,t1n),(t2(m-n+2),...,t2m),N)
else if t11= (t110,t111,...,t11s) & t110 ε V2* then
        let σ´=<t111,λx1...xs . h1(x1,...,xs),
                    ...,
                    λx1...xs . hm-n+1(x1,...,xs)>
```

```
    in
            σ´ o SIMPLIFY1 (σ´t_1, t_2, N)
else  let σ´= SIMPLIFY1 (t_11, t_21, N) in
      if σ´= F  then  F
         else σ´ o SIMPLIFY1 (σ´(t_12,...,t_1n), (t_22,...,t_2m), N)
```

The procedure MATCHAPPLICATION selects a pair of terms from a given node. Since the node is simplified the first term t_1 of the pair is an application. The value of the procedure is a set of independent substitution tupels. Each of them is representing a different way to match the two terms given to the matching procedure. During simplification, however, some of them may be proved invalid.

If the selected term t_1 does not contain any application of a multivariable MATCHAPPLICATION produces substitution tupels for the variable being the head of t_1. However, if this variable has an argument that is an application of a multivariable we do not know its arity. That´s why first a substitution tupel for the multivariable is produced. Hereby the problem about the arity may appear again. So we have to search for the somehow inner-most multivariable application called the "fixed V_{2*}-application".

```
FIXED-V_2*-APPLICATION ((s_1,...,s_n)):
for 1≤i≤n do
   if s_1 is a multivariable application then
      let s´= FIXED-V_2*-APPLICATION (s_i) in
      if there is such an s´
      then return s´ else return s_i
```

```
MATCHAPPLICATION (N)
Select a pair <t_1, t_2> from N
If there is a fixed V_2*-application ψ(q_1,...,q_k) in t_1
then one substitution tupel for ψ containing the following
      values:
```

a) all projections $(\lambda x_1 \ldots x_k .\ x_i)$, $1 \leq i \leq k$

b) all imitations

$(\lambda x_1 \ldots x_k .\ f_{i1}(h_{i1}(x_1,\ldots,x_k),\ldots,h_{iri}(x_1,\ldots,x_k)))$

where $(f_{i1}, f_{i2},\ldots,f_{iri})$ is a list contained in t_2 and h_{ij} are new second-order variables

<u>else</u> <u>let</u> $t_1 = \phi(p_1,\ldots,p_n)$ <u>in</u>

<u>if</u> t_2 is atomic

<u>then</u> all substitution tupels $\langle\phi,\ \lambda x_1 \ldots x_n .\ x_i\rangle$, $1 \leq i \leq n$ and the imitation $\langle\phi, \lambda x_1 \ldots x_n .\ t_2\rangle$

<u>else</u> <u>let</u> $(g_o, g_1,\ldots,g_m) = t_2$ <u>in</u>

all $\langle\phi,\lambda x_1 \ldots x_n .\ x_i\rangle$ and the imitation

$\langle\phi,\lambda x_1 \ldots x_n .\ g_o(h_1(x_1,\ldots,x_n),\ldots,\ h_m(x_1,\ldots,x_n))\rangle$

In this algorithm the word "all" is to be modified: When producing the projections a look-ahead can be made checking whether the result of the projection will match the corresponding subterm of t_2. In this way the number of projections can be reduced. Point 4. and 5. at the beginning of this section have to be regarded when creating the imitations: e.g. a recursive function call will normally not be imitated.

The last step of the matching procedure is the <u>reduction</u> of the result. The value of a second-order variable with a multivariable application as an argument is an abstraction of maximal arity created by MATCHAPPLICATION. Parameters of that abstraction which are not used in the body may be abolished. But the corresponding values of the multivariable must be deleted, too.

Example: Given the scheme $\phi(\psi(x),y)$ and the substitution

$\{\langle\phi,\lambda x_1 x_2 x_3 .\ F(x_1,x_3)\rangle,\langle\psi,\lambda x_1 . x_1,\lambda x_1 .\ G(x_1)\rangle\}$

we may reduce the latter and have

$\{\langle\phi,\lambda x_1 x_3 .\ F(x_1,x_3)\rangle,\langle\psi,\lambda x_1 . x_1\rangle\}$

Furthermore all substitution tupels belonging to a variable h_i created by the matching procedure are eliminated by REDUCE.

IV. A Transformational Example

In this section we will show how to obtain the iterative version of the mult operation starting with the noniterative definition

```
mult (x,y):= if x=0 then 0 else
                  if y=0 then 0 else add (x,mult(x,y-1))
```

using the rule $\langle \Sigma, \Sigma', X \rangle$ with

```
Σ  ≡ L(m):= if B(m) then φ(L(K(m)), E(m)) else H(m)
Σ´ ≡ L(m):= G(m,H(c)),
     G(m,z):= if B(m) then G(K(m),φ(z,E(m))) else z
X  ≡ ∃c: B≅ λw. neq(w,c) &
         ∀r,s,t: φ(φ(r,s),t) = φ(φ(r,t),s)
```

First we will replace the infix operators by the corresponding prefix operators, eq for equality, and pred for predecessor:

```
mult (x,y):= if eq(x,0) then 0 else
                  if eq (y,0) then 0 else add (x,mult(x,pred(y)))
```

Then a normalization procedure must be performed since our recursive function scheme looks like:

```
f(u):= if b(u) then ... f ... else h(u)
```

In our example, this normalization has to combine the conditionals:

```
mult(x,y):= if and (neq(x,0),neq(y,0))
             then add(x, mult(x,pred(y)))
             else 0
```

The term Σ is a pattern for the class of functions the rule is applicable to. The pattern Σ' describes the result of the transformation in terms of the variables used in Σ (and X). Additionally there is a condition X that must be fulfilled to make the transformation semantics preserving. X contains the variables of Σ, but may also introduce new variables by existential qualification.

To apply this rule to a definition the following steps are to be done:

1. Find a matcher σ such that $\sigma\Sigma$ equals the definition. If there are several independent matchers, try step 2. with all of them.
2. Find a matcher $\sigma' > \sigma$ such that σ X is a true predicate.
3. Compute $\sigma'\Sigma'$ to gain the result of the transformation.

The following matcher can be used here:

$$\sigma'=\{\langle L,add\rangle,\langle m,x,y\rangle,\langle c,0,0\rangle,$$
$$\langle B,\lambda uv.and(neq(u,0),neq(v,0))\rangle,\ \langle \phi,\lambda uv.add(v,u)\rangle$$
$$\langle K,\lambda uv.u,\lambda uv.pred(v)\rangle,\langle H,\lambda uv.0\rangle,\ \langle E,\lambda uv.u\rangle\ \}$$

There are several ways how to check the condition X, and find the substitution tupel for c:

- Have a look at a knowledge base where the underlying data types and their operators are described. Since our condition is a rather special one we will not find it in our knowledge base. But we will gain some basic properties like associativity and commutativity of the operators.
- Use some automated proof method to show the condition using the basic properties of the underlying operations. (In our simple example ϕ denotes the add operation. In more complex cases ϕ may be an arbitrary composition of many other operations including conditionals.)
- Ask the user whether the condition holds.

In our system we will use all these methods. Moreover, our goal is to minimize the usercalls, and, when a user call is unavoidable, to provide all the tools available in the specification system.

The result of the transformation is the term $\sigma\Sigma'$:

```
mult(x,y) := G(x,y,0) ,
G(x,y,z)  := if and (neq(x,0), neq(y,0))
             then G(x, pred(y), add(z,x))
```

Conclusion

We described a matching procedure for a second-order term-language. Such a procedure is necessary to apply transformation rules to programs in order to develop them towards greater efficiency. A lot of rules for recursion removal being a great issue in optimizing functional programs can be found in [Petersen, 1983] and [Bauer, Wössner, 1981]. The matching procedure has been implemented in INTERLISP and is used by a system for automated recursion removal [Geißler, 1984]. It applies to linear, cascaded and nested recursions as well as to somespecial classes of recursions, and transforms them to tail recursion as far as it is possible within the current state of art.

Appendix: Scoring Nodes

The matching procedure contains two steps that are non-deterministic so far: In the top-loop it has to select a node representing an alternative in the search for independent matchers. In the function MATCHAPPLICATION it has to choose a pair of terms in order to proceed with the second-order matching task.

In this appendix we will suggest a decision procedure suitable for the use in a program transformation system. If there is a second-order variable Φ in the given scheme that has a recursive call among its arguments as well as another application of a second-order variable, the matching procedure will produce the substitution with the most simple value of Φ first.

The node selection is achieved by a scoring function SCORENODE that computes the distance between a node and the most simple solution. The node with the best score will be selected at the top-loop decision point.

FINDGOAL is an auxiliary function looking for the function names that must occur in the value of a second-order variable in order to meet the requirement that a recursive call must not be part of the value of any second-order variable:

FINDGOALS (t):

<u>if</u> $t = t_o(t_1, \ldots, t_n)$ &

t_o is not a recursive call &

t contains a recursive call

<u>then</u> $\{t_o\} + \bigcup_i$ FINDGOALS (t_i)

<u>else</u> {}

GOAL is calling FINDGOALS for all relevant schema variables:

```
GOALS (N):
{<t10,FINDGOALS (t2) > |
     <t1, t2> ε pairs(N) &
     t1 = t10 (t11, ..., t1n ) &
     t10 ε V2 + V2* &
     (t11, ..., t1n ) contains a recursive call &
     ∄ j: t1j is a second-order application }

SCORENODE (N):
  score:= 0
  for   (x,u1, ..., un) ε subst(N)  do
      for all atoms a in (u1, ..., un)  do
           if  ∄ <x, g> ε GOALS(N)
           then  if   a ε g
                 then  score :=  score  + 999
                 else  score :=  score  - 1
           elsif  x is not a name generated by the matching
                        procedure
           then    score := score +1

  score
```

It depends on the selection of a pair of terms in the function MATCHAPPLICATION how long it takes for an invalid alternative to be shown incorrect. When selecting the pairs pertaining to automatically generated function variables first, each imitation produced by MATCHAPPLICATION is processed towards its success or failure before other imitations are generated. This realizes a depth-first strategy which has be shown to be very efficient.

References

Bauer, F. L., Wössner, H.: Algorithmische Sprache und Programmentwicklung, Springer Verlag, Berlin, 1981

Broy, M., Pepper,P.: Program Development as a Formal Activity, IEEE Tr. on Software Engineering, Vol. SE-7, No. 1, January 1981, p. 14-22

Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs, JACM 24, 1 (Jan. 77) 44-67

Church, A.: A Formulation of the Simple Theory of Types, J. Symb. Logic 5(1) (1940) . 56-68

Geißler, C.: Ein System zur automatischen Elimination von Rekursionen, Diplomarbeit, Bonn, Kaiserslautern, erscheint 1984

Goldfarb, D.: The Undecidability of the Second Order Unification Problem, J. of Theoretical Computer Science, 13, 1981, p. 225-230

Huet, G., Lang, B.: Proving and applying Program Transformations Expressed with Second-Order Patterns, Acta Informatica 11, 31-55 (1978)

Petersen, U.: Die Elimination von Rekursionen, SEKI-Memo 83-10, Kaiserslautern, October 1983

11/22/1983

SEKI Memos

The following memos are available free of charge from

Mrs. Dorothea Kilgore
Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
West Germany

MEMO SEKI-81-01	U. Bartels, W. Olthoff and P. Raulefs: APE: An Expert System for Automatic Programming from Abstract Specifications of Data Types and Algorithms.
MEMO SEKI-81-03	Peter Raulefs: Expert Systems: State of the Art and Future Prospects.
MEMO SEKI-81-04	Christoph Beierle: Programmsynthese aus Beispielsfolgen.
MEMO SEKI-81-05	Erich Rome: Implementierungen Abstrakter Datentypen in terminaler Algebrasemantik.
MEMO SEKI-81-06	Christoph Beierle: Synthesizing Minimal Programs from Traces of Observable Behaviour.
MEMO SEKI-81-07	Dieter Wybranietz: Ein verteiltes Betriebssystem für CSSA.
MEMO SEKI-81-08	Ulrich Bartels and Walter Olthoff: APE - Benutzerbeschreibung.
MEMO SEKI-82-01	Hans Voß: Programming in a Distributed Environment: A Collection of CSSA Examples.
MEMO SEKI-82-02	Hartmut Grieneisen: Eine algebraische Spezifikation des Software-Produkts INTAKT.
MEMO SEKI-82-03	Christian Beilken, Friedemann Mattern and Michael Spenke: Entwurf und Implementierung von CSSA - Beschreibung der Sprache, des Compilers und des Mehrrechnersimulationssystems. Printed in 6 volumes, which can be ordered individually: Vol-A: Konzepte Vol-B: CSSA-Sprachbeschreibung Vol-C: CSSA-Systembenutzung Vol-D: CSSA-Programmbeispiele

Vol-E1: Programmdokumentation Teil I
Vol-E2: Programmdokumentation Teil II

MEMO SEKI-83-01	Wilfried Schrupp and Johann Tamme: Spezifikation und abstrakte Implementierung des Aufbereitungsteils von INTAKT.
MEMO SEKI-83-02	Frank Puppe and Bernd Puppe: Overview on MED1: A Heuristic Diagnostic System with an Efficient Control-Structure.
MEMO SEKI-83-03	Elisabeth Hülsmann: LISP-SP : A portable INTERLISP Subset Interpreter for Mini-Computers.
MEMO SEKI-83-04	FrankPuppe: MED1 - Ein heuristisches Diagnosesystem mit effizienter Kontrollstruktur.
MEMO SEKI-83-05	Horst Peter Borrmann: MODIS - Ein Expertensystem zur Erstellung vonReparaturdiagnosen für den Ottomotor und seineAggregate.
MEMO SEKI-83-06	Harold Boley: From Pattern-Directed to Adapter-Driven Computation via Function-Applying Matching.
MEMO SEKI-83-07	Christoph Beierle and Angi Voß: Canonical Term Functors and Parameterization-by-use for for the Specification of Abstract Data Types.
MEMO SEKI-83-08	Christoph Beierle and Angi Voß: Parameterization-by-use for hierarchically structured objects.
MEMO SEKI-83-09	Christoph Beierle, Michael Gerlach and Angi Voß: Parameterization without parameters in : The History of a Hierarchy of Specifications.
MEMO SEKI-83-10	Ulrike Petersen: Elimination von Rekursionen.
MEMO SEKI-83-11	Gerd Krützer: An Approach to Parameterized Continuous Data Types.
MEMO SEKI-83-12	Richard Göbel: A Completion Procedure for Globally Finite Term Rewriting Systems.
MEMO SEKI-83-13	Michael Gerlach: A Second-Order Matching Procedure for the Practical Use in a Program Transformation System.