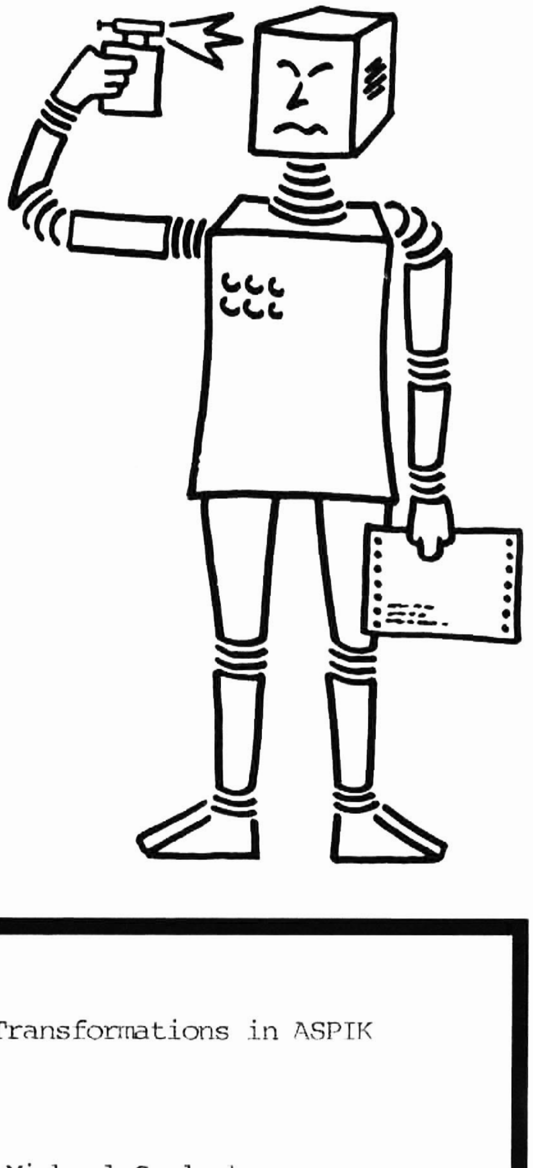
SEKI-PROJEKT

SEKI MEMO

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



Program Transformations in ASPIK

Michael Gerlach

MEMO SEKI-85-04

PROGRAM TRANSFORMATIONS IN ASPIK

Michael Gerlach

Fachbereich Informatik
University of Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern
West Germany

August 1985

MEMO-SEKI-85-04

Abstract

Program transformations need a uniform framework for expressing algorithms as well as algebraic properties. In this paper we present the embedding of program transformations in a software specification system. We demonstrate the methodology by developing a sorting algorithm towards more efficiency, and show how the semantics of the specification language is the foundation for proving transformation rules correct. We are discussing theoretical aspects - the correctness of transformations - as well as technical issues, e.g. two kinds of user interfaces to a program transformation system.

Contents

1. Introduction

Program transformations are a tool to develop reliable software: starting with well-understood high-level programs we get low-level programs correct w.r.t. the high-level programs, supposing the transformation itself is correct. This approach contrasts with writing programs on different levels using an editor, and proving the correctness afterwards. Both methods are complementing each other according to the following development paradigm:

- State some formal or informal requirements
- Write algorithms on a level that is easy to understand
- Perform the validation or the correctness proof of the algorithms using appropriate tools
- Develop the algorithms using program transformations thus loosing perspicuity, but saving correctness.

Both algorithmic and algebraic knowledge is employed by program transformations. Algorithms are the objects of the transformations, algebraic properties are used for simply rewriting subterms, or to show conditions ensuring the correctness of complex transformation rules. In this paper we will describe the embedding of a program transformation system in a specification environment ([BGGORV84]). Since specifications written in our specification language ASPIK consist of an algebraic and an algorithmic part within one syntactic frame, the program transformation subsystem has access to both knowledge sources in a uniform way.

There are two major ways to implement program transformations ([Ba79]): The transformation rules can either be described as algorithms, which take a given program as input and produce an equivalent one as output, or can be given as an ordered pair of templates together with an applicability condition denoting a conditional production rule. In our program transformation system both techniques have been employed, the former for the basic transformations unfold, fold etc. (Chapter 3), the latter for recursion removal (Chapter 4).

2. The Specification Language

In this chapter, we give a brief overview on our specification language ASPIK, as far as it is necessary to understand the following examples (see [BGGORV84] for further details).

A specification denotes an abstract data type introducing sorts and operations, maybe using other specifications, thus being an enrichment or a refinement. It may also contain formulas in first order logic stating properties about the defined or imported operations. Additionally, there is an optional algorithmic part, defining operations in terms of other operations in a purely functional manner.

2.1 The Example

We shall demonstrate the various transformation methods by applying them to specifications about sorting lists. These specifications have been developed in a systematical manner in [BGV83], and are the starting point of our transformations.

Fig.1 and 2 show the standard data types BOOL, NAT, ELEM, ORDELEM and LIST. ELEM and ORDELEM are examples for so-called loose specifications. They denote - roughly spoken - any algebra

```
spec BOOL
 /* standard definition of the booleans */
 ...
endspec

spec NAT
 /* standard definition of the natural numbers */
 ...
endspec

spec ELEM
 /* just a sort with its equality */
   use BOOL
   sorts elem
   ops  =  : elem elem → bool
endspec

spec ORDELEM
 /* ELEM enriched with an ordering */
   use ELEM
   ops  ≤  : elem elem → bool
  props
    all x,y ∈ elem:
        x≤x  ∧
       (x≤y  ∧  y≤z  →  x≤z)  ∧
       (x≤y  ∧  y≤x  →  x=y)
endspec
```

Fig.1: The basic data types

```
spec LIST
 /* lists of elems */
   use    ELEM,NAT
   sorts list
   ops    empty: → list
          put: elem list → list
          first: list → elem
          rest: list → list
          empty?, simple?: list → bool
          in?: elem list → bool
          append: list list → list
          length: list → nat
          occurences: elem list → nat
   props
     all l ∈ list, x ∈ elem:
       empty?(put(x,l)) = false ∧
       empty?(empty) = true     ∧
       first(put(x,l)) = x      ∧
       rest(put(x,l)) = l

     all l1, l2 ∈ list:
       append(l1,empty) = l1      ∧
       append(empty,l1) = l1      ∧
       append(append(l1,l2),l3) =
       append(l1,append(l2,l3))

spec body
  constructors empty, put
  define ops
     first(l)    = case l is  *empty: error-elem
                              *put (n,l1):n        esac
     rest(l)     = case l is  *empty: error-elem
                              *put (n,l1):l1       esac
     empty?(l)   = case l is  *empty: true
                              otherwise : false    esac

     simple?(l) = if empty?(l) then true
                   else empty?(rest(l))
     in?(e,l) = if empty?(l)
                then false
                else first(l)=e
                     or in?(e,rest(l))
     append(l1,l2) = if empty?(l1)
                     then l2
                     else put(first(l1), append(rest(l1),l2))
     length(l) = if empty(l)
                 then 0
                 else 1 + length(rest(l))
     occurences(e,l) = if empty(l)
                       then 0
                       else if first(l)=e
                            then 1 + occurences(e,rest(l))
                            else occurences(e,rest(l))
endspec
```

Fig.2: The specification LIST

```
spec SORT-PREDICATES
   use LIST, ORDELEM
   ops permutation? : list list → bool
       ordered? : list → bool
   props all l1, l2 ∈ list:
          (permutation?(l1,l2) ↔
            all e ∈ elem: occurences(e,l1)=occurences(e,l2))
        all l ∈ list: simple?(l) → ordered?(l)
        all l ∈ list: ¬simple?(l) →
              (ordered?(l) ↔
                   first(l) ≤ first(rest(l)) ∧ ordered(rest(l)))
endspec

spec SORT-AXIOM
   use SORT-PREDICATES
   ops sort : list → list
   props all l ∈ list:
             permutation?(l,sort(l)) ∧
             ordered? (sort(l))
endspec
```

Fig.3: What sorting should do...

```
spec SORT-PRIMITIVES
   use LIST
   ops part1, part2 : list → list
       combine : list list → list
   props all l ∈ list : ¬simple?(l) →
            (ordered?(part1(l)) ∧ ordered?(part2(l)) →
             ordered?(combine(part1(l), part2(l)))) ∧
            permutation?(combine(part1(l), part2(l)),l)
endspec

spec SORT-ALG
   use SORT-PRIMITIVES
   ops sort : list → list
spec body
  define ops
         sort(l) = if simple?(l)
                   then l
                   else combine(sort(part1(l)), sort(part2(l)))
endspec
```

Fig.4: ...and how it can be done

that provides operations and sorts with the required properties. LIST, however, is an algorithmic specification. The structure of the lists' carrier set is given by the constructor clause, and must consist of all the terms that can be constructed using the operation symbols empty and push (or any isomorphic set). The operations of LIST are defined in terms of the carrier's structure and of already defined operations.

```
spec SELECTION-SORT-PRIMITIVES
   use LIST
   ops  minlist, allbutmin, allbutone: list → list
           min: list → elem
spec body
   define ops
             min(L) = if simple?(L)
                      then first(L)
                      else let m= min(rest(L)) in
                           if first(L) ≤ m
                           then first(L)
                           else m
             minlist(L) = put(min(L),empty)
             allbutone(L,e) = if e=first(L)
                              then rest(L)
                              else
                              put(first(L),allbutone(rest(L),e))
             allbutmin(L) = allbutone(L,min(L))
endspec

spec SELECTION-SORT
   use SORT-ALG(SORT-PRIMITIVES → SELECTION-SORT-PRIMITIVES
                     ops part1 = minlist
                         part2 = allbutmin
                         combine = append)
endspec
```

Fig.5: Constructing selection sort

The first approach to solve the sorting problem is the formalization of the problem itself. In SORT-PREDICATES (Fig.3) we define what it means for a list being an ordered list, and in SORT-AXIOM we give a first specification of the sorting task: sorting a list means finding a permutation which is ordered.

The first algorithmic solution SORT-ALG (Fig.4) employes the well-known divide-and-conquer strategy ([Sm83],[VeLo80]). Although the sort operation is algorithmically defined, there is still a large variety of functions for its auxiliary operations part1/2 and combine, constrained only by the properties stated in SORT-PRIMITIVES. But it can be proved that for all possible instantiations of SORT-PRIMITIVES (and, hence, for all algorithmic definitions of part1/2 and combine fulfilling the constraints) the sort algorithm of SORT-ALG is a correct implementation of the sort operation specified by SORT-AXIOM.

Now we will perform one dedicated instantiation using SELECTION-SORT-PRIMITIVES (Fig.5). I.e. in SORT-ALG we replace part1 by minlist, part2 by allbutmin and combine by append (inherited from LIST), thus constructing the selection sort algorithm which can be described as "select the lowest element, put it in front of the list, and sort the rest". This instantiation can be viewed as a type of program transformation changing the data type. However, in the rest of the paper we shall not talk about transformations of whole abstract data types, but concentrate on the development of the sort operation in SELECTION-SORT towards an optimized tail-recursive version.

2.2 Semantics

Specifications

A specification is a pair $<\Sigma,P>$ where Σ is a signature and P is a set of formulas and constructive definitions.

Signatures, Σ-Algebras

A signature is a pair $<S,F>$ where S is a set of sort symbols, and F is a family $F = (F^{(w,s)})_{w \in S^*, s \in S}$ of function symbols. For each sort S there must be an error symbol $error_s \in F^{(\lambda,s)}$.
A Σ-Algebra A is an algebra with a flat c.p.o. A_s for every sort symbol s, and for every function symbol $f \in F^{(s1,...,sn,s)}$ a function $f_A : A_{s1}...A_{sn} \rightarrow A_s$.
Two additional conditions must hold:
1. $error_{sA}() = \omega_{A_s}$ is the bottom element of A_s.
2. All f_A are continuous. (Since our carriers are flat c.p.o.'s this is equivalent to monotonic. And a monotonic function is either strict or a constant function.)

Alg(Σ) is the category of all Σ-Algebras. It is called the abstract data type denoted by Σ.

Formulas

Let $P = \{p_1,p_2,...\}$ be a set of formulas in first-order predicate calculus, using only sorts and functions of $\Sigma = <S,F>$. A Σ-algebra is satisfying P if the interpretation of each p_i is true in A.

Constructive Functors

A constructive definition is a pair $<OP,\rho>$ where $OP = \{f_1...f_n\}$ is a set of recursive equations about these functions:
$\rho = \{f_i = \rho_i\ [f_1...f_n]\ |\ i=1..n\ \}$.
The semantics of a constructive definition $<OP,\rho>$ is the functor
sem $<OP,\rho>$: Alg (Σ) $\rightarrow$ Alg (Σ-OP)
where
  $\forall A \in$ Alg (Σ-OP) : sem $<OP,\rho>$ (A) = A $\cup$ $\{f_A\ |\ f \in OP \cap \Sigma\}$
    where $\rho_A : OP_A \rightarrow OP_A$ is the functional that is
      the result of the natural interpretation of ρ in A with
      parameter set OP
    and $\{f_A\ |\ f \in OP\}$ is the least fixpoint of ρ_A.
A Σ-Algebra A satisfies a constructive definition $<OP,\rho>$ if
  sem$<OP,\rho>$ (A $-$ $\{f_A\ |\ f \in OP\}$) $\cong$ A

Correctness of transformations

A transformation T is a mapping on constructive definitions. It is correctness preserving if it does not effect the semantics of a specification, i.e.
Alg $<\Sigma$, P $\cup$ $<OP,\rho>>$ $\cong$
Alg $<\Sigma$, P $\cup$ T$<OP,\rho>>$
This is true if for all Σ-Algebras A satisfying P the following proposition holds:
  sem $<OP,\rho>$ (A) $\cong$ sem T$<OP,\rho>$ (A)
Following the definition of sem it is sufficient to show that for all Σ-Algebras A satisfying P the least fixpoint of $T(\rho)_A$ is identical to the least fixpoint of ρ_A restricted to A.

Adding and transforming single recursive equations

The restriction just mentioned ensures that adding a new function to a constructive definition is a correct transformation. It will not effect the equations for the other functions. (The formal proof is omitted here. There is no great idea behind it.) In the next chapter we will study transformations that modify only the right hand side of one equation. Let it be the first one, then

$\rho = \{ f_i = \rho_i [f_1 \ldots f_n] \mid i=1..n \}$

is transformed to

$\rho' = \{ f_1 = \rho_1' [f_1 \ldots f_n]\} \cup$
$\{ f_i = \rho_i [f_1 \ldots f_n] \mid i=2 \ldots n\}$

The semantics of each equation $f_i = \rho_i [f_1 \ldots f_n]$ is a functional, and the semantics of a set of recursive equations is completely defined by the semantics of its equations. Hence the transformation above is correct if in any algebra A fulfilling P, the functionals ρ_{1A} and ρ'_{1A} have the same least fixpoint for all interpretations of $f_2 \ldots f_n$.
Many transformations are creating new operations and are expressing an old function in terms of the new operations. I.e.

$\rho = \{f_i = \rho_i [f_1 \ldots f_n] \mid i=1..n\}$

is transformed to

$\rho' = \{f_1 = \rho_1' [f_1 \ldots f_{n+m}]\} \cup$
$\{f_i = \rho_i [f_1 \ldots f_n] \mid i=2..n\} \cup$
$\{f_j = \rho_j [f_1 \ldots f_{n+m}] \mid j=n+1..n+m\}$

where $\forall j \in n+1..n+m \ \forall i \in 2..n : f_i \neq f_j$.
This transformation is correct if the least fixpoint of ρ' restricted to $f_1..f_n$ is equal to the least fixpoint of ρ. We may consider $\rho'-\rho$ as a functional mapping $[f_1 \ldots f_{n+m}]$ to $[f_1, f_{n+1} \ldots f_{n+m}]$. To prove the transformation $\rho \rightarrow \rho'$ correct we must show that f_1 in the least fixpoint of $\rho'-\rho$ is equal to the least fixpoint of ρ_1 (for all $f_2 \ldots f_n$ as described above).

3. Unfold-fold operations

There are two major ways to implement program transformations : The transformation rules can either be described in the form of algorithms, which take a given program as input and produce an equivalent one as output, or can be given as an ordered pair of templates together with an applicability condition denoting a conditional production rule. In our program transformation system both techniques have been employed, the former for the basic transformations unfold, fold etc., the latter for recursion removal (see Chapter 4).
The unfold-fold method has been developed by Burstall and Darlington [BuDa77]. It provides a small set of substitution rules as a formal tool for the stepwise refinement of functional programs.

1. Definition:
   Introduce an equation for a new function symbol.
2. Unfold:
   Replace a function application by the body of the applied function substituting formal by actual parameters.
3. Fold:
   The inverse operation to unfold. Replace the occurence of a function's body by an application of the function with the appropriate actual parameters.
4. Using laws:
   Use an equation for rewriting a term. In our specification environment the equations are provided by the algebraic portion of the specifications.
5. Abstraction:
   We may introduce a let-clause abbreviating a subterm of a definition by a new variable. This is a purely syntactical operation. It is correctness preserving by its definition.

Definition

The introduction of new functions has been handled in the previous chapter.

Unfold

A recursive definition is written as
$$F \equiv \tau[F]$$
where $\tau[F]$ is a composition of the function variable F and other function symbols which are considered as free variables all over the following proofs. We say that F is less defined than G - $F \leq G$ - if the following condition holds: $\forall x : F(x) \neq \omega \rightarrow G(x) = F(x)$. Since F,G are said to be equal - $F \equiv G$ - if for all x they have the same value or are both undefined, it is clear that
$$F \equiv G \leftrightarrow (F \leq G \wedge G \leq F)$$
F_τ is the least fixpoint of the functional τ if $F_\tau \equiv \tau[F_\tau]$, and, for any g, $g \equiv \tau[g]$ implies $F_\tau \leq g$.

Given two functionals $F \equiv \tau[F,G]$, $G \equiv \gamma[F,G]$, the result of unfolding τ with γ is the definition
$$F \equiv \tau[F,\gamma[F,G]]$$
Let us define $\tau_1[F,G] \equiv \tau[F,\gamma[F,G]]$.
To prove the correctness of the unfold operation we have to show that $F_\tau \equiv F_{\tau 1}$.

a) $F_{\tau 1} \subseteq F_\tau$:

$F_\tau \equiv \tau[F_\tau, G_\gamma]$ (F_τ is fixpoint)
$\equiv \tau[F_\tau, \gamma[F_\tau, G_\gamma]]$ (G_γ is fixpoint)
$\equiv \tau_1[F_\tau, G_\gamma]$

I.e. F_τ is a fixpoint of τ_1. Since $F_{\tau 1}$ is the least one, we have $F_{\tau 1} \subseteq F_\tau$.

b) $F_\tau \subseteq F_{\tau 1}$

We will show that $P(F_\tau, F_{\tau 1}, G_\gamma)$ holds where
$P(F, F_1, G) \equiv F \subseteq F_1 \wedge F \subseteq \tau[F,G] \wedge G \subseteq \gamma[F,G]$
using the computational induction method ([MNV72]). This method is valid only if γ and τ are continuous functionals, but this is ensured by the semantics of our specification language.
Starting the induction we have to show $P(\Omega,\Omega,\Omega)$ where Ω is the never defined function: Since τ and γ are continuous this is true.
Then we must prove
$\forall F, F_1, G: P(F, F_1, G) \rightarrow P(\tau[F,G], \tau_1[F,G], \gamma[F,G])$
First inclusion:

$\tau_1[F,G] \equiv \tau[F, \gamma[F,G]]$ (Definition of τ_1)
$\supseteq \tau[F,G]$ (Ind. Hyp., continuity)

The other two inclusions follow immediately since τ and γ both are known to be continuous.

<u>Fold</u>

Given two functions $F \equiv \tau[F,G]$, $G \equiv \gamma[F,G]$, folding F with G is possible only if there is a functional τ_1 with
$\tau[F,G] \equiv \tau_1[F, \gamma[F,G]]$
The result of folding is the definition
$F \equiv \tau_1[F,G]$

a) $F_{\tau 1} \subseteq F_\tau$
This proposition ensures partial correctness of folding. When renaming τ_1 to τ and vice versa it is identical to the statement b) in the previous proof.

b) You may loose termination when applying the fold operation. But there is a simple condition ensuring that this will not happen: γ must be independent of F, i.e.
$\forall F_1, F_2, G: \gamma[F_1, G] \equiv \gamma[F_2, G]$.
With this condition we can show that $F_\tau \subseteq F_{\tau 1}$:

$F_{\tau 1} \equiv \tau_1[F_{\tau 1}, G_\gamma]$ ($F_{\tau 1}$ is fixpoint)
$\equiv \tau_1[F_{\tau 1}, \gamma[F_\tau, G_\gamma]]$ (G_γ is fixpoint)
$\equiv \tau_1[F_{\tau 1}, \gamma[F_{\tau 1}, G_\gamma]]$ (γ is independent of F)
$\equiv \tau[F_{\tau 1}, G_\gamma]$ (definition of τ_1)

I.e. $F_{\tau 1}$ is a fixpoint of τ. Since F_τ is the least one we have $F_\tau \subseteq F_{\tau 1}$.

<u>Using Laws</u>

The application of laws is correct if they come from the specification which is the source of the constructive definition.
Given
- a specification $<\Sigma, P>$
- a formula $\forall x_1 \ldots x_n : t_1 (x_1 \ldots x_n) \equiv t_2 (x_1 \ldots x_n)$ in P
- and an equation $f_i \equiv \varphi_i (f_1, f_2 \ldots)$

we may substitute t_1 by t_2 in φ_i and vice versa without modifying the semantics of φ_i in any Σ-Algebra satisfying P. (Note, that by definition '$\equiv$' denotes the strong equality where $\omega \equiv \omega$ is true.)

Remark

A quite different approach has been taken by [Ko82]. His work is based on the algebraic semantics of recursive programs, and contains some interesting results about unfold-fold operations. Since he is considering continuous functions on partially ordered sets as we do, we may apply some of his results to our program transformation framework:
Consider a sequence of unfolds, followed by applications of laws, and finally some foldings, all together modifying only one recursive equation $F \equiv \tau[F_1 \ldots F_n]$.

a) If the last folding is not using τ then the sequence of transformations is totally correct.
b) If the last folding is using τ, and this is the only folding with τ in the whole sequence, and the number of all unfolds is greater than or equal to the number of all folds, then the transformation sequence is correct, too. This proposition is true only if all functions are strict. Following the notions on Σ-Algebras in chapter 2 this is true, as long as no constant functions are involved in the transformation sequence.

Example

Let us have a look on our example now. We start with the definition of sort in SELECTION-SORT:

```
      sort(l)= if simple?(l) then l
               else append (sort (minlist (l)),
                            sort (allbutmin (l)))
```

All the following transformations apply only to the term sort(minlist(l)):

```
     sort (minlist (l))
[unfold sort]
   = if simple? (minlist(l)) then minlist (l)...
[unfold simple?]
   = if (if empty? (minlist (l))
         then true else empty? (rest (minlist (l)))...
[unfold minlist]
   = if (if empty? (put (l,empty))
         then true else empty? (rest (minlist (l)))...
[apply law (empty?(put(x,y)) = false)]
   = if (if false then true else empty? (rest (minlist (l)))...
[apply law (if false then x else y = y), unfold minlist]
   = if empty? (rest (put (l,empty))) then minlist (l)...
[apply law (rest(put(x,y)) = y) and (empty?(empty) = true)]
   = if true then minlist (l)...
[apply law (if true then x else y = x)]
   = minlist (l)
```

Hence, this sequence transforms the cascaded recursion (two recursive calls of sort) into a linear recursion. In the next chapter we will go one step further by converting this definition into tail recursion, and use the unfold-fold method again to simplify the result.

4. Recursion removal using second-order patterns

4.1 Overview

Let us start with a classification of recursive functions. A function is called recursive if it is defined using itself. If the body of a procedure f contains a term like f(...f...) we have a nested recursion. The best known example of this type of recursion is the 91-function ([MNV 72]):

$f_{91}(x) = \underline{if}\ x>100\ \underline{then}\ x-10\ \underline{else}\ f_{91}(f_{91}(x+11))$

If f contains several calls of f and is not nested we call the recursion cascaded. The definition of sort in Fig.4 shows a cascaded recursion. A recursion that is neither nested nor cascaded is linear. C.f. the sort definition that is the result of the previous chapter. A linear recursion is in iterative form if the recursive call is the dynamically last action in it's body. Iterative recursions can be translated to loops in a very simple manner according to the following rule:

```
   f(x):= if b(x) then f(g(x)) else h(x)
                      |
   -------------------+-------------------
                      ↓
   f(x):= while b(x) do x:= g(x); h(x)
```

This transformations, however, can not be performed on the specification level, since loops and assignments are not allowed in our purely functional language. Indeed, our system supports the Pascal implementation of algorithmic specifications, and during that implementation iterative recursions may be replaced by loops. Thus the generation of iterative recursions from more complex functions on the abstract level results in programs written in a procedural language, and less time and space consuming than recursive functions.
Each recursive procedure can be transformed to an equivalent non-recursive procedure as any Lisp and Pascal compiler does. Those compiler generate procedures, however, are using a stack for storing parameter values and return addresses. I.e. the program's recursivity is merely transferred to the data structure.
But what's about the class of functions that can be transformed to iterative functions which use only a fixed number of storage allocations? [WaSt73] call those functions "flowchartable" and show some general results about flowchartability:
1. Each linear recursion is flowchartable.
2. The general scheme for a cascaded recursion is not flowchartable.
3. The flowchartability of a given procedure is undecidable.

However, there are some critical notions about those results:
- The general rule to flowchart linear recursions [Pat70] leads to very inefficient calculations. Therefore an implementation using a stack must be preferred in the general case. Storing the return addresses is unnecessary with linear recursions.
- When developing a system for the automated elimination of recursions it is very important to use knowledge about the operations occuring in a given procedure. With knowledge like this it is often possible to flowchart even complicated recursions without any stack or at least with a significantly smaller one.

- A compiler can implement the stack operations that are associated with a function call very efficiently because it can use machine instructions. Therefore the usage of a stack on the abstract level will increase the efficiency of the resulting program only if the stack becomes much simpler than the full compiler generated one.

The unfold/fold method can be employed for recursion removal, too. In this chapter, however, we follow an approach based on second-order patterns (c.f. [HuLa78]). A transformation rule is a tripel <Σ,X,Σ'> containing:
- 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.

Σ, X, Σ' are terms in a second-order language which is described in full detail in [Gerl83]. Using denotational semantics we can prove transformation rules correct by the inductive methods (especially computational induction) given in [MNV72].
The starting point in the development of our rule data base was the collection of rules provided by [BaWö81] for the CIP-L language. [Pet83] performed the correctness proofs, found some generalizations and brought the rules together in a production-rule like, though semi-formal representation. [Geiss84] adapted these rules to our specification language ASPIK and put them into a formal network representation (see below) s.t. it is possible now to perform these transformations automatically in our specification environment.

Our knowledge base contains rules for the simplification and elimination of linear, cascaded and nested recursions. In the appendix we show the complete set of rules for eliminating linear recursions. The rules for cascaded and nested recursion, however, are not presented here since they are rather complex (see the papers cited above for all details).
There are several rules known from literature which are not part of our knowledge base:
1. [Ar79], [Au78], [Bi77] and [Ro80] are based on procedural languages, [GaLu], [PaHe70] and [St71] use flowcharts as the goal "language". So they are outside the scope of our interest.
2. Special transformation rules for arithmetic functions have been developed by [PaPe76] and [Hi79] and are a topic for our future work.
3. The first rules for recursion removal were published in [Co66]. He investigated the ideas being the background of the rules pertaiming to linear recursion.
4. A program that automatically transforms some classes of recursive LISP-functions into non-recursive ones is described in [Ri73]. His transformations can be viewed as instantiations of the general rules described here using the semantics of special LISP-functions.

4.2 Examples

The following two examples will demonstrate the transformation method described above. The first example will give an idea how functions with more than one parameter are handled. The function to be transformed is very simple:

```
   x*y  = if y≠0
          then x+(x*(y-1))
          else 0
```

And here is the appropriate transformation rule:

```
F(m) = if B(m) then Φ(F(k(m)),E(m)) else H(m)
```

$$\Big\downarrow \qquad \exists \Psi \forall r,s,t:\ \Phi(\Psi(r,s),t) = \Psi(\Phi(r,t),s) \wedge \exists c \forall m:\ H(m)=c \wedge \Phi(c,m) = \Psi(c,m)$$

```
F(m) = G(m,c)
G(m,n) = if B(m) then G(K(m), Ψ(n,E(m))) else n
```

The variable bindings for this example are:

m	→ x,y
F	→ *
B	→ λuv. v≠0
Φ	→ λuv. v+u
K	→ λuv. u, λuv. v-1
E	→ λuv. u
H	→ λuv. 0

And we see that the conditions are fulfilled for $\Phi=\Psi$ and $c=0$. Hence we gain the iterative result

```
x*y      = G(x,y,0)
G(x,y,n) = if y≠0
           then G(x,y-1, x+n)
           else n
```

Remarks:

1. The reader should not be confused about the mixture of infix and prefix notations. The transformations are performed on an internal representation which is prefix, but the user can communicate with the system in the mixed notation via a sophisticated interface.
2. B, Φ, K, E and H are second-order variables. Hence their values are λ-terms, denoting functions.
3. The variables m and K are so-called "multivariables" (introduced in [Gerl83]) which can bind multiple values, separated by commas in the table above.
4. Remember: In a procedural language the result can be expressed using loops. Many compilers perform that last transformation step automatically.

The second example continues the development of the selection-sort operation. The result of chapter 3 was:

```
sort(l) = if simple?(l) then l
          else append (minlist(l), sort(allbutmin (l)))
```

This is a linear recursion that can be eliminated using the following rule:

```
F(m) = if B(m) then Φ(F(K(m)), E(m)) else H(m)

                      ∀r,s,t: Φ(Φ(r,s),t) = Φ(r,Φ(s,t))
                      ∧ ∃e∀r: Φ(r,e) = r

F(m) = G(m,e)
G(m,n) = if B(m) then G(K(m), Φ(E(m),n))
           else Φ(H(m),n)
```

Here Φ matches append which is known to be associative and to have empty as a neutral element. So we may apply this rule, and gain an iterative version of sort:

```
sort(l) = sort1(l, empty)
sort1(L,K) = if simple?(L) then append(K,l)
               else sort1(allbutmin(l), append(K,minlist(l)))
```

Remarks:

1. The rule application module contains a normalization procedure. Among other tasks it must transpose the conditional branches of the input function when the recursion is in the else part, and perform the inverse operation on the result.
2. The algebraic knowledge about the function append comes from the specification LIST (Fig.2). Thus the rule application module has direct access to the specification environment.
3. The unfold/fold method can be used again for a further optimization of the sort1 definition above by transforming the else-part. Unfolding allbutmin and minlist and abstracting min(l) yields:

```
   sort1(L,K) = if simple?(L) then append (K,L) else
                     let m= min(l) in
                       sort1(allbutone(L,m),
                             append(K,put(m,empty)))
```

4.3 The Correctness of Second-order rules

Let us describe a second-order rule by a tupel $<\varphi, \varphi', p>$ where
- φ is a recursive equation
- p is a logical formula
- φ' is a set of recursive equations.

φ, φ' and p contain first-order and second-order variables. A substitution is a mapping from variables to terms and λ-abstractions. Performing a substitution σ on a recursive equation φ means substituting all variables and evaluating applications of λ-abstractions. (A formal treatment of second-order substitution and matching has been performed in [Gerl83])
Applying a rule $<\varphi, \varphi', p>$ to a specification $<\Sigma, P \cup \{<OP,\varphi>\}>$ is defined as follows:

1. Find a substitution σ s.th. there is a recursive equation $f_i \equiv \varsigma_i [f_1 ... f_n]$ in ς equal to $\sigma\varphi$, and σp can be deduced

from P.
2. Replace $\{\sigma\}$ in φ by $\sigma\varphi'$, and extend OP appropriately.

According to the general considerations of chapter 2 this transformation is correct if
1. $\{f_j \mid \exists \rho_j: f_j \equiv \rho_j[...] \in \sigma\varphi'\} \cap OP = \{f_i\}$
   i.e. only f_i is redefined
2. f_i in the least fixpoint of $\sigma\varphi'$ is equal to the least fixpoint of $\sigma\varphi$

To prove the correctness of a transformation rule we have to show that for all variable bindings the least fixpoint of φ is equal to the corresponding component of the least fixpoint of φ'. (The first condition above can be ensured by a generator mechanism, and is only a technical issue.)

<u>Example</u>

The following well-known rule ([BaWö81],[Co66]) transforms a class of linear recursive functions to tail-recursive definitions:

φ = $f(m) \equiv$ <u>if</u> $m \neq c$ <u>then</u> $\Phi\ (f(k(m)),\ E(m))$ <u>else</u> $H(c)$
p = $\forall r,s,t:\ \Phi(\Phi(r,s),t) = \Phi(\Phi(r,t),s)$
φ' = $\{f(m) \equiv g(m,\ H(c)),$
      $g(m,z) \equiv$ <u>if</u> $m \neq c$ <u>then</u> $g(K(m),\ \Phi(z,E(m))$ <u>else</u> $z\ \}$

We can apply it to the square function:
$sq(x) \equiv$ <u>if</u> $x \neq 1$ <u>then</u> $(2x-1)+sq(x-1)$ <u>else</u> 1
with the variable bindings
$f \to sq$, $m \to x$, $c \to 1$,
$\Phi \to \lambda uv.\ v+u$, $K \to \lambda u.\ u-1$,
$E \to \lambda u.\ 2u-1$, $H \to \lambda u.u$

The condition p is the commutativity of + in this example. The transformation result is
$\{sq(x) \equiv g(x,1),$
 $g(x,z) \equiv$ <u>if</u> $x \neq 1$ <u>then</u> $g(x-1,\ (2x-1)+z)$ <u>else</u> $z\ \}$

The semantics of '=' used above is the weak equality, i.e. $(\omega=\omega) \equiv \omega$ to ensure that the defined functions become continuous. And the semantics of <u>if</u> is defined as usual:
<u>if</u> true <u>then</u> a <u>else</u> b $\equiv$ a
<u>if</u> false <u>then</u> a <u>else</u> b $\equiv$ b
<u>if</u> ω <u>then</u> a <u>else</u> b $\equiv \omega$
where a and b may be any defined value or ω.

Now we will prove the correctness of the rule using computational induction on the predicate Q:
$Q(f,g) \equiv \forall m:\ f(m)=g(m,H(c)) \wedge$
     $\forall r,s,t:\ \Phi(g(r,s),t)=g(r,\Phi(s,t))$
We want to show that $Q(f_\varphi,\ g_{\varphi}')$ holds where f_φ is the least fixpoint of φ and g_φ' the least fixpoint of the second equation of φ'.
a) $Q(\Omega,\Omega)$:
   This is true because all functions are supposed to be monotonic.
b) $\forall f,g:\ Q(f,g) \to Q(\varphi[f],\ \varphi'[g])$
   First equation
   $\varphi[f](m)$
   [Definition of φ]
    $=$ if $m \neq c$ then $\Phi(f(K(m)),\ E(m))$ else $H(c)$

[Induction hypthesis, first equation]
 = if $m \neq c$ then $\Phi(g(k(m), H(c)), E(m))$ else $H(c)$
[Induction hypothesis, second equation]
 = if $m \neq c$ then $g(K(m), \Phi(H(c), E(m)))$ else $H(c)$
[Definition of φ']
 = $\varphi'[g](m,H(c))$

Second equation
$\Phi(\varphi'[g](r,s),t)$
[Definition of φ']
 = $\Phi($if $r \neq c$ then $g(K(r), \Phi(s,E(r))$ else $s, \; t)$
[Semantics of if, monotony of Φ]
 = if $r \neq c$ then $\Phi(g(K(r), \Phi(s,E(r)),t)$ else $\Phi(s,t)$
[Induction hypothesis, second equation]
 = if $r \neq c$ then $g(K(r), \Phi(\Phi(s,E(r)),t))$ else $\Phi(s,t)$
[Condition p]
 = if $r \neq c$ then $g(K(r), \Phi(\Phi(s,t),E(r)))$ else $\Phi(s,t)$
[Definition of φ']
 = $\varphi'[g](r,\Phi(s,t))$

./.

The first equation of the predicate Q implies the correctness of the transformation rule:
$\forall m$: $f_\varphi(m) = g_{\varphi}'(m,H(c)) = f_{\varphi}'(m)$ [Definition of φ']

4.4 Knowledge Representation

How to represent the rules in the transformation system? One could think of writing a production rule system in a straight forward manner, but this is not the best way for the following reasons:

1. We must minimize the number of matches being performing during a transformation since our terms contain second-order variables. Matching a definition and a second-order pattern takes typically one second of CPU time.
2. The table of transformation rules shown in the appendix exhibits the inherent structure that should be exploited during the transformation process.
3. Side effects have to be performed when transforming a definition (generating new names for auxiliary functions, creating data types for stacks and tupels). This can be expressed more naturally in a procedural framework.

Hence, we use a procedural network for the representation of our rule data base. The nodes denote states during the application of a transformation, the arcs are labelled with the tasks to be performed during a transformation step, c.f. matching terms, testing conditions or creating the result.
The formalism has been taken from Wood's Augmented Transition Networks [Wo 70], but we use only the syntactical frame. Some concepts essential for ATN processing, as consuming an input stream, and the recursive call of subnets, have been omitted in our implementation (in ATN terminology: we have only JUMP arcs, and the input is given as an initial register setting). However, the concept of registers - setting a register on one arc and using it within an other - has heavily been used.

The syntax for a network is quite simple:

```
net :: (state*)
state :: (name arc*)
arc :: (condition action* goal)
```

The arcs' components have been developed for our specific purposes. Important conditions are:
- matching a register's value with a second-order pattern. As a side effect the variable bindings resulting from the matching process are stored as register values.
- calling a deduction component to prove a proposition, or to solve an existentially qualified equation.
- testing for any other property of register values (with the full power of LISP).

Actions are:
- generating new names for variables or auxiliary functions, creating data types (specifications) for stacks or tupels
- generating parts of the final result

The goal of on arc is:
- either the name of the state where processing must continue
- or a pattern made of register names and constants, denoting the final result of the transformation.

Fig.6 shows a cut of the network in the internal representation just described. By convention, processing starts with the first state (INIT), the definition to be transformed is assumed to be the value of the register INPUT. The names of variables correspond to their type: The number of leading '?' expresses their order, the '*' afterwards marks multivariables.

Algorithms

A second-order pattern matcher is described in [Gerl83]. It elaborates [HuLa78] in the way that multivariables (see above) are handled, and some heuristics have been incorporated to ensure that in most cases the desired variable bindings (which may be one of several possible matches) are found first.
Before matching we have to normalize the input. It has the following effects [Geiss84]:
- If there are several recursive calls in mutually exclusive conditional branches we can replace them by one recursive call putting the conditionalization into an auxiliary function.
- Recursive calls are moved from the else- to the then-part by negating the condition
- Each "Let" surrounding a recursive call is replaced substituing the bindings
- Each "Case" is replaced by equivalent conditionalizations

Checking conditions performed in different ways:
- Searching specifications for appropiate algebraic properties
- Accessing a knowledge base specific for the program transformation module. Of course, we must ensure the consistency of this knowledge base and the specification environment.
- Deducing the desired properties using the unfold/fold technique (chapter 3)
- Proving the condition via the automated theorem prover ([KA 84]). However, this is only possible for first-order propositions.

```
((INIT
 ((MATCH ((?F ?*M)
          (IF (??B ?*M)
              (??PHI (?F (??*K ?*M))
                     (??*E ?*M))
              (??H ?*M)))
             INPUT)
  (TO LIN)) ... )

 (LIN
 ((EQUAL ??*E ())
  (SETV ??PSI ??PHI)
  (TO LIN-COMMUTING))
 ((AND (NEWVARS ?R 1)
       (NEWVARS ?S (LENGTH ??*E))
       (NEWVARS ?T (LENGTH ??*E))
       (EX ??PSI ALL ?R ?S ?T
          (??PHI (??PSI ?R ?S) ?T) =
          (??PSI (??PHI ?R ?T) ?S)))
  (TO LIN-COMMUTING)) ... )

 (LIN-COMMUTING
 ((EX ??C ALL ?*M
     (AND (??H ?*M) = (??C)
          (??PHI (??C) ?*M) = (??PSI (??C) ?*M)))
  (GEN-OP ?F1 (SORT-OF ?*X)(SORT-OF ?F)(SORT-OF ?F))
  (GEN-VAR ?X1)
  (RETURN (((?F ?*X)
            (?F1 ?*X (??C)))
           ((?F1 ?*X ?X1)
            (IF (??B ?*X)
                (?F1 (??*K ?*X)
                     (??PSI ?X1 (??*E ?*X))
                      ?X1))))))
..... ))
```

Fig.6: A cut of the network representation

5. Interfaces

The program transformation system described in this paper is a part of the specification system SPESY ([BGGORV84], [So84], [Ma84]). SPESY includes the following facilities:

- A database for ASPIK specifications, procedural programs written in ModPascal [OLt84], and maps (i.e. relations) between those objects
- Syntax-oriented editors for specifications, ModPascal programs and maps
- Access to tools for validation and verification: Symbolic interpreter for specifications, automated theorem prover, rewrite rule laboratory etc.

This system has been developed in InterLisp on a large time-sharing computer providing only a simple teletype interface.

The program transformation module is entered by typing

```
    PT <specname>
```

Then you are alloved to manipulate the specification with the given name via the following commands (implemented by [BeWo84], except ELIM):

FOLD, opid1, opid2, n
  Replace the nth occurrence of the body of opid2 in the definition of opid1 by an application of opid2

UNFOLD, opid1, opid2, n
  Replace the nth occurence of an application of opid2 in the definition of opid2 by the body of opid2

BIND, opid, var=term
  Abstraction: Insert a let-clause in the definition of opid, abbreviating term by var.

UNBIND, opid, var
  Unabstraction: Remove the let-clause binding var.

SHOWLAWS, opid1, opid2
  List all equations that 1. are valid in the current specification environment, 2. contain opid2, 3. match any term inside the definition of opid1. Mark each of those equations by a unique number.

USELAW, opid, n, l
  Apply the equation with number l to the nth matching subterm of the definition of opid

ELIM, opid
  Transform the definition of opid to tail recursion using the methods described in chapter 4. This command may create new functions, or may have no effect at all if none of the available rules applies.

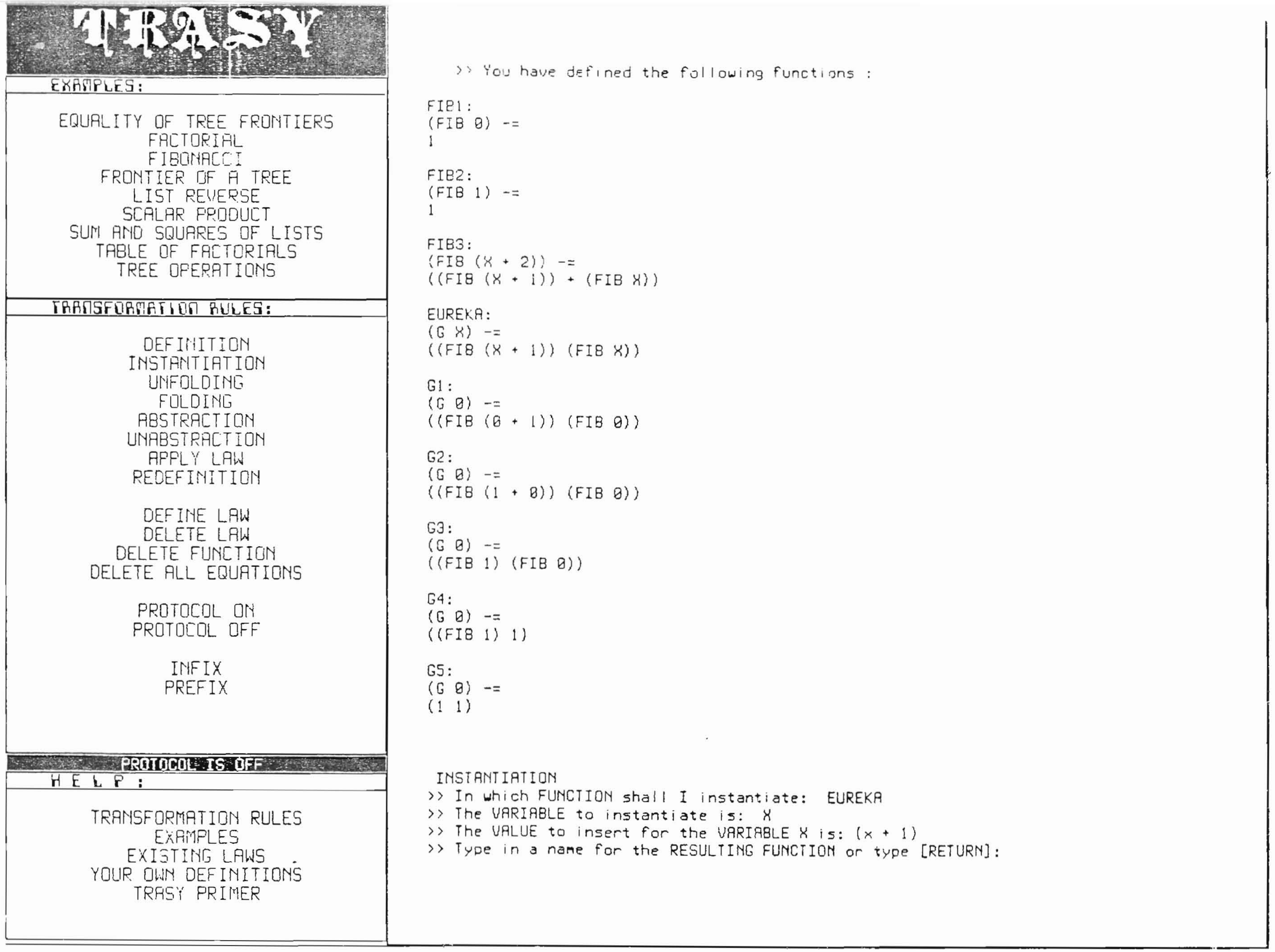
We want to stress the experimental character of program transformations. I.e. one does not know whether a started sequence of transformations will lead to a satisfactory result. So you want to be able to undo transformations, or to restart the transformation process at any earlier stage of the development.
We implemented this facility by storing all versions of all transformed functions s.th. for every function there is a history list containing all the versions of that function. There is an "actual version" defaulting to the newest, but there are commands to let another version being the actual one. The transformation commands above always take the actual versions as their input, and append their result to the end of the history list. Afterwards the transformation result is the new actual version.

There is a quite different implementation of a program transformation system ([Ste83]). It is a standalone system, i.e. the algorithmic definitions and equations do not come from the specification environment SPESY, but are created within the program transformation system. Its main task was to explore the possibilities of man-machine interaction using advanced input-output devices: A bitmap-display, highly sophisticated system software for dividing the screen into different interaction areas (windows), and the mouse, a device for selecting items displayed on the screen, e.g. starting commands by pointing at their names.
The screen contains the following windows (Fig.7):
- A menue of prepared examples
- The menue of the available commands
- A menue of several help facilities
- The main interaction area showing the already defined functions

You may enter commands in two different ways:
a) Prefix order: Select a command from a menue, then select the arguments for this commands.
b) Postfix order: Select a term to be modified, then select the type of transformation.

Two examples:
a) Select the unfold command, then point to the function call to be unfolded, or
b) select a function call in the definition window. Then a small temporary window appears offering operations that may be applied to the selected term: Folding, unfolding, abstraction etc.

Note that the UNFOLD command in the first implementation has three arguments. In the screen oriented interaction mode, however, one argument is enough: Selecting a term from the sreen uniquely determines the three arguments necessary with the teletype interface.
This prototype implementation of a program transformation system has shown the advantages of an integration of screen-oriented editing and formal program transformation. Additionally, the importance of ergonomic issues (c.f. [Ba83]) has been confirmed. The quality of the user interface is essential for the efficiency and acceptance of any software development system.



```
   >> You have defined the following functions :

FIB1:
(FIB 0) -=
1

FIB2:
(FIB 1) -=
1

FIB3:
(FIB (X + 2)) -=
((FIB (X + 1)) + (FIB X))

EUREKA:
(G X) -=
((FIB (X + 1)) (FIB X))

G1:
(G 0) -=
((FIB (0 + 1)) (FIB 0))

G2:
(G 0) -=
((FIB (1 + 0)) (FIB 0))

G3:
(G 0) -=
((FIB 1) (FIB 0))

G4:
(G 0) -=
((FIB 1) 1)

G5:
(G 0) -=
(1 1)


 INSTANTIATION
>> In which FUNCTION shall I instantiate:  EUREKA
>> The VARIABLE to instantiate is:  X
>> The VALUE to insert for the VARIABLE X is: (x + 1)
>> Type in a name for the RESULTING FUNCTION or type [RETURN]:
```

Fig.7: A program transformation interface

6. Conclusion

The specification language ASPIK sketched in this paper provides a uniform framework for both algebraic properties and algorithmic definitions. Thus it provides an appropriate base for program transformations which always rely on algebraic and algorithmic knowledge. A program transformation system has been implemented which employes the unfold-fold-method as well as the application of conditionalized second-order production rules. Both concepts may be combined arbitrarily thus providing a powerful development tool. The embedding in the specification environment guarantees that those transformations will not destroy the correctness of the manipulated algorithms.

Appendix: Rules for eliminating linear recursion

The general pattern for linear recursive functions is

$F(x) = \underline{if}\ B(x)\ \underline{then}\ \Phi(F(K(x)),\ E(x))\ \underline{else}\ H(x)$

The arities of the admissible functions are:

$F: \lambda_1 \dots \lambda_n \rightarrow \rho$
$B: \lambda_1 \dots \lambda_n \rightarrow \mathrm{bool}$
$\Phi: \rho\ \gamma_1 \dots \gamma_m \rightarrow \rho$
$K: \lambda_1 \dots \lambda_n \rightarrow \lambda_1 \dots \lambda_n$
$E: \lambda_1 \dots \lambda_n \rightarrow \gamma_1 \dots \gamma_m$
$H: \lambda_1 \dots \lambda_n \rightarrow \rho$

The rules (see Fig.8):

1.1 $\exists \Psi: \rho\ \gamma_1 \dots \gamma_m \rightarrow \rho$
$\Phi(\Psi(r,s),t) = \Psi(\Phi(r,t),s)$
This includes the simple case m=0: $\Phi(\Phi(r)) = \Phi(\Phi(r))$

```
1.1.1 $H(x) \equiv const.$ :
      $F(x) = F_1(x, const)$
      $F_1(x,y) = \underline{if}\ B(x)$
                  $\underline{then}\ F_1(K(x),\ \Psi(y,E(x)))$
                  $\underline{else}\ y$

1.1.2 $B(x) \equiv x \neq const$ :
      $F(x) = F_1(x, H(const))$
      $F_1$ as 1.1.1

1.1.3 $F(x) = F_1(x, F_2(x))$
      $F_2(x) = \underline{if}\ B(x)\ \underline{then}\ F_2(K(x))\ \underline{else}\ H(x)$
      $F_1$ as 1.1.1
```

1.2 $m=1 \wedge \exists \Psi: \Phi(\Phi(r,s),t) = \Phi(r,\Psi(s,t))$
$\Psi: \gamma_1\ \gamma_1 \rightarrow \gamma_1$

```
1.2.1 $\Phi \equiv \Psi \wedge \exists e\ \forall x: \Phi(x,e) = x$ :
      $F(x) = F_1(x,e)$
      $F_1(x,y) = \underline{if}\ B(x)$
                  $\underline{then}\ F_1(K(x),\ \Phi(E(x),y))$
                  $\underline{else}\ \Phi(H(x),y)$

1.2.2 $F(x) = \underline{if}\ B(x)\ \underline{then}\ F_1(K(x),E(x))\ \underline{else}\ H(x)$
      $F_1(x,y) = \underline{if}\ B(x)$
                  $\underline{then}\ F_1(K(x),\Psi(E(x),y))$
                  $\underline{else}\ \Phi(H(x),y)$
```

1.3 $\exists R : R(K(x)) = x$

```
1.3.1 $B(x) \equiv x \neq const$ :
      $F(x) = F_1(const, H(const), x)$
      $F_1(x,y,z) = \underline{if}\ x \neq z$
                    $\underline{then}\ F_1(R(x),\Phi(y,E(R(x))),z)$
                    $\underline{else}\ z$

1.3.2 $F(x) = F_1(F_2(x),H(F_2(x)),x)$
      $F_2(x) = \underline{if}\ B(x)\ \underline{then}\ F_2(K(x))\ \underline{else}\ x$
      $F_1$ as 1.3.1
```

```
1.4 F(x)    = let <s,y> = F_1(x,empty) in
                  F_2(s,y)
    F_1(x,s) = if B(x)
               then F_1(K(x),push(s,x))
               else <s,H(x)>
    F_2(s,y) = if ¬empty(s)
               then F_2(pop(s),Φ(x,E(top(s))))
               else x
```

The variable s is of type λ-stack with the following operations:

```
empty: → λ-stack
push   : λ-stack λ_1 ... λ_n → λ-stack
      pop
      : λ-stack → λ_1 ... λ_n
      empty?: λ-stack → bool

      <.,.> is a tupling operation with arity
      λ-stack ϱ → λ-stack-ϱ-tupel
```

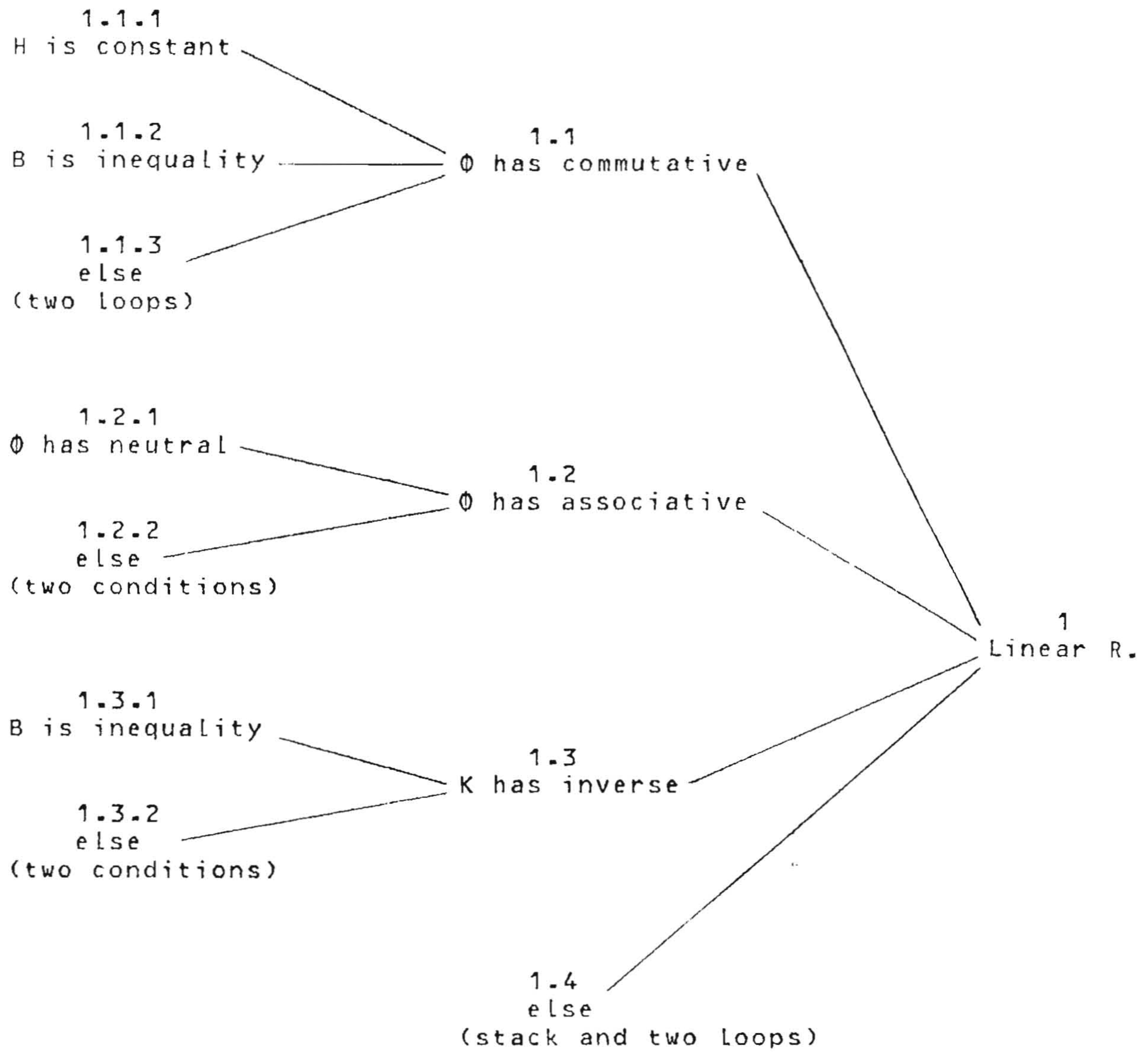


Fig.8: The decision tree for linear recursion

Bibliography

[Ar79] Arsac, J.:
Syntactic Source to Source Transforms and Program Manipulation.
In: CACM 22,1 (1979), pp.43-53

[ArKo82] Arsac, J., Kodratoff:
Some Techniques for Recursion Removal from Recursive Functions.
In: ACM Tr. on Prog. Lang. and Systems, Vol.4, No.2, April 1982, pp.295-322

[Au78] Auslander, M.A., Strong, H.R.:
Systematic Recursion Removal.
In: CACM 21,2 (1978), pp.127-134

[Ba76] Balzer, R., Goldman,N., Wile, D.:
On The Transformational Implementation Approach To Programming.
In: Proc. of the Second Int. Conf. on Software Engeneering, 1976

[Ba79] Bauer, F.L., Broy, M., Partsch, H., Pepper, P., Wössner, H.: Systematics of Transformation Rules
In: Bauer, F.L., Broy, M. (Ed.): Program Construction, Springer 1979

[Ba83] Balzert, H. (Ed.):
Software-Ergonomie.
Teubner Verlag, Stuttgart, 1983

[BaWö81] Bauer, F.L., Wössner, H.:
Algorithmic Language and Program Development
Springer, New York, 1981

[BeKo84] Becker, R., Koch, P.:
Implementierung eines Programmtransformationsmoduls fUr ein Spezifikationssystem.
Studienarbeit, FB Informatik, Universitaet Kaiserslautern, 1984

[BeVo83] Beierle, Ch., Voss, A.:
Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types.
SEKI-Projekt, MEMO SEKI-83-07, University of Kaiserslautern, FB Informatik, 1983.

[Bi77] Bird, R.S.:
Notes on recursion elimination.
CACM, Vol.20, No.6, pp.434-439
New York, 1977

[BuDa77] Burstall, M., Darlington, J.:
A Transformation System for Developing Recursive Programs.
in: Journal of the ACM.
Vol.24, No1, January 1977. pp.44-67.

[BGGORV84] Beierle, Ch., Gerlach, M., Göbel, R., Olthoff, W., Raulefs, P., Voss, A.:
Integrated Program Development And Verification.
in: H. L. Hausen (ed.): Symposium on Software Validation, North-Holland Publ. Co., Amsterdam 1984.

[BGV83] Beierle, Ch., Gerlach, M., Voss, A.:
Parameterization without Parameters in: The History of a Hierarchy of Specifications.
SEKI-Projekt, MEMO SEKI-83-09
University of Kaiserslautern, 1983.

[BroPe81] Broy, M., Pepper, P.:
Program Development as a Formal Activity.
In: IEEE Transactions On Software Engineering.
Vol. SE-7. No.1, January 1981. pp.14-22.

[Co66] Cooper, D.C.:
The equivalence of certain computations.
In: Computing Journal.
9/1966. pp.45-52.

[DaBu73] Darlington, J., Burstall, R.M.:
A System which Automatically Improves Programs.
In: Proc. of the 3rd Int. Conf. on Artificial Intelligence, 1973

[Fe82] Feather, M.S.:
A System for Assisting Program Transformation.
In: ACM Tr. on Prog. Lang. and Systems, Vol.4, No.1, January 1982, pp.1-20

[GaLu72] Garland, S.S., Luckham, D.C.:
Translating Recursion Schemes into Program Schemes.
In: SIGPLAN Notices, Vol.7, No.1, 1972

[Geiss84] Geissler, Ch.:
MARE - Methodischer Ansatz der Rekursions-Elimination,
SEKI-Projekt, MEMO SEKI-84-11
University of Kaiserslautern, 1984

[Gerh75] Gerhart, S.L.:
Correctness-Preserving Program Transformations.
In: Conf. Record of the Second ACM Symposium on Principles of Prog. Lang.
Palo Alto, 1975

[Gerl83] Gerlach, M.:
A Second-Order Matching Procedure for the Practical Use in a Program Transformation System.
SEKI-Projekt, MEMO SEKI-83-13
University of Kaiserslautern, 1983

[Hi79] Hikita, T.:
On a class of recursive procedures and equivalent iterative ones.
Acta Informatica, No.12, pp.305-320
Springer, London, 1979

[HuLa78] Huet, G., Lang, B.:
Proving and Applying Program Transformations Expressed with Second-Order Patterns.
in: Acta Informatica 11, pp.31-55, 1978

[KA84] KARL MARK G RAPH
The Markgraf Karl Refutation Procedure.
SEKI-Project, MEMO SEKI-84-01,
University of Kaiserslautern, 1984.

[Ko82] Kott, L.:
Unfold/Fold Program Transformations.
Rapports de Recherche No. 155,
INRIA, Le Chesnay Cedex, France, 1982

[Lo77] Loveman, D.B.:
Program Improvement by Source-to-Source Transformation.
in: JACM 24,1 (1977), pp.121-145

[Ma84] Matheis, H.:
Ein interaktives und syntaxorientiertes Eingabesystem fuer algebraische Spezifikationen, Band I.
SEKI-Projekt, MEMO SEKI-84-03-I
University of Kaiserslautern, 1984

[MNV72] Manna, Z., Ness, S., Vuillemin, J.:
Inductive Methods For Proving Properties Of Programs.
in: Proc. of an ACM Conference on prooving assertions about programs. SIGPLAN Notices Vol.7, No.1, 1972

[MaVu72] Manna, Z., Vuillemin, J.:
Fixpoint Approach to the Theory of Computation.
In: CACM 15,7 (19729, pp.529-536

[OLt84] OLthoff, W.:
ModPascal Report.
SEKI-Projekt, MEMO-SEKI-84-9,
University of KaisersLautern, 1984

[PaHe70] Paterson, M.S., Hewitt, C.E.:
Comparative Schematology
Record of Project MAC
Conference on concurrent systems and parallel computation,
pp.119-128
ACM, New York, 1970

[PaPe76] Partsch, H., Pepper, P.:
A Family of Rules for Recursion Removal.
In: Information Processing Letters,
Vol.5, No.6, 1976, pp.174-177

[Pat70] Paterson, M.S., Hewitt, C.E.:
Comparative Schematology.
in: Conf. Record on Concurrent Systems and Parallel Computations, ACM, 1970

[Pep81] Pepper, P.:
On Program Transformations for Abstract Data Types and Concurrency.
Report. No. STAN-CS-81-883, Stanford University,
October 1981

[Pet83] Petersen, U.: Elimination von Rekursionen.
SEKI-Projekt, MEMO SEKI-83-10
University of KaisersLautern, 1983

[Ri73] Risch, T.:
REMREC - A program for automatic recursion removal in LISP.
Memo. DLU 73/24, University of Uppsala, 1973

[Ro80] Rohl, I.S.:
The elimination of linear recursion: a tutorial Proceedings of the 3rd Australian Computer Conference, 1980

[Sm83] Smith, D.R.:
A Problem Reduction Approach To Program Synthesis.
in: Proc. of the 8th Int. Conf. on Artificial Intelligence,
1983.

[So84] Sommer, W.:
SPESY - Ein interaktives System zur Unterstuetzung integrierter Programmspezifikation und -verifikation. Band I.
SEKI-ProjeKt, MEMO SEKI-84-02-I
University of KaisersLautern, 1984.

[Str83] Stenger, B.:
Ein interaktives System zur Transformation von Programmen.
Studienarbeit, Fb Informatik, Universitaet KaisersLautern,
1983

[St71] Strong, H.R.:
Translating Recursion Equations into Flow Charts.
in: JounaL of Computer and System Science, Vol.5 (1971),
pp.254-285

[VeLo80] Veloso, P.A.S., Lopes, M.A.:
Sorting By Divide-And-Conquer Data Types:
An Example Of Problem-Solving
in: VII Conf. Latinoamericana de Informatica.
Caracas, 1980

[WaSt73] Walker, S.A., Strong, H.R.:
Characterizations of Flowchartable Recursions.
in: Journal Of Computer And System Science 7,
pp.404-447, 1973

[Wi76] Winterstein, G.:
Unification in Second Order Logic.
Internal Report, University of KaisersLautern, March 1976

[Wo70] Woods, W.A.:
Netzwerkgrammatiken für die Analyse natürlicher Sprachen.
in: Eisenberg, P. (Ed.): Maschinelle Sprachanalyse I, 1970,
pp.98-136