**A. Pettorossi, M. Proietti**

# RULES AND STRATEGIES FOR TRANSFORMING FUNCTIONAL AND LOGIC PROGRAMS

**R. 423 Dicembre 1995**

**Alberto Pettorossi** — Dipartimento di Informatica, Sistemi e Produzione, Via della Ricerca Scientifica, 00133 Roma, Italy
**Maurizio Proietti** — Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185 Roma, Italy

**Abstract**

We present an overview of the program transformation methodology, focusing our attention on the so-called 'rules + strategies' approach in the case of functional and logic programs. The paper is intended to offer an introduction to the subject. The various techniques we present are illustrated via simple examples.

A preliminary version of this report has been published in: Möller, B., Partsch, H., and Schuman, S. (eds.): Formal Program Development. Lecture Notes in Computer Science 755, Springer Verlag (1993) 263–304. Also published in: ACM Computing Surveys, Vol 28, No. 2, June 1996.

## 1 Introduction

The program transformation approach to the development of programs has first been advocated by [Burstall-Darlington 77], although the basic ideas were already presented in previous papers by the same authors [Darlington 72, Burstall-Darlington 75].

In that approach the task of writing a correct and efficient program is realized in two phases: the first phase consists in writing an initial, maybe inefficient, program whose correctness can easily be shown, and the second phase, possibly divided into various subphases, consists in transforming the initial program so to get a new program which is more efficient.

This methodology may avoid the difficulty of designing the invariant assertions of loops which are often quite intricate, especially for very efficient algorithms.

The experience gained by the scientific community during the past two decades shows that the methodology of program transformation is very valuable and attractive, in particular for the task of programming 'in the small', that is, when writing single modules of large software systems.

Program transformation has also the advantage of being adaptable to various programming paradigms. In this paper we will focus our attention to the functional and logic cases.

The basic idea of the program transformation approach is depicted in Fig. 1. From the initial program $P_0$, which is the given specification, we want to obtain a final program $P_n$ with the same semantic value, that is, $Sem(P_0) = Sem(P_n)$ for some given semantic function $Sem$. This is often done in various steps, by constructing a sequence $\langle P_0, \ldots, P_n \rangle$ of programs such that $Sem(P_i) = Sem(P_{i+1})$ for $0 \leq i < n$.

Sometimes, if the initial program is nondeterministic (that is, it may produce more than one answer for any given input), we may allow transformation steps which are *sound*, but not *complete*, in the sense that $Sem(P_n)(v) \subseteq Sem(P_0)(v)$, for any input value $v$.

A given cost function $C$ which measures the space or time complexity of the execution of a program, should satisfy the following inequation: $C(P_0) \geq$

Figure 1: The program transformation idea: from program $P_0$ to program $P_n$ preserving the semantic value $V$.

$C(P_n)$. We may allow ourselves to derive a program version, say $P_i$, for some $i > 0$, such that $C(P_0) < C(P_i)$, because subsequent transformations may lead to a program version whose cost is smaller than the one of $P_0$. Unfortunately, there is no general theory of program transformations which deals with this situation in a satisfactory way.

## 2 Transformation Rules and Strategies for Functional Programs

In this section we will use a programming language similar to the one presented in [Burstall-Darlington 77], where programs are written as sets of recursive equations. Many examples will be given below. The extension to the case of functional languages, like ML or Miranda, is straightforward.

We need the following notions. The expression $e_1$ is an *instance* of the expression $e_2$ iff there exists a substitution $\theta$ so that $e_1 = e_2\theta$. For example, $f(x+3)$ is an instance of $f(x+y)$ and the matching substitution $\theta$ is $\{y = 3\}$.

A *context expression* $C[\ldots]$ is an expression $C$ in which one or more occurrences of a given subexpression are 'deleted'. For example, $f(x + (\ldots + 1))$ is a context expression.

The semantics of a program is assumed to be the *least fixed point* of the corresponding set of recursive equations and thus, while transforming programs, we want to preserve the least fixed point semantics.

The equations of any given program are assumed to be: i) *mutually exclusive*, that is, two different left hand sides do not have common instances, and ii) *exhaustive*, that is, for any element $v$ of the domain of any function, say $f$, defined by the recursive equations, there exists at least one left hand side which matches $f(v)$.

## 2.1 Transformation Rules for Functional Programs

The rules for transforming programs expressed as recursive equations, are the following ones:

i) *Definition Rule.* It consists in adding to the current program $P$ a set of mutually exclusive and exhaustive recursive equations, say $f(t_1) = e_1, \ldots, f(t_n) = e_n$, where $f$ is a function symbol not occurring elsewhere in $P$. Thus, the left hand side of any equation introduced by definition is not an instance of the left hand side of any equation already existing in the current program.

ii) *Unfolding Rule.* It consists in replacing an occurrence of an instance of the l.h.s. of an equation by the corresponding instance of its r.h.s., thereby producing a new equation.

For example, if we unfold the subexpression $f(n+1)$ in $f(n+2) = (n+2) \times f(n+1)$ using the equation $f(n+1) = (n+1) \times f(n)$, we get the new equation:

$f(n+2) = (n+2) \times ((n+1) \times f(n))$.

iii) *Folding Rule.* It is the inverse of the unfolding rule. It consists in replacing an occurrence of an instance of the r.h.s. of an equation by the corresponding instance of its l.h.s., thereby producing a new equation.

iv) *Instantiation Rule.* It consists in the introduction of an instance of an already existing equation.

For example, by instantiating $n$ to $m+1$ in the equation: $f(n+1) = (n+1) \times f(n)$, we get, after simplification, $f(m+2) = (m+2) \times f(m+1)$.

v) *Where-Abstraction Rule.* We replace a recursive equation $f(\ldots) = C[e]$ by the new equation: $f(\ldots) = C[z]$ **where** $z = e$, provided that $z$ is a variable which does not occur in $f(\ldots) = C[e]$.

The use of the where-clause has the advantage that in the call-by-value mode of execution, the evaluation of the subexpression $e$ is performed only once.

vi) *Algebraic Replacement Rule.* We derive a new equation by using equalities which hold when we interpret function variables as least fixed points of the corresponding recursive equations, and we interpret basic function symbols, such as $+$, $\times$, etc., as operators in a given algebra.

For instance, the equation $f(n+1) = f(n) \times (n+1)$ can be derived from the equation $f(n+1) = (n+1) \times f(n)$, if we interpret $\times$ as the usual multiplication operation on natural numbers, by using the equality $f(n) \times (n+1) = (n+1) \times f(n)$.

Given the following recursive equations which define the function *sum*:

$sum([\,]) = 0$

$sum(a\!:\!l) = a + sum(l)$,

we have the equality, called *distributivity* of *sum* w.r.t. **if-then-else**:

$sum(\textbf{if } x \textbf{ then } y \textbf{ else } z) = \textbf{if } x \textbf{ then } sum(y) \textbf{ else } sum(z)$,

which can be shown by computational induction [Manna 74].

In this paper we will assume that an equality $expr_1 = expr_2$ holds w.r.t. a given program $P$ iff for all values of the free variables in the expressions $expr_1$ and $expr_2$, *either* the evaluations of $expr_1$ and $expr_2$ using the program $P$ are both nonterminating, *or* they are both terminating and yield the same values.

If the equality $expr_1 = expr_2$ holds w.r.t. $P$ we will also say that $expr_1$ is *equivalent* to $expr_2$ w.r.t. $P$, and when no confusion arises, we will feel free to omit the reference to program $P$.

New programs are obtained from old ones by deriving new equations through the use of the definition, unfolding, folding, instantiation, where-abstraction, and algebraic replacement rules and then selecting a set of mutually exclusive and exhaustive equations.

It can be shown that by using the above six tranformation rules, partial correctness is preserved, that is, if the derived program terminates (by using a fixed point computation rule) then it computes the value computed by the initial program [Kott 78, Courcelle 90, Sands 94].

However, during program transformation we usually need to preserve total correctness, that is, i) the initial program terminates iff the derived program terminates, and ii) in the case of termination, both programs compute the same result. Thus, after applying the transformation rules listed above, we should also check that the derived program terminates for all input values for which the initial program terminates.

### 2.2 Transformation Strategies for Functional Programs

When performing program transformations we may end up with a final program which is equal to the initial one (recall that the folding rule is the inverse of the unfolding rule). Thus, during the transformation process, we need *strategies* which guide the application of the transformation rules and may allow us to derive programs with improved performance.

One of the most relevant steps in applying a transformation strategy is the introduction of new functions, often called in the literature *eureka functions*. In the early days of the transformation methodology, those functions were generated by clever insights (hence their name) and not by strategies.

Unfortunately, there is no general theory of strategies which ensure that the derived programs are more efficient than the initial ones. However, partial results are available for some classes of programs.

We now present some of the transformation strategies which have been proposed in the literature. The basic ideas on which those strategies are based, are present, in various forms, in a large number of papers on program transformation, program synthesis, and automated theorem proving. Our references to the literature are not exhaustive, and they only indicate some relevant papers where the reader may also find material for further study. We will see the strategies in action in various examples below.

i) *Composition Strategy* [Burstall-Darlington 77, Darlington 81, Feather 82, Paige-Koenig 82, Bauer et al. 87, Partsch 90]. If the subexpression $f(g(x))$ occurs in an expression $e$ which is the r.h.s. of a recursive equation in a given program,

- we introduce, by the definition rule, a new function $h$ defined as follows:

  $h(x) = f(g(x))$,

- we find recursive equations for $h(x)$ where the functions $f$ and $g$ do not occur, and
- by folding we replace $f(g(x))$ by $h(x)$ in $e$, and possibly elsewhere in the given program.

Variants of the composition strategy are the *Internal Specialization* technique [Scherlis 81] and the *Deforestation* technique [Wadler 88].

By using the composition strategy (or its variants) one may avoid the construction of intermediate data structures which are produced by $g$ and used as inputs for $f$. This strategy often allows the derivation of programs with improved performance w.r.t. those obtained by lazy evaluation [Wadler 85] if the intermediate data structures are infinite and the 'next item' of the output of $f$ can be produced by knowing only a finite portion of the output of $g$.

ii) *Tupling Strategy* [Burstall-Darlington 77, Pettorossi 77, Darlington 81, Feather 82, Paige-Koenig 82, Bauer et al. 87, Partsch 90]. Given a recursive equation of the form $f(x_1, \ldots, x_n) = C[f_1(e_1), \ldots, f_r(e_r)]$, where the expressions $e_1, \ldots, e_r$ all share the free variable $x$,

- by using the definition rule we introduce the function:

  $h(x, y_1, \ldots, y_m) = \langle f_1(e_1), \ldots, f_r(e_r) \rangle$

  where $x, y_1, \ldots, y_m$ are the free variables occurring in $e_1, \ldots, e_r$,
- we find recursive equations for $h(x, y_1, \ldots, y_m)$ without occurrences of the functions $f_1, \ldots, f_r$, and
- by where-abstraction and folding we replace the given equation $f(x_1, \ldots, x_n) = C[f_1(e_1), \ldots, f_r(e_r)]$ by the new equation:

  $f(x_1, \ldots, x_n) = C[u_1, \ldots, u_r]$ **where** $\langle u_1, \ldots, u_r \rangle = h(x, y_1, \ldots, y_m)$.

  where $u_1, \ldots, u_r$ are fresh variables. By using the algebraic replacement rule we also replace elsewhere in the program the occurrences of $f_i(e_i)$ by $\pi_i(h(x, y_1, \ldots, y_m))$, where $\pi_i$ denotes the $i$-th projection function, for $i = 1, \ldots, r$.

The use of the tupling strategy is very useful if the functions $f_1, \ldots, f_r$ all have access to the same data structure $x$ which is used by no other function. In this case the store used for $x$ can be released as soon as the value of $h$ has been computed. Often, by doing so, one may improve the time$\times$space performance [Pettorossi 84].

The tupling strategy is also very effective when several functions require the computation of the *same subexpression*, in which case we tuple together those functions.

By avoiding either multiple accesses to data structures or common subcomputations one often gets linear recursive programs (that is, equations whose r.h.s. has at most one recursive call) from non-linear recursive programs.

iii) *Generalization Strategy.* It is of various kinds as we now indicate.

1. *Generalization from expressions to variables* [Boyer-Moore 75, Wegbreit 76, Burstall-Darlington 77, Aubin 79, Darlington 81, Feather 82, Bauer et al. 87, Partsch 90]. Given a recursive equation of the form: $f(x_1, \ldots, x_n) = e$, such that a subexpression $s$ occurs in $e$,

   - we introduce the generalized recursive equation: $g(x, y_1, \ldots, y_m) = e[x/s]$

     where $x$ is a fresh variable, $e[x/s]$ denotes the expression $e$ with $x$ substituted for $s$, and $x, y_1, \ldots, y_m$ are the free variables occurring in $e[x/s]$,

   - we find recursive equations for $g(x, y_1, \ldots, y_m)$, and

   - we express the function $f(x_1, \ldots, x_n)$ in terms of $g(x, y_1, \ldots, y_m)$ by folding one or more instances of $e[x/s]$.

2. *Generalization from functions to functions by implicit definition* [Darlington 81, Pettorossi 84]. Given a recursive equation of the form: $f(x_1, \ldots, x_n) = C[expr]$ in a program $P$,

   - we introduce the function $g$ with arity $k$, such that there exist some expressions $e_1, \ldots, e_k$, possibly with the variables $x_1, \ldots, x_n$, such that for all values of the variables $x_1, \ldots, x_n$ we have that $C[g(e_1, \ldots, e_k)]$ is equivalent to $C[expr]$ w.r.t. program $P$,

- we find recursive equations for $g(x_1, \ldots, x_k)$, and
- we replace the equation $f(x_1, \ldots, x_n) = C[expr]$ by the following one:
  $f(x_1, \ldots, x_n) = C[g(e_1, \ldots, e_k)]$.

3. *Lambda Abstraction* [Pettorossi-Skowron 87, Pettorossi 87b]. It consists in replacing a given expression $e_1$ by $e_1[x/e_2]$ **where** $x = e_2$, or equivalently, $(\lambda x.e_1[x/e_2])\ e_2$, where $x$ is a fresh new variable and $e_2$ is a subexpression of $e_1$.

   Lambda Abstraction is a form of generalization which can be applied when i) an expression and one of its subexpressions both visit the same data structure and ii) that subexpression determines a mismatch which makes it impossible to perform a folding step (see Example 6 below).

### 2.3 Examples of Transformations of Functional Programs

We now give some examples of application of the rules and strategies we have introduced.

**Example 1 (The Composition Strategy: List Processing)** The following program computes the sum of the even elements of a list, say $l$, of natural numbers.

1.1 $evensum(l) = sum(even(l))$
1.2 $even([\,]) = [\,]$
1.3 $even(a\!:\!l) = \textbf{if } odd(a) \textbf{ then } even(l) \textbf{ else } a\!:\!even(l)$
1.4 $sum([\,]) = 0$
1.5 $sum(a\!:\!l) = a + sum(l)$.

Now, by using the composition strategy we can avoid the construction of the intermediate list of the even elements of $l$, which is the value of $even(l)$ and it is passed as input to $sum(\_)$. We compose the functions $sum(\_)$ and $even(\_)$ and we define the new function:

$h(l) = sum(even(l))$.

Thus,

1.6 $h([\,]) = sum(even([\,])) = \{\text{unfolding}\} = sum([\,]) = \{\text{unfolding}\} = 0$

1.7 $h(a{:}l) = sum(even(a{:}l)) = \{\text{unfolding}\} =$

$= sum(\textbf{if } odd(a) \textbf{ then } even(l) \textbf{ else } a{:}even(l)) =$

$= \{\text{distributivity of } sum \text{ w.r.t. } \textbf{if-then-else}\} =$

$= \textbf{if } odd(a) \textbf{ then } sum(even(l)) \textbf{ else } sum(a{:}even(l)) =$

$= \{\text{unfolding}\} =$

$= \textbf{if } odd(a) \textbf{ then } sum(even(l)) \textbf{ else } a + sum(even(l)) =$

$= \{\text{folding}\} =$

$= \textbf{if } odd(a) \textbf{ then } h(l) \textbf{ else } a + h(l)$.

We finally express *evensum* in terms of the composite function $h$. In our case from Equation 1.1 we simply have: $evensum(l) = h(l)$. □

The various transformation steps of the derivation of Equation 1.7 for $h(\_)$ have been suggested by the so called *need-for-folding* (see [Darlington 81], where it is called *forced folding*), that is, the requirement of making recursive calls to the composite function $h(\_)$, rather than to the component functions $sum(\_)$ and $even(\_)$. In particular, the use of the distributivity of *sum* w.r.t. **if-then-else** was suggested by the syntactic requirement of producing the occurrences of $sum(even(l))$ which can then be folded using $h(l)$. By performing those folding steps, the efficiency improvements due to the unfolding steps which lead from the expression '$sum(even(a:l))$' to the expression '$\textbf{if } odd(a) \textbf{ then } sum(even(l)) \textbf{ else } a + sum(even(l))$', can be iterated *at each level of recursion*, and thus, they become computationally significant. Indeed, in what follows we will see that by performing folding steps we may transform an exponential time algorithm into a linear time one.

The idea of need-for-folding plays a major role in the program transformation methodology. It can be regarded as a meta-strategy because, as we will see in the examples below, it is the need-for-folding that often suggests the suitable strategies to apply.

**Example 2 (The Tupling Strategy: Towers of Hanoi)** Let us consider the following problem. There are 3 pegs: $A$, $B$, and $C$, and there are $n$ disks, with $n \geq 0$, of different sizes which are stacked as a tower on peg $A$ with smaller disks on top of larger disks. It is required to move the disks from peg $A$ to peg $B$ using peg $C$ as an auxiliary peg. It is possible to move one disk at a time only, and that disk has to be the smallest one of both the tower it comes from and the tower it goes to.

Let $M$ be the set of possible elementary moves, that is, $M = \{AB, BC, CA, BA, CB, AC\}$, where $XY$ denotes the move that takes the top disk from peg $X$ to the top of peg $Y$. Let $M^*$ be the free monoid of sequences of elementary moves. The constant *skip* denotes the empty sequence of moves, and '::' denotes the associative concatenation of sequences. A solution to the Towers of Hanoi Problem is provided by the function $f : N \times \mathit{Pegs}^3 \to M^*$, where $N$ is the set of natural numbers and *Pegs* is the set $\{A,B,C\}$, defined as follows:

2.1 $f(0, a, b, c) = \mathit{skip}$
2.2 $f(n+1, a, b, c) = f(n, a, c, b) :: ab :: f(n, c, b, a) \qquad \text{for } n \geq 0,$

where the variables $a$, $b$, and $c$ assume different values in *Pegs*.

We will derive a solution which does not require the use of a stack and is more efficient than the recursive one. We proceed in two steps as follows: (Step $\alpha$) we transform the general recursion of Equation 2.2 into a linear recursion, and then (Step $\beta$) we transform the linear recursive program into an iterative program. In Step $\alpha$) we will make use of the tupling strategy, and in Step $\beta$) we will use a simple schema equivalence.

*Step* $\alpha$) The program transformation process begins by performing some unfolding steps. We get, for $n \geq 1$:

$$\begin{aligned} f(n+1, a, b, c) = {} & (f(n-1, a, b, c) :: ac :: f(n-1, b, c, a)) :: ab :: \\ & (f(n-1, c, a, b) :: cb :: f(n-1, a, b, c)). \end{aligned}$$

Here and in the sequel we avoid the explicit use of the instantiation rule by assuming that given a context $C[\ldots]$, the expression '$f(n) = C[n-k]$ for $n \geq k$' stands for '$f(n+k) = C[n]$ for $n \geq 0$'.

Now we have that $f(n-1, a, b, c)$ and $f(n-1, b, c, a)$ require the common subcomputation $f(n-2, a, c, b)$, while $f(n-1, b, c, a)$ and $f(n-1, c, a, b)$ require the common subcomputation $f(n-2, b, a, c)$. Thus, in order to avoid repeated subcomputations we apply the tupling strategy and we define a new function $t$ as follows:

2.3 $t(n, a, b, c) = \langle f(n, a, b, c), f(n, b, c, a), f(n, c, a, b)\rangle$.

We then look for the recursive equations of $t(n, a, b, c)$ and we get:

2.4 $t(n+2, a, b, c) =$
$=\langle f(n+2, a, b, c), f(n+2, b, c, a), f(n+2, c, a, b)\rangle = \{\text{unfolding}\} =$
$=\langle (f(n, a, b, c) :: ac :: f(n, b, c, a)) :: ab :: (f(n, c, a, b) :: cb :: f(n, a, b, c)),$
$(f(n, b, c, a) :: ba :: f(n, c, a, b)) :: bc :: (f(n, a, b, c) :: ac :: f(n, b, c, a)),$
$(f(n, c, a, b) :: cb :: f(n, a, b, c)) :: ca :: (f(n, b, c, a) :: ba :: f(n, c, a, b))\rangle =$
$=\{\text{folding and where-abstraction}\} =$
$=\langle (u :: ac :: v) :: ab :: (w :: cb :: u),$
$(v :: ba :: w) :: bc :: (u :: ac :: v),$
$(w :: cb :: u) :: ca :: (v :: ba :: w)\rangle$ **where** $\langle u, v, w\rangle = t(n, a, b, c)$ for $n \geq 0$.

Since $t(n+2, a, b, c)$ is defined in terms of $t(n, a, b, c)$, in order to ensure the termination of the evaluation of $t(n+2, a, b, c)$ for $n \geq 0$, we need to provide the equations for $t(0, a, b, c)$ and $t(1, a, b, c)$. By unfolding using Equations 2.1, 2.2, and 2.3, we get:

2.5 $t(0, a, b, c) = \langle skip, skip, skip\rangle$
2.6 $t(1, a, b, c) = \langle ab, bc, ca\rangle$.

Thus, we have obtained a linear recursive program for the function $t$. We then express the function $f(n, a, b, c)$ in terms of $t(n-2, a, b, c)$ by unfolding, where-abstraction, and folding steps. Recalling that '::' is associative, we get:

2.7 $f(n, a, b, c) =$
$=\langle (f(n-2, a, b, c) :: ac :: f(n-2, b, c, a)) :: ab ::$
$(f(n-2, c, a, b) :: cb :: f(n-2, a, b, c))\rangle =$
$=\langle u :: ac :: v :: ab :: w :: cb :: u\rangle$ **where** $\langle u, v, w\rangle = t(n-2, a, b, c)$ for $n \geq 2$.

In order to preserve termination, since $f(n, a, b, c)$ is defined in terms of $t(n-2, a, b, c)$, we have to provide the values of $f(0, a, b, c)$ and $f(1, a, b, c)$. Th value of $f(0, a, b, c)$ is given by Equation 2.1. The value of $f(1, a, b, c)$ is obtained from Equations 2.1 and 2.2:

2.8 $f(1, a, b, c) = ab$.

Figure 2: A schema equivalence for transforming linear recursion into iteration.

Thus, we have derived a linear recursive program for the function $f$. It is made out of Equations 2.1, 2.4, 2.5, 2.6, 2.7, and 2.8.

*Step β)* Now we can transform the linear recursive program we have derived into an iterative program, and thus, we avoid the use of a stack.

We apply the schema equivalence which transforms Equations 2.1, 2.7, and 2.8. into the program:

---

$\{n = N \; and \; \langle T_1, T_2, T_3\rangle = t(n-2, a, b, c)\}$
**if** $n = 0$ **then** $F := skip$ **else if** $n = 1$ **then** $F := ab$ **else**
$F := T_1 :: ac :: T_2 :: ab :: T_3 :: cb :: T_1$
$\{F = f(N, a, b, c)\}$

---

We also apply the schema equivalence of Fig. 2. This equivalence which can be proved by induction on the natural number $N$, is a simple generalization of the one obtained by dropping the constant argument $r$ from the functions $t$ and $h$. The matching substitution is given by: $z_0 = \langle skip, skip, skip\rangle$, $z_1 = \langle ab, bc, ca\rangle$, $r = \langle a, b, c\rangle$, and

$$\begin{aligned}
&\lambda x.\, h(x, \langle a, b, c\rangle) = \\
&\quad = \lambda x.\, \langle (u :: ac :: v) :: ab :: (w :: cb :: u), (v :: ba :: w) :: bc :: (u :: ac :: v), \\
&\qquad\quad (w :: cb :: u) :: ca :: (v :: ba :: w)\rangle \qquad \textbf{where} \;\; \langle u, v, w\rangle = x.
\end{aligned}$$

We get the iterative program Hanoi listed below, where, for $i = 1, 2, 3$, $T_i$ denotes the $i$-th projection of $T$, and the assignment to $T$ in the body of the while-do loop is a parallel assignment, that is, each new projection of $T$ is independently computed starting from the old projections of $T$. □

$\{n = N \geq 0\}$ Program Hanoi

**if** $n = 0$ **then** $F := \mathit{skip}$ **else if** $n = 1$ **then** $F := ab$ **else**
**begin** $n := n - 2$;
  **if** $\mathit{even}(n)$ **then** $T := \langle \mathit{skip}, \mathit{skip}, \mathit{skip} \rangle$ **else** $T := \langle ab, bc, ca \rangle$;
  **while** $n > 1$ **do begin** $T := \langle T_1 :: ac :: T_2 :: ab :: T_3 :: cb :: T_1,$
    $T_2 :: ba :: T_3 :: bc :: T_1 :: ac :: T_2,$
    $T_3 :: cb :: T_1 :: ca :: T_2 :: ba :: T_3 \rangle$;
    $n := n - 2$
  **end**;
  $F := T_1 :: ac :: T_2 :: ab :: T_3 :: cb :: T_1$
**end** $\{F = f(N, a, b, c)\}$

During the program derivation process we have described in the above Example 2, many other choices for the eureka function $t$ can be made, because there are many other sets of function calls which have common subcomputations. However, not all choices allow us to derive a linear recursive program. We will now present a technique based on the so called *unfolding tree* (or *tree of recursive calls*), which tells us which functions should be tupled together for achieving linear recursion.

The unfolding tree is constructed from a given function call by performing some unfolding steps and recalling only the recursive function calls with their father-son relationship. We then compress that tree by identifying every two nodes having the same recursive call whereby obtaining the *minimal descent directed acyclic graph*, or *m-dag* for short [Bird 80]. We have depicted the m-dag for $f(n, a, b, c)$ in Fig. 3.

Given the minimal descent dag of a recursively defined function, we define an irreflexive and transitive ordering on nodes as follows: for any two nodes $m$ and $n$, we have that $m > n$ holds iff the function call associated with node $m$ requires the computation of the function call associated with node $n$.

A *cut* in an m-dag is a set of nodes such that if we remove them, with their incoming and outgoing edges, we are left with two disconnected subgraphs $g_1$

Figure 3: Unfolding tree of $f(n,a,b,c)$ by using $f(n,a,b,c) = f(n-1,a,c,b) :: ab :: f(n-1,c,b,a)$. The label '$l$' denotes the left call $f(n-1,a,c,b)$ and the label '$r$' denotes the right call $f(n-1,c,b,a)$.

and $g_2$ such that for any node $m$ in $g_1$ and any node $n$ in $g_2$ we have that $m > n$.

A sequence of cuts $\langle c_i \mid 0 \leq i \rangle$ in an m-dag is said to be *progressive* [Pettorossi 84] iff

i) $\forall i > 0.\ c_{i-1}$ and $c_i$ have the same finite cardinality, and

ii) $\forall i > 0.\ c_{i-1} \neq c_i$, and

iii) $\forall i > 0.\ \forall n \in c_i.\ \exists m \in c_{i-1}.$ if $n \neq m$ then $m > n$, and

iv) $\forall i > 0.\ \forall m \in c_{i-1}.\ \exists n \in c_i.$ if $n \neq m$ then $m > n$.

From i) and ii) it follows that for all $i > 0$ neither $c_{i-1}$ is contained in $c_i$ nor $c_i$ is contained in $c_{i-1}$. In intuitive terms, while moving along a progressive sequence of cuts from $c_{i-1}$ to $c_i$ we delete 'large' nodes and we add 'small' nodes. Thus, given the m-dag of Fig. 3 where $m > n$ is depicted by positioning the node $m$ above the node $n$, we have, among others, the following cuts:

$c_0 = \{f(n-2,a,b,c), f(n-2,b,c,a), f(n-2,c,a,b)\}$,
$c_1 = \{f(n-4,a,b,c), f(n-4,b,c,a), f(n-4,c,a,b)\}$,
$\ldots$,
$c_i = \{f(n-2i-2,a,b,c), f(n-2i-2,b,c,a), f(n-2i-2,c,a,b)\}$,
$\ldots$,

and $S = \langle c_0, c_1, \ldots, c_i, \ldots \rangle$ is a progressive sequence of cuts.

It can be shown that if a progressive sequence of cuts $\langle c_i \mid 0 \leq i \rangle$ exists for the m-dag of the function call $f$ such that:

i) the initial function call of $f$ can be computed from the values of the function calls of the cut $c_0$, and

ii) there exists a function, say $h$, such that $h$ does not depend on $i$, and for each $i \geq 0$ the function calls of $c_{i-1}$ can be computed from the function calls of $c_i$ using $h$,

then the tupling strategy which tuples together the function calls in a cut, generates a linear recursive program for the computation of $f$.

This result allowed us to perform Step $\alpha$) of the derivation of Example 2. Indeed, Equation 2.3 has been determined by tupling the function calls of a cut in the above mentioned sequence $S$.

With respect to Step $\beta$) of that derivation, that is, the transformation from a linear recursive program to an iterative program, we recall that that there is a general result [Paterson-Hewitt 70] which ensures that every linear recursion can be translated into an iteration using a constant number of memory cells (that is, avoiding the use of a stack). However, that result cannot be used in the context of the program transformation methodology, because it degrades the time performance from $O(n)$ to $O(n^2)$. Thus, we use, instead, the schema equivalence of Fig 2.

Various techniques for the automation of the tupling strategy are described in [Chin 90, Chin 93] .

**Example 3 (The Generalization Strategy from Expressions to Variables: Linear Recursion Without Stack)** Let us consider the following linear recursive program:

3.1 $f(x) =$ **if** $p(x)$ **then** $a(x)$ **else** $b(c(x), f(d(x)))$.

Let us assume that the function $b(\_,\_)$ is strict w.r.t. its second argument and $b(\_,\_)$ is associative. Thus, $b(\_,\_)$ distributes w.r.t. **if-then-else**. In order to improve efficiency when evaluating $f(x)$, we want to derive an iterative program. This can be done in two steps: $\alpha$) we first look for a tail recursive program, and $\beta$) we then apply a schema equivalence to transform this tail recursive program into an iterative one.

*Step* $\alpha$) We have:

$$
\begin{aligned}
f(x) = & \ \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } b(c(x), f(d(x))) = \{\text{unfolding}\} = \\
& = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else} \\
& \quad b(c(x), \textbf{if } p(d(x)) \textbf{ then } a(d(x)) \textbf{ else } b(c(d(x)), f(d(d(x))))) = \\
& = \{\text{distributivity of } b(\_,\_) \text{ w.r.t. } \textbf{if-then-else}\} = \\
& = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else} \\
& \quad \textbf{if } p(d(x)) \textbf{ then } b(c(x), a(d(x))) \textbf{ else } b(c(x), b(c(d(x)), f(d(d(x))))) = \\
& = \{\text{associativity of } b(\_,\_)\} = \\
& = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else} \\
& \quad \textbf{if } p(d(x)) \textbf{ then } b(c(x), a(d(x))) \textbf{ else } b(b(c(x), c(d(x))), f(d(d(x)))).
\end{aligned}
$$

A tail recursive program can be achieved if we can fold the expression $b(b(c(x), c(d(x))), f(d(d(x))))$, call it $e_1$, with the previously derived expression $b(c(d(x)), f(d(d(x))))$, call it $e_2$, which is the value of $f(d(x))$ for $p(d(x)) =$ *false*. The mismatch between the first arguments of the outermost $b$'s in $e_1$ and $e_2$ does not allow us to perform a folding step.

Thus, the need-for-folding suggests us to generalize the first argument of the function $b$ to a variable, say $y$, and to define, by applying the generalization strategy, the following function $F$:

3.2 $F(y, x) = b(y, f(x))$,

whose r.h.s. is an expression which generalizes both $e_1$ and $e_2$. We then get the following recursive equation for $F$:

$$
\begin{aligned}
F(y, x) & = b(y, f(x)) = \{\text{using Equation 3.1}\} = \\
& = b(y, \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } b(c(x), f(d(x)))) = \\
& = \{\text{distributivity of } b(\_,\_) \text{ w.r.t. } \textbf{if-then-else}\} = \\
& = \textbf{if } p(x) \textbf{ then } b(y, a(x)) \textbf{ else } b(y, b(c(x), f(d(x)))) = \\
& = \{\text{associativity of } b(\_,\_)\} = \\
& = \textbf{if } p(x) \textbf{ then } b(y, a(x)) \textbf{ else } b(b(y, c(x)), f(d(x))) = \{\text{folding}\} = \\
& = \textbf{if } p(x) \textbf{ then } b(y, a(x)) \textbf{ else } F(b(y, c(x)), d(x)).
\end{aligned}
$$

Thus, by folding Equation 3.1 using Equation 3.2 we get:

Figure 4: A schema equivalence for tail recursion.

3.3 $f(x) = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } F(c(x), d(x))$.

Now we have derived the following tail recursive program for computing $f(x)$:

3.3 $f(x) = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } F(c(x), d(x))$

3.4 $F(y, x) = \textbf{if } p(x) \textbf{ then } b(y, a(x)) \textbf{ else } F(b(y, c(x)), d(x))$.

*Step $\beta$)* This tail recursive program can be transformed into an iterative program by applying the schema equivalence shown in Fig. 4, which can easily be proved by computational induction.

We finally get the following program:

```
                {x = X}
if p(x) then res := a(x) else
begin y := c(x);  x := d(x);
    while not p(x) do begin y := b(y, c(x));  x := d(x) end;
    res := b(y, a(x))
end             {res = f(X)}
```

Now we continue our program derivation by assuming that there exists a neutral element, say $\eta$, for $b(\_,\_)$, that is, $\forall x.\ b(\eta, x) = x = b(x, \eta)$. We have that:

3.3* $f(x) = \{\text{neutral element}\} = b(\eta, f(x)) = \{\text{folding using 3.2}\} = F(\eta, x)$.

Thus, by applying again the schema equivalence of Fig. 4 for $Y = \eta$, we get the following iterative program for computing the value of $f(X)$ which is equal to $F(\eta, X)$:

$\{true\}$
$y := \eta;\ x := X;$
**while** *not* $p(x)$ **do begin** $y := b(y, c(x));\ x := d(x)$ **end**;
$res := b(y, a(x))$
$\{res = F(\eta, X) = f(X)\}$

In particular, the program:

3.5 $fact(x) =$ **if** $x = 0$ **then** $1$ **else** $x \times fact(x-1)$ for $x \geq 0$

is an instance of Equation 3.1 for $p(x) = (x = 0), a(x) = 1, b(y, x) = y \times x$, $c(x) = x$, and $d(x) = x - 1$. We have that the neutral element $\eta$ for $b(\_,\_)$ is 1. Thus, Equations 3.3* and 3.4 produce the following tail recursive program for $fact(x)$:

3.6 $fact(x) = G(1, x)$ for $x \geq 0$
3.7 $G(y, x) =$ **if** $x = 0$ **then** $y$ **else** $G(y \times x, x - 1)$.

The corresponding iterative program obtained by using again the schema equivalence of Fig. 4, is:

$\{N \geq 0\}$ Program Factorial.1
$y := 1;\ x := N;$ **while** $x \neq 0$ **do begin** $y := y \times x;\ x := x - 1$ **end**;
$res := y \times 1$
$\{res = G(1, N) = fact(N)\}$

where the precondition $\{true\}$ of Fig. 4 has been strengthened by $\{N \geq 0\}$. In this last program we can replace $x \neq 0$ by $x > 0$ (because $N \geq 0$) and $y \times 1$ by $y$.

We can also get rid of the variable *res*, and use, instead, the variable $y$. Thus, the assignment $res := y \times 1$ can be avoided. By performing these simplifications we get:

$\{N \geq 0\}$ Program Factorial.2
$y := 1;\ x := N;$ **while** $x > 0$ **do begin** $y := y \times x;\ x := x - 1$ **end**;

$\{y = fact(N)\}$

---

□

The program for the function $f$ consisting of Equations 3.3* and 3.4 can also be derived from Equation 3.1 by applying the so called *accumulation strategy* [Bird 84], whereby given an associative operation ($b(\_,\_)$, in our case), we can store the partial results of its application at each recursive call, in a new argument ($y$, in our case). The derivation we have presented in the above Example 3, indicates that such a new argument which plays the role of an accumulator, does not come from an ad hoc invention, but it comes from a generalization step.

Now we give some more examples where it will be shown that:

i) the generalization strategy from (sub)expressions to variables may allow us to get a tail recursive program, even though the initial program is not linear recursive (Example 4),

ii) the generalization from functions to functions by implicit definition may allow us to exponentially reduce the time complexity (Example 5), and

iii) the Lambda Abstraction strategy allows us to define and manipulate pointers in a disciplined way and visit data structures only once (Example 6). (For a transformational approach to the derivation of programs which use pointers, the reader may also refer to [Möller 93]).

**Example 4 (The Generalization Strategy from Expressions to Variables: Tree Processing)** Let us consider the following program which computes the height of a binary tree [Chatelin 76].

4.1 $h(tip(m)) = 0$
4.2 $h(tree(S,T)) = 1 + max(h(S), h(T))$.

We perform some unfolding steps starting from Equation 4.2 and we have:

4.3 $$\begin{aligned} h(tree(S,T)) &= 1 + max(h(S), h(T)) = \\ &= max(1 + h(S), 1 + h(T)) = \qquad (\dagger) \\ &= \{\text{assuming } S = tree(S1, S2)\} = \end{aligned}$$

$$
\begin{aligned}
&= max(2 + max(h(S1), h(S2)), 1 + h(T)) = \\
&= \{\text{distributivity of } + \text{ over } max\} = \\
&= max(max(2 + h(S1), 2 + h(S2)), 1 + h(T)) = \\
&= \{\text{associativity of } max\} = \\
&= max(2 + h(S1), max(2 + h(S2), 1 + h(T))).
\end{aligned}
$$

In order to get a tail recursive program by folding, we match this last expression against a previous r.h.s. of Equation 4.3 and, in particular, we choose $max(1+h(S), 1+h(T))$ (see †). We have that the expression $max(2+h(S1), max(2+h(S2), 1+h(T)))$ is an instance of $max(1+h(S), 1+h(T))$ if we generalize the constant 1 (occurring in $1+h(S)$) and the second argument $1+h(T)$ to the variables $n$ and $u$, respectively. Thus, by applying the generalization strategy we introduce the following function $g$:

4.4 $g(n, S, u) = max(n + h(S), u)$.

We have that:

4.5 $h(T) = \{\text{properties of } max\} = max(0 + h(T), 0) = \{\text{folding using } 4.4\} = g(0, T, 0)$.

The recursive equations for the function $g$ are as follows:

4.6 $g(n, tip(m), u) = \{\text{by 4.4 and 4.1}\} = max(n+0, u) = max(n, u)$

4.7 
$$
\begin{aligned}
g(n, tree(S,T), u) &= max(n + h(tree(S,T)), u) = \{\text{unfolding}\} = \\
&= max(n + 1 + max(h(S), h(T)), u) = \\
&= \{\text{distributivity of } + \text{ over } max\} = \\
&= max(max(n+1+h(S), n+1+h(T)), u) = \\
&= \{\text{associativity of } max\} = \\
&= max(n+1+h(S), max(n+1+h(T), u)) = \\
&= \{\text{folding using } 4.4\} = \\
&= g(n+1, S, max(n+1+h(T), u)).
\end{aligned}
$$

By a final folding step using 4.4 we get:

4.8 $g(n, tree(S,T), u) = g(n+1, S, g(n+1, T, u))$.

Thus, one of the two genuine recursive calls of the function $h$ in the r.h.s. of Equation 4.2 has been transformed into a tail recursive one.

The final program is given by the Equations 4.5, 4.6, and 4.8. □

**Example 5 (The Generalization Strategy from Functions to Functions: Linear Recurrence Relations in Logarithmic Time)** We first consider the simple case of the Fibonacci function.

| | |
|---|---|
| 5.1 $f(0) = 1$ | Program Fibonacci.0 |
| 5.2 $f(1) = 1$ | |
| 5.3 $f(n+2) = f(n+1) + f(n)$ | for $n \geq 0$. |

We first generalize the two occurrences of the constant 1 in the r.h.s.'s of Equations 5.1 and 5.2 to the distinct variables $a_0$ and $a_1$. We get the following equations, where $n \geq 0$:

5.4 $G(a_0, a_1, 0) = a_0$
5.5 $G(a_0, a_1, 1) = a_1$
5.6 $G(a_0, a_1, n+2) = G(a_0, a_1, n+1) + G(a_0, a_1, n)$
5.7 $f(n) = G(1, 1, n)$.

We then generalize the constant 2 occurring in Equation 5.6 to a variable, say $k$, and we define the following function for $n \geq 0$ and $k \geq 0$:

5.8 $F(a_0, a_1, n, k) = G(a_0, a_1, n+k)$.

We have:
5.9 $F(a_0, a_1, n, 0) = G(a_0, a_1, n)$
5.10 $F(a_0, a_1, n, 1) = G(a_0, a_1, n+1)$
5.11 $F(a_0, a_1, n, k+2) = G(a_0, a_1, n+k+2) = \{\text{unfolding}\} =$
$$= G(a_0, a_1, n+k+1) + G(a_0, a_1, n+k) = \{\text{folding}\} =$$
$$= F(a_0, a_1, n, k+1) + F(a_0, a_1, n, k).$$

We may then perform some unfolding steps starting from $F(a_0, a_1, n, k+2)$. We get:

$$\begin{aligned} F(a_0, a_1, n, k+2) &= F(a_0, a_1, n, k+1) + F(a_0, a_1, n, k) = \\ &= 2\,F(a_0, a_1, n, k) + F(a_0, a_1, n, k-1) = \\ &= 3\,F(a_0, a_1, n, k-1) \; + \; 2\,F(a_0, a_1, n, k-2) \end{aligned}$$

and eventually we get:

$$\begin{aligned} F(a_0, a_1, n, k+2) &= c_1 \; F(a_0, a_1, n, 1) + c_2 \; F(a_0, a_1, n, 0) = \{\text{folding}\} = \\ &= c_1 \; G(a_0, a_1, n+1) + c_2 \; G(a_0, a_1, n). \end{aligned}$$

We can now perform a generalization step from functions to functions *by implicit definition* and we assume that the constants $c_1$ and $c_2$ are the values of two functions, say $s(k)$ and $r(k)$. Thus, we assume that:

5.12 $F(a_0, a_1, n, k) = r(k)\ G(a_0, a_1, n) + s(k)\ G(a_0, a_1, n+1)$

and we look for the explicit definition of the functions $r(k)$ and $s(k)$.

From Equations 5.9 and 5.12 we get:
$G(a_0, a_1, n) = r(0)\ G(a_0, a_1, n) + s(0)\ G(a_0, a_1, n+1)$, and since this equality should hold for all values of $a_0$ and $a_1$, we get: $r(0) = 1$ and $s(0) = 0$.

Analogously, from Equations 5.10 and 5.12 we get: $r(1) = 0$ and $s(1) = 1$. From the r.h.s.'s of Equations 5.11 and 5.12 (for $k+2$ instead of $k$) we get:

$$F(a_0, a_1, n, k+1) + F(a_0, a_1, n, k) =$$
$$= r(k+2)G(a_0, a_1, n) + s(k+2)G(a_0, a_1, n+1).$$

By Equation 5.12 for $k$ and $k+1$ we have:

$$r(k+1)G(a_0, a_1, n) + s(k+1)G(a_0, a_1, n+1)+$$
$$r(k)G(a_0, a_1, n) + s(k)G(a_0, a_1, n+1) =$$
$$= r(k+2)G(a_0, a_1, n) + s(k+2)G(a_0, a_1, n+1)$$

which gives us the two equations: $r(k+2) = r(k+1) + r(k)$ and $s(k+2) = s(k+1) + s(k)$.

Therefore, having $r(0) = 1$, $r(1) = 0$, and $r(k+2) = r(k+1) + r(k)$ we derive by Equations 5.4, 5.5, and 5.6 that:

$$r(k) = G(1, 0, k) \quad \text{and} \quad s(k) = G(0, 1, k).$$

Thus, from Equation 5.12 and Definition 5.8 we get for $n \geq 0$ and $k \geq 0$:

5.13 $G(a_0, a_1, n+k) = G(1, 0, k)\ G(a_0, a_1, n) + G(0, 1, k)\ G(a_0, a_1, n+1)$.

From Equation 5.13 for $n = k$ and $n = k+1$ we get Equations 5.14 and 5.15 of the following program for computing the Fibonacci function, where $n \geq 0$ and $k > 0$:

5.7 $f(n) = G(1,1,n)$ Program Fibonacci.1
5.4 $G(a_0, a_1, 0) = a_0$
5.5 $G(a_0, a_1, 1) = a_1$
5.14 $G(a_0, a_1, 2k) = G(1,0,k)\, G(a_0, a_1, k) + G(0,1,k)\, G(a_0, a_1, k+1)$
5.15 $G(a_0, a_1, 2k+1) = G(1,0,k)\, G(a_0, a_1, k+1) + G(0,1,k)\, G(a_0, a_1, k+2)$.

Despite of the fact that at each recursive call the third argument of $G$ is divided by 2 (apart from some additive constants), this program does *not* have a logarithmic running time, because the recursion for $G$ is *not* linear. However, the transformation process may continue by applying the tupling strategy, because the recursive calls of $G(1,0,k)$ and $G(0,1,k)$ have the common subcomputation $G(1,0,k/2)$, and we introduce, by the definition rule, the following function $p(k)$ for $k \geq 0$:

$$p(k) = \langle G(1,0,k), G(0,1,k) \rangle.$$

Looking for the recursive equations of $p(k)$ we get:

$$\begin{aligned}
&p(0) = \langle 1, 0 \rangle \\
&p(1) = \langle 0, 1 \rangle \\
&p(2k) = \langle G(1,0,2k), G(0,1,2k) \rangle = \{\text{unfolding using Equation 5.13}\} = \\
&\qquad = \langle G(1,0,k)G(1,0,k) + G(0,1,k)G(1,0,k+1), \\
&\qquad\quad G(1,0,k)G(0,1,k) + G(0,1,k)G(0,1,k+1) \rangle = \\
&\qquad = \{G(1,0,k+1) = G(0,1,k) \\
&\qquad\quad \text{and } G(0,1,k+1) = G(0,1,k) + G(1,0,k)\} = \\
&\qquad = \langle a^2 + b^2, b^2 + 2ab \rangle \qquad \textbf{where} \;\; \langle a, b \rangle = p(k) \\
&p(2k+1) = \langle G(1,0,2k+1), G(0,1,2k+1) \rangle = \\
&= \{\text{unfolding using Equation 5.13}\} = \\
&\qquad\quad = \langle 2ab + b^2, (a+b)^2 + b^2 \rangle \qquad \textbf{where} \;\; \langle a, b \rangle = p(k).
\end{aligned}$$

The computation time of the function $p$ is logarithmic and we achieved our goal. Now we need to express the function $f$ in terms of the function $p$, instead of $G$, and we have:

$$f(0) = 1$$

$$
\begin{aligned}
&f(1) = 1\\
&f(2k) = G(1,1,2k) = \{\text{by Equation 5.13}\} =\\
&\qquad = G(1,0,k)\, G(1,1,k) + G(0,1,k)\, G(1,1,k+1) =\\
&\qquad = \{\text{by Equation 5.13}\} =\\
&\qquad = G(1,0,k)\, (G(1,0,k) + G(0,1,k)) +\\
&\qquad\quad G(0,1,k)\, (G(1,0,k) + 2G(0,1,k)) =\\
&\qquad = (a+b)^2 + b^2 \qquad\qquad \textbf{where } \langle a,b\rangle = p(k)\\
&f(2k+1) = G(1,1,2k+1) = \{\text{analogously to the case for } f(2k)\} =\\
&\qquad (a+b)^2 + 2b(a+b) \qquad\qquad \textbf{where } \langle a,b\rangle = p(k).
\end{aligned}
$$

We finally get the following program, where $k > 0$:

---

Program Fibonacci.2

$$
\begin{aligned}
&f(0) = 1\\
&f(1) = 1\\
&f(2k) = (a+b)^2 + b^2 && \textbf{where } \langle a,b\rangle = p(k)\\
&f(2k+1) = (a+b)^2 + 2b(a+b) && \textbf{where } \langle a,b\rangle = p(k)\\
&p(0) = \langle 1,0\rangle\\
&p(1) = \langle 0,1\rangle\\
&p(2k) = \langle a^2+b^2, 2ab+b^2\rangle && \textbf{where } \langle a,b\rangle = p(k)\\
&p(2k+1) = \langle 2ab+b^2, (a+b)^2+b^2\rangle && \textbf{where } \langle a,b\rangle = p(k).
\end{aligned}
$$

---

The derivation of this program for the logarithmic evaluation of the Fibonacci function shows the strength of the combined use of the generalization strategy together with the tupling strategy. In what follows we will see more examples of this synergism between the two strategies.

A final derivation step consists in finding an iterative program for computing the Fibonacci function. We can apply the schema equivalence given in Fig. 5, whose correctness can easily be proved by induction on the natural number $K$.

In our case, since the $b$ and $d$ operations are integer divisions by 2, the elements in the stack before (and after) performing the assignment $P := e$ give us the binary expansion '$k(0), \ldots, k(s)$' of the number $K$, if we take $A = 0$ and $C = 1$. The top of the stack is the most significant bit $k(s)$. We may assume that the empty stack represents the binary expansion of 0.

Figure 5: A schema equivalence from recursion to iteration.

From Program Fibonacci.2 we easily get the following iterative program, where the assignments to $u$ and $v$ are parallel assignments:

---

$\{N \geq 0\}$ Program LogFibonacci

**if** $N \leq 1$ **then** $F := 1$ **else**
**begin** $K := N \; div \; 2;\; s := \lfloor \log_2(K) \rfloor;$
$\{K = \sum_{i=0}^{s} k(i)\; 2^i\}$
$u := 1;\; v := 0;\; i := s;$
**while** $i \geq 0$ **do begin if** $k(i) = 0$ **then** $u, v := u^2 + v^2, 2uv + v^2;$
**if** $k(i) = 1$
**then** $u, v := 2uv + v^2, (u+v)^2 + v^2;$
$i := i - 1$
**end**;
**if** $even(N)$ **then** $F := (u+v)^2 + v^2$ **else** $F := (u+v)^2 + 2v(u+v)$
**end** $\{F = f(N)\}$

---

□

We can extend the derivation shown in Example 5 to any homogeneous linear recurrence relation of order $r$ with constant coefficients in any semiring structure whose operations are denoted by $+$ and $\times$. Hence, we can derive a logarithmic time algorithm for computing any function $h$ defined as follows:

$$h(0) = h_0$$
$$\ldots$$
$$h(r-1) = h_{r-1}$$
$$h(n) = b_0 \times h(n-r) + \cdots + b_{r-1} \times h(n-1) \qquad \text{for } n \geq r.$$

The interested reader may refer to [Pettorossi-Burstall 82].

**Example 6 (The Lambda Abstraction Strategy: Palindrome in One Visit Avoiding the Append Function)** Let us consider the following program for testing whether or not a given list $l$ is palindrome.

6.1 $palindrome(l) = eqlist(l, rev(l))$

6.2 $eqlist([\,],[\,]) = true$

6.3 $eqlist(a\!:\!l_1, b\!:\!l_2) \;=\; (a=b) \textit{ and } eqlist(l_1, l_2)$

6.4 $rev([\,]) = [\,]$

6.5 $rev(a:l) = rev(l) :: [a]$

6.6 $x :: y = \textbf{if } x = [\,] \textbf{ then } y \textbf{ else } hd(x)\!:\!(tl(x) :: y)$

where : and :: are the infix operators for *cons* and *append*, respectively. This program visits the given list twice, a first time for its reversal (when computing *rev*) and a second time for testing equality (when computing *eqlist*). We look for a program which visits the given list only once. We also want to avoid the use of the *append* function which is expensive, because using Equation 6.6 the number of *cons* operations needed for reversing a list of length $n$ is $O(n^2)$.

Equations 6.4 and 6.5 can be transformed into a tail recursive form as indicated in Example 3. Indeed, they are an instance of Equation 3.1 for $p(x) = null(x)$, $a(x) = \eta = [\,]$, $b(x,y) = y :: x$, $c(x) = [hd(x)]$, and $d(x) = tl(x)$. By writing $h(x,y)$ instead of $F(y,x)$, from Equations 3.3* and 3.4 we get:

6.7 $rev(l) = h(l,[\,])$

6.8 $h([\,],x) = x$

6.9 $h(a\!:\!l, x) = h(l, [a] :: x) = h(l, a\!:\!x)$.

By Equation 3.2 we also have: $h(l,x) \;=\; F(x,l) \;=\; b(x, f(l)) \;=\; rev(l) :: x$. Thus, we obtain a program without the *append* function by replacing the Equations 6.4, 6.5, and 6.6 by the Equations 6.7, 6.8, and 6.9. Then, in order to derive a program which goes through the input list only once, we continue our program transformation by applying the composition strategy. We get:

6.10 $palindrome([\,]) = eqlist([\,], rev([\,])) = \{\text{unfolding}\} = true$

6.11 $palindrome(a\!:\!l) = eqlist(a\!:\!l, rev(a\!:\!l)) = \{\text{unfolding}\} =$
$= (a = hd(rev(a\!:\!l)))\ and\ eqlist(l, tl(rev(a\!:\!l))).$

When trying to fold the r.h.s. of Equation 6.11 using Equation 6.1 we have a mismatch between the expression $eqlist(l, rev(l))$ in Equation 6.1 and $eqlist(l, tl(rev(a\!:\!l)))$ in Equation 6.11. We can apply the Lambda Abstraction strategy which consists in abstracting away the mismatching argument. In our case we rewrite Equation 6.11 as follows:

6.12 $palindrome(a\!:\!l) = eqlist(a\!:\!l, rev(a\!:\!l)) = \{\text{by Lambda Abstraction}\} =$
$= (\lambda x.\ eqlist(a\!:\!l, x))\ rev(a\!:\!l) = \{\text{Equation 6.7}\} =$
$= (\lambda x.\ eqlist(a\!:\!l, x))\ h(a\!:\!l, [\,]).$

Now, both $\lambda x.\ eqlist(a : l, x)$ and $h(a : l, [\,])$ visit the same data structure $a\!:\!l$. We can use the tupling strategy and we define:

6.13 $Q(l) \ = \ \langle \lambda x.\ eqlist(l, x), h(l, [\,])\rangle$, whose recursive equations are as follows:

$Q([\,]) = \langle \lambda x.\ eqlist([\,], x), h([\,], [\,])\rangle = \{\text{unfolding}\} = \langle \lambda x.\ null(x), [\,]\rangle$
$Q(a\!:\!l) = \langle \lambda x.\ eqlist(a\!:\!l, x), h(a\!:\!l, [\,])\rangle = \{\text{unfolding}\} =$
$= \langle \lambda x.\ (a = hd(x))\ and\ eqlist(l, tl(x)), h(l, [a])\rangle.$

We cannot fold this last equation using Equation 6.13 because of the mismatch between $h(l, [\,])$ and $h(l, [a])$. Thus, we generalize in the definition of the function $Q(l)$ the list $[\,]$ to a variable, say $y$, and we define the tupled function:

6.14 $R(l, y) \ = \ \langle \lambda x.\ eqlist(l, x), h(l, y)\rangle$, whose recursive equations are as follows:

6.15 $R([\,], y) = \langle \lambda x.\ eqlist([\,], x), h([\,], y)\rangle = \langle \lambda x.\ null(x), y\rangle$

6.16 $R(a\!:\!l, y) = \langle \lambda x.\ a = hd(x)\ and\ eqlist(l, tl(x)), h(l, a\!:\!y)\rangle = \{\text{folding}\} =$
$= \langle \lambda x.\ a = hd(x)\ and\ u(tl(x)), v\rangle$ **where** $\langle u, v\rangle = R(l, a\!:\!y).$

We finally express Equation 6.12 in terms of the components of the function $R$, thereby getting Equation 6.12* below. We have:

| | |
|---|---|
| 6.11 $palindrome([\,]) = true$ | Program Palindrome.1 |
| 6.12* $palindrome(a\!:\!l) = u(v)$ | **where** $\langle u, v\rangle = R(a\!:\!l, [\,])$ |
| 6.15 $R([\,], y) = \langle \lambda x.\, null(x), y\rangle$ | |
| 6.16 $R(a\!:\!l, y) = \langle \lambda x.\, a = hd(x)\ and\ u(tl(x)),\ v\rangle$ | **where** $\langle u, v\rangle = R(l, a\!:\!y)$. |

This final algorithm visits the input list only once because the recursive definition of $R(a\!:\!l, y)$ is in terms of $R(l, y)$ only.

Notice that if instead of Lambda Abstraction, we apply generalization from expressions to variables, we should introduce the function $S(x, l, y) = \langle eqlist(l, x), h(l, y)\rangle$, instead of $R(l, y)$. However, by doing so we would not be able to express *palindrome* in terms of $S(x, l, y)$.

We can further improve the linear recursive program Palindrome.1 we have derived, by transforming it into an iterative program as follows. From Equation 6.1 we have:

$$\begin{aligned} 6.17\quad palindrome(a\!:\!l) &= eqlist(a\!:\!l, rev(a\!:\!l)) = \{\text{commutativity of } eqlist\} = \\ &= eqlist(rev(a\!:\!l), a\!:\!l) = \\ &= \{\text{folding using Equation 6.14}\} = \\ &= \pi_1(R(rev(a\!:\!l), y))\,(a\!:\!l) \text{ for some } y, \end{aligned}$$

where, as usual, $\pi_1(\langle x, y\rangle) = x$.

From Equation 6.16 we also have that $\pi_1(R(a : l, \_))$ does not depend on the value of $\pi_2(R(l, \_))$. Thus, when computing $palindrome(a : l)$ according to the above Program Palindrome.1, we do not need the second projection of $R(rev(a\!:\!l), \_)$.

Moreover, in the recursive definition of $R$ we have that the second argument of $R$ is only used for modifying itself. This means that there exists a function $T$ such that $\forall l, y.\, T(l) = \pi_1(R(l, y))$. Indeed, we have: $T(l) = \lambda x.\, eqlist(l, x)$.

Thus, from Equation 6.17 we get:

6.17* $palindrome(a\!:\!l) = T(rev(a\!:\!l))(a\!:\!l)$.

By using Equations 6.15 and 6.16 we derive the following program:

| | |
|---|---|
| 6.11 $palindrome([\,]) = true$ | Program Palindrome.2 |
| 6.17* $palindrome(a\!:\!l) = T(rev(a\!:\!l))(a\!:\!l)$ | |

Figure 6: A schema equivalence for the *palindrome* function.

6.18 $T([\,]) = \lambda x.\, null(x)$
6.19 $T(a\!:\!l) = \lambda x.\, a = hd(x)\ \ and\ \ (T(l)tl(x))$.

---

We can then apply the schema equivalence of Fig. 6, which can be proved by induction on $l$, and we derive the iterative program ItPalindrome listed below.

---

$\{l = L\}$ Program ItPalindrome

**if** $l = [\,]$ **then** $P := true$ **else**
**begin** $initl := l$;   $u := \lambda x.\, null(x)$;
  **while** $l \neq [\,]$ **do**
  **begin** $u := \lambda x.\, (hd(l) = hd(x))\ \ and\ \ u(tl(x))$;   $l := tl(l)$ **end**;
  $P := u(initl)$
**end**  $\{P = palindrome(L)\}$

---

The final assignment $P := u(initl)$ in the program ItPalindrome could be considered as a second visit of the given list $l$, and it may seem a bit strange to claim that we have obtained an algorithm which visits the input list only once. However, a detailed analysis [Pettorossi-Proietti 87] shows that the number of *hd* and *tl* operations used by the ItPalindrome program is reduced by 1/3 with respect to the naive algorithm (that is, Equations 6.1–6.3 and 6.7–6.9) which makes two visits of the list $l$.

The ItPalindrome program also indicates that via program transformation we can discover quite intricate algorithms. Indeed, when developing an algorithm which checks whether or not a given list is palindrome in one visit, we face the main difficulty of discovering the middle element of the list which is known only when the visit is completed. That difficulty can be overcome by using a stack of pointers in a sophisticated way. Our program transformation above shows that this clever use of pointers can be automatically derived by Lambda Abstraction using bound variables within suitable lambda expressions. □

## 3 Transformation Rules and Strategies for Logic Programs

In this section we will present the 'rules + strategies' approach to program transformation in the case of logic programs. Logic programs compute relations rather than functions, and the *nondeterminism* inherent in relations does affect the various transformation techniques we have presented in the previous section in the case of functional programs. We will face new problems, but the key ideas of tupling, generalization, and need-for-folding still play an essential role and turn out to be very effective.

An interesting approach to program development and transformation in the presence of nondeterminism is also given in [Möller 91].

The syntax and the semantics of logic programs are defined as follows. We assume that we are given a first order language $L$ made out of a finite set of *function symbols* with arities (including at least one *constant*, that is, a 0-ary function symbol), a finite set of *predicate symbols*, and a countably infinite set of *variable symbols*. Function and predicate symbols are denoted by lower case letters, while variables are denoted by upper case letters.

Given a predicate symbol $p$ of arity $n$ and the terms $t_1, \ldots, t_n$ built out of function and variable symbols, we say that $p(t_1, \ldots, t_n)$ is an *atom*. A *goal* is a (possibly empty) conjunction of atoms.

A (*definite*) *clause*, say $C$, is a formula of the following form:

$$\forall X_1, \ldots, X_k . (A_1 \wedge \ldots \wedge A_m \rightarrow H)$$

which is also written as:

$$H \leftarrow A_1, \ldots, A_m,$$

where: i) $X_1, \ldots, X_k$ are the variables occurring in $A_1 \wedge \ldots \wedge A_m \rightarrow H$, ii) $A_1, \ldots, A_m$ with $m \geq 0$, is a goal, called the *body* of the clause and denoted by $bd(C)$, and iii) $H$ is an atom, called the *head* of the clause and denoted by $hd(C)$. By consistently changing the names of the variables in a clause we get a *variant* of that clause.

A (*definite*) *logic program* is a conjunction of clauses.

Given a term $t$ we denote by $vars(t)$ the set of variables occurring in $t$. The same notation will also be used for the variables occurring in atoms, goals, and clauses. A term (or an atom) is said to be *ground* if no variable occurs in it. Given a clause $C$ of the form $H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$, the *linking* variables of $A_1, \ldots, A_m$ in $C$ are those occurring in $A_1, \ldots, A_m$ and also in $H, B_1, \ldots, B_n$.

The *Herbrand universe* $H_L$ associated with $L$, is the set of terms which are built out of the function symbols of $L$. The *least Herbrand model* $M(P)$ of a program $P$ is the following set of ground atoms:

$$M(P) = \{p(t_1, \ldots, t_n) \mid p \text{ is a predicate symbol in } L,\ \{t_1, \ldots, t_n\} \subseteq H_L, \\ \text{and } P \models p(t_1, \ldots, t_n)\}.$$

For any given predicate symbol $p$ occurring in $L$, $M(p, P)$ is the subset of $M(P)$ consisting of atoms with predicate $p$. Two programs $P_1$ and $P_2$ are said to be *equivalent* w.r.t. the predicate $p$ iff $M(p, P_1) = M(p, P_2)$.

We will assume that the reader has some familiarity with the operational semantics of logic programs which is based on SLD-resolution [Lloyd 87].

### 3.1 Transformation Rules for Logic Programs

We now introduce the transformation rules which will be used for obtaining new logic programs from old ones.

i) *Definition Rule.* It consists in introducing a new clause which defines a new predicate in terms of already existing predicates. Clauses introduced by the definition rule are called *definition clauses*.

ii) *Unfolding Rule.* It consists in performing a resolution step. There are the following two main differences with respect to the unfolding rule in the functional case.

1. The resolution step produces a *unifying* substitution (not a *matching* substitution, as it happens in a rewriting step for functional programs). Thus, when we unfold a clause $C$ w.r.t. an atom $A$ in $bd(C)$ using a

clause $D$ in the current program, we replace $A$ by $bd(D)$ and we then apply to the resulting clause the substitution which unifies $A$ and $hd(D)$.

2. In the functional case we have assumed that the equations defining a program are mutually exclusive. Thus, by unfolding a given equation we may get at most one new equation. In contrast, in the logical case there may be several clauses whose heads are unifiable with an atom $A$ in the body of a clause $C$. As a result, by unfolding $C$ w.r.t. $A$ using the clauses of the current program, we may obtain several clauses, say $C_1, \ldots, C_n$. By an application of the unfolding rule we replace $C$ by (the conjunction of) all clauses $C_1, \ldots, C_n$.

iii) *Folding Rule.* It consists in replacing a subgoal, say $B$, of the body of a clause, say $C$, such that $B$ is an instance of the body of a definition clause, say $D$, by the corresponding instance of the head of $D$.

Similarly to the functional case, we also require that an application of the folding rule be the inverse of an application of the unfolding rule, in the sense that if a clause $F$ is derived by folding clause $C$ using the definition clasue $D$, then by unfolding $F$ using $D$ we may get back (a variant of) clause $C$.

There are some pitfalls to be avoided when applying the folding rule. For instance, we cannot fold clause $C$: $p(X) \leftarrow q(t(X))$ using $D$: $r \leftarrow q(Y)$, even though the body of $C$ is an instance of the body of $D$. Indeed, by replacing $q(t(X))$ by the head $r$ of $D$ we get the clause $p(X) \leftarrow r$, from which by unfolding we get $p(X) \leftarrow q(Y)$ which is different from clause $C$.

Notice that if instead of clause $D$ we consider the clause $E$: $r(Y) \leftarrow q(Y)$, where the variable $Y$ is an argument of the head predicate, then it is possible to fold clause $C$ using $E$. In general, it is the case that by considering extra variables as arguments of the head predicate we can perform folding steps which otherwise are impossible.

iv) *Goal Replacement Rule.* Let $P$ be a program, $C$ a clause in $P$, and $G_1$ a goal occurring in the body of $C$. Suppose that for some goal $G_2$ the following formula is true in the least Herbrand model $M(P)$ of $P$:

$$\forall X_1, \ldots, X_k. (\exists Y_1, \ldots, Y_m. G_1 \leftrightarrow \exists Z_1, \ldots, Z_n. G_2)$$

where: i) $X_1, \ldots, X_k$ are the linking variables of $G_1$ in $C$, ii) $\{Y_1, \ldots, Y_m\}$ $= vars(G_1) - \{X_1, \ldots, X_k\}$, iii) $\{Z_1, \ldots, Z_n\} = vars(G_2) - \{X_1, \ldots, X_k\}$, and iv) $\{Z_1, \ldots, Z_n\} \cap vars(C) = \{\}$. Then, in the body of $C$ we may replace $G_1$ by $G_2$.

v) *Clause Deletion Rule.* Let $P$ be a program and $C$ a clause in $P$. We get a new program $Q$ by deleting $C$ from $P$ if $C$ is true in $M(Q)$.

We may apply clause deletion in the following cases: i) $bd(C)$ contains an atom which is not unifiable with the head of any other clause in $P$, and ii) $C$ is *subsumed* by another clause $D$ in $P$, that is, there exists a substitution $\theta$ such that $hd(D)\theta = hd(C)$ and $bd(D)\theta$ occurs in $bd(C)$.

The application of the transformation rules preserves *soundness*, in the sense that if we derive program $P_2$ from program $P_1$, then for each predicate $p$ in $P_1$ we have that: $M(p, P_2) \subseteq M(p, P_1)$. By forcing some restrictions on the use of those transformation rules [Tamaki-Sato 84], we also preserve *completeness*, that is, we get: $M(p, P_2) \supseteq M(p, P_1)$.

The reader may easily verify that the use of the rules in the examples of program transformation we will present in this section, preserves both soundness and completeness.

Variants of the above transformation rules can be shown to be correct w.r.t. richer logic languages and other program semantics (see, for instance, [Aravindan-Dung 93, Bossi-Cocco 93, Bossi-Cocco-Etalle 92, Gardner-Shepherdson 91, Kanamori-Horiuchi 87, Kawamura-Kanamori 88, Maher 93, Proietti-Pettorossi 91a, Sato 92, Seki 91, Seki 93], and [Pettorossi-Proietti 94] for an overview).

### 3.2 Transformation Strategies for Logic Programs

Here we list some of the strategies we use for transforming logic programs.

i) *Predicate Tupling Strategy.* This strategy, also called *tupling*, for short, consists in selecting some atoms, say $A_1, \ldots, A_n$, with $n \geq 1$, occurring in the body of a clause $C$. We introduce a new predicate *newp* defined by a clause $T$ of the form:

$newp(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_n$

where $X_1, \ldots, X_k$ are the linking variables of $A_1, \ldots, A_n$ in $C$. We then look for the recursive definition of the predicate *newp* by performing some unfolding, goal replacement, and clause deletion steps followed by some folding steps using clause $T$.

We finally fold the atoms $A_1, \ldots, A_n$ in the body of clause $C$ using clause $T$.

The tupling strategy is often applied when $A_1, \ldots, A_n$ share some variables. The program improvements which can be achieved by using this strategy in the case of logic programs are similar to those which can be achieved by using the composition and/or the tupling strategies in the case of functional programs. In particular, we need to evaluate only once the subgoals which are common to the computations evoked by the tupled atoms $A_1, \ldots, A_n$. We can also avoid multiple visits of data structures and the construction of intermediate bindings [Debray 88, Proietti-Pettorossi 91b].

In addition, by using the tupling strategy sometimes we can obtain a new program version which simulates the execution of the initial program according to a computation rule different from the 'left-to-right' computation rule used by Prolog. An automated technique which applies this transformation strategy for reducing the nondeterminism of Prolog programs is the *Compiling Control* technique [Bruynooghe et al. 89].

As in the functional case, the folding steps needed for deriving the recursive definition of the predicate *newp* can often be performed only if we introduce some new predicates by means of definition clauses.

In the sequel we will consider the following three strategies which can be used for introducing those new predicates: the *loop absorption*, the *generalization*, and the *implicit definition* strategy.

In order to describe those strategies we will represent the process of performing unfolding and goal replacement steps starting from a clause, say $R$, as a tree of clauses, called *unfolding tree*. The root of the unfolding tree is clause $R$, and for each clause derived by applying either unfolding or goal replacement to a clause $N$ of the unfolding tree we construct a new son of $N$. In an unfolding tree we also have the usual relations of descendant clause and ancestor clause.

ii) *Loop Absorption Strategy.* Suppose that a clause $C$ in an unfolding tree has the form: $H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$, and the body of a descendant $D$ of $C$ contains an instance of $A_1, \ldots, A_m$. Suppose also that the clauses in the path from $C$ to $D$ have been generated by applying no transformation rule to $B_1, \ldots, B_n$.

The loop absorption strategy is applied by introducing a new predicate defined by the following clause $A$:

$$newp(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_m$$

where $\{X_1, \ldots, X_k\}$ is the minimum subset of $vars(A_1, \ldots, A_m)$ which is necessary to fold both $C$ and $D$ using a clause whose body is $A_1, \ldots, A_m$. We then look for the recursive definition of the predicate *newp*.

The loop absorption strategy is formally introduced in [Proietti-Pettorossi 90] and more details may be found in that paper. Similar strategies are used by some other techniques for program transformation, such as the Partial Evaluation technique (see Section 4) and the above mentioned Compiling Control.

iii) *Generalization Strategy.* Given a clause $C$ of the form: $H \leftarrow A_1, \ldots, A_m$, $B_1, \ldots, B_n$, we define a new predicate *genp* by a clause $G$ of the form:

$$genp(X_1, \ldots, X_k) \leftarrow GenA_1, \ldots, GenA_m$$

where $(GenA_1, \ldots, GenA_m)\theta = A_1, \ldots, A_m$, for a given substitution $\theta$, and $X_1, \ldots, X_k$ are the variables which are necessary to fold $C$ using $G$. We then fold $C$ using $G$ and we get:

$$H \leftarrow genp(X_1, \ldots, X_k)\theta, B_1, \ldots, B_n.$$

We finally look for the recursive definition of the predicate *genp*.

As already mentioned when describing the transformation strategies of functional programs, the generalization strategy originated in the area of Automated Theorem Proving [Boyer-Moore 75] and it has been used in a number of program transformation techniques.

A suitable form of the clause $G$ introduced by generalization can often be obtained by matching clause $C$ against one of its descendants, say $D$, in the unfolding tree which is considered during program transformation. In particular, we will consider the case where:

1. $D$ is the clause $K \leftarrow E_1, \ldots, E_m, F_1, \ldots, F_r$, and $D$ has been obtained from $C$ by applying no transformation rule to $B_1, \ldots, B_n$,

2. for $i = 1, \ldots, m$ the atom $E_i$ has the same predicate of $A_i$,

3. for $i = 1, \ldots, m$ $E_i$ is not an instance of $A_i$,

4. the goal $GenA_1, \ldots, GenA_m$ is the most concrete generalization of both $A_1, \ldots, A_m$ and $E_1, \ldots, E_m$, and

5. $\{X_1, \ldots, X_k\}$ is the minimum subset of $vars(GenA_1, \ldots, GenA_m)$ which is necessary to fold both $C$ and $D$ using a clause whose body is $GenA_1, \ldots, GenA_m$.

In our logical language we do not have abstraction operators, and thus, we cannot consider techniques similar to the Lambda Abstraction strategy which has been presented in the case of functional programs (see Section 2). However, we will show in Example 8 below that the effects of abstractions may sometimes be simulated in logic programming by exploiting the power of the unification mechanism.

A strategy which is analogous to the generalization from functions to functions by implicit definition presented in Section 2, can be obtained as a particular case of the following one [Proietti-Pettorossi 94].

iv) *Implicit Definition Strategy.* Let $P$ be a program and $C$ one of its clauses of the form: $H \leftarrow A_1, \ldots, A_m, G_1$, where $G_1$ is a goal made out of one or more atoms. Suppose that we want to replace $G_1$ by a goal of the form: '$G_2, impl(\ldots)$', where *impl* is a new predicate symbol and the arguments of *impl* are variables. In order to do so, according to the Goal Replacement Rule we have to look for a set *Expl* of clauses such that in the least Herbrand model $M(P \wedge Expl)$ the following formula holds:

$$\forall X_1, \ldots, X_k.(\exists Y_1, \ldots, Y_m.G_1 \leftrightarrow \exists Z_1, \ldots, Z_n.(G_2, impl(\ldots))) \qquad (1)$$

where: i) $X_1, \ldots, X_k$ are the linking variables of $G_1$ in $C$, ii) $\{Y_1, \ldots, Y_m\} = vars(G_1) - \{X_1, \ldots, X_k\}$, iii) $\{Z_1, \ldots, Z_n\} = vars(G_2, impl(\ldots)) - \{X_1, \ldots, X_k\}$, and iv) $\{Z_1, \ldots, Z_n\} \cap vars(C) = \{\}$.

The set *Expl* provides an explicit definition of the predicate *impl* which is implicitly defined by formula (1). A practical method to find the set *Expl* consists of the following four steps:

*Step* 1. We consider two different definitions of a new predicate, say *newp*, by introducing two clauses with body $G_1$ and $(G_2, impl(\ldots))$, respectively, that is:

$D_1$. $newp(X_1, \ldots, X_k) \leftarrow G_1$

$D_2$. $newp(X_1, \ldots, X_k) \leftarrow G_2, impl(\ldots)$

*Step* 2. We look for the recursive definition of *newp* by using the program $P \wedge D_1$ thereby producing a program $Q_1$ equivalent to $P \wedge D_1$.

*Step* 3. We then unfold every occurrence of *newp* in $Q_1$ by using clause $D_2$ and get a program $Q_2$.

*Step* 4. We finally obtain the set *Expl* by requiring that by unfolding the occurrences of $impl(V_1, \ldots, V_h)$ in $P \wedge D_2$ using the clauses in *Expl* we derive program $Q_2$.

Notice that the validity of the formula (1) in $M(P \wedge Expl)$ follows from the fact that both programs $(P \wedge D_1 \wedge Expl)$ and $(P \wedge D_2 \wedge Expl)$ can be transformed into $(Q_1 \wedge Expl)$, and therefore, the two definitions of *newp* are equivalent. Indeed, by Step 2 we have that $(P \wedge D_1 \wedge Expl)$ can be transformed into $(Q_1 \wedge Expl)$. On the other hand, by Step 4 we have that $(P \wedge D_2 \wedge Expl)$ can be transformed into $(Q_2 \wedge Expl)$ by performing an unfolding step, and by Step 3 program $Q_1$ can be obtained from $Q_2$ by performing a folding step.

The above implicit definition strategy generalizes the one presented in [Kanamori-Maeji 86] and it is related to the technique described in [Kott 82] for proving properties of functional programs based on unfold/fold transformations.

### 3.3 Examples of Transformations of Logic Programs

In the following Example 7 we will show that the rules and strategies described above can be used for reducing the amount of nondeterminism in a given program. In particular, we will make use of the tupling, loop absorption, and implicit definition strategies. We will end this section by demonstrating in Example 8 the use of the generalization strategy.

**Example 7 (Prime Numbers)** Let us consider the problem of computing the list $P$ of the first $N$ prime numbers for any given natural number $N > 0$. To this purpose we define a relation $primes(N, P)$ by means of the following program, called *Primes*:

---

7.1 $primes(N, P) \leftarrow initial(L),\ not\_div\_by\_smaller(L, P),\ length(P, N)$
7.2 $initial([2]) \leftarrow$
7.3 $initial([H, K|T]) \leftarrow initial([K|T]),\ plus(K, 1, H)$
7.4 $not\_div\_by\_smaller([\,], [\,]) \leftarrow$
7.5 $not\_div\_by\_smaller([H|T], P) \leftarrow not\_div\_by\_smaller(T, Q),$
$div\_tests(H, Q, P)$
7.6 $div\_tests(H, Q, [H|Q]) \leftarrow not\_div\_any(H, Q)$
7.7 $div\_tests(H, Q, Q) \leftarrow div\_some(H, Q)$
7.8 $length([\,], 0) \leftarrow$
7.9 $length([H|T], N) \leftarrow length(T, N1),\ plus(N1, 1, N)$

---

where we assume that:
i) $plus(X, Y, Z)$ holds iff $X + Y = Z$, ii) $not\_div\_any(H, L)$ holds iff $H$ is not divisible by any number of the list $L$, and iii) $div\_some(H, L)$ holds iff $H$ is divisible by a number in $L$.

Notice that: i) $initial(L)$ holds iff $L$ is the list $[M, M-1, \ldots, 2]$ of consecutive natural numbers for some number $M (\geq 2)$, ii) $not\_div\_by\_smaller(L, P)$ holds iff $P$ is the sublist of $L$ such that an element of $L$, say $X$, is in $P$ iff for each $Y$ which is to the right of $X$ in $L$, $X$ is not divisible by $Y$, and iii) $length(P, N)$ holds iff the length of the list $P$ is $N$.

Let us assume the left-to-right, depth-first Prolog evaluation strategy and suppose that the program *Primes* is evaluated for a goal of the form $primes(N, P)$, where $N$ is bound to a positive number and $P$ is an unbound variable. The execution corresponding to a goal of that form has a very large degree of nondeterminism. Indeed, the program *Primes* generates an initial segment, say $L$, of numbers greater than 1, then it constructs the list $P$ of the prime numbers contained in $L$, and it finally matches the length of $P$ with the given $N$. If the matching fails, the program *Primes* generates the next initial segment of natural numbers and it constructs the prime numbers contained in that segment, without taking advantage of the results of the divisibility tests performed by *div_tests* before failure. If we assume that the evaluation of $div\_tests(H, Q, R)$ performs $O(length(Q))$ tests of divisibility, then the program *Primes* performs $O(K \times N^2)$ tests of divisibility, where $K$ is the largest generated prime number and $N$ is the number of generated primes (that is, $N$ is the length of $P$).

We will improve the program *Primes* by i) avoiding the construction of the intermediate lists which are shared by the predicates *initial*, *not_div_by_smaller*, and *length*, and ii) reducing the nondeterminism of the derived program.

For clarity, we will break our derivation into five parts.

Part 1. *Application of the tupling and loop absorption strategies for avoiding intermediate lists.*

We begin our transformation process by applying the tupling strategy to the atoms $initial(L)$, $not\_div\_by\_smaller(L, P)$, and $length(P, N)$ which share the variables $L$ and $P$ in the body of clause 7.1. Since the atoms to be tupled together constitute the whole body of clause 7.1 defining the predicate *primes*, we do not need to introduce (by the definition rule) a new predicate and we look for the recursive definition of the predicate *primes*. By unfolding, from clause 7.1 we get:

7.10 $primes(1, [2]) \leftarrow$

7.11 $primes(N, [H|P]) \leftarrow initial([K|T]),\ plus(K, 1, H),$
$\quad not\_div\_by\_smaller([K|T], P),\ not\_div\_any(H, P),$
$\quad length(P, N1),\ plus(N1, 1, N)$

7.12 $primes(N, P) \leftarrow initial([K|T]),\ plus(K, 1, H),$
$\quad not\_div\_by\_smaller([K|T], P),$
$\quad div\_some(H, P),\ length(P, N)$

The bodies of clauses 7.11 and 7.12 both contain instances of the body of clause 7.1. However, we cannot perform any folding step using clause 7.1 because the variable $L$ of clause 7.1 does not occur in the head. Thus, we are in a situation where we may apply the loop absorption strategy. By doing so, we introduce the new clause:

7.13 $new1(L, N, P) \leftarrow initial(L),\ not\_div\_by\_smaller(L, P),\ length(P, N)$

whose body is equal to that of clause 7.1, and whose head contains the variables which are required for folding.

By performing a few unfolding and folding steps we get the following recursive definition of $new1$:

7.14 $new1([2], [2], 1) \leftarrow$

7.15 $new1([H, K|T], [H|P], N) \leftarrow new1([K|T], P, N1),\ plus(K, 1, H),$
$\qquad not\_div\_any(H, P),\ plus(N1, 1, N)$

7.16 $new1([H, K|T], P, N) \leftarrow new1([K|T], P, N),\ plus(K, 1, H),$
$\qquad div\_some(H, P)$

By folding we obtain a definition of the predicate *primes* in terms of the new predicate $new1$:

7.17 $primes(N, P) \leftarrow new1(L, P, N)$

Part 2. *Fusion of clauses with common atoms.*

Now we notice that the predicate $new1$ occurs in the body of both clause 7.15 and clause 7.16. This fact may cause the repeated evaluation of identical atoms when backtracking occurs. We can avoid this repeated evaluation if we replace clauses 7.15 and 7.16 by the following three clauses:

7.18 $new1([H, K|T], P, N) \leftarrow new1([K|T], P1, N1), p(K, H, P, N, P1, N1)$

7.19 $p(H, K, [H|P], N, P, N1) \leftarrow plus(K, 1, H), not\_div\_any(H, P),$
$\qquad plus(N1, 1, N)$

7.20 $p(H, K, P, N, P, N) \leftarrow plus(K, 1, H), div\_some(H, P)$

This transformation is sometimes called *clause fusion* [Debray-Warren 88, Deville 90] and its correctness is ensured by the fact that clauses 7.15 and 7.16 can be obtained by unfolding $p$ in clause 7.18 using clauses 7.19 and 7.20.

The derived program consisting of clauses 7.17, 7.14, 7.18, 7.19, and 7.20, is more efficient than the initial program because it avoids the visit of some lists. In particular, the atom $new1(L, P, N)$, while generating a segment $L$ of integers, collects in the list $P$ the prime numbers occurring in $L$ and also computes the length $N$ of $P$. On the contrary, the initial program performs the same computations by using distinct predicates which first construct and then visit the lists $L$ and $P$.

Part 3. *Use of the implicit definition strategy for reducing nondeterminism.*

In order to allow a clearer presentation of the transformations which we will indicate below, we begin by rewriting the program we have derived so far, as follows:

---

7.17 $primes(N, P) \leftarrow new1(L, P, N)$

7.21 $new1(L, P, N) \leftarrow L = [2],\ P = [2],\ N = 1$

7.18* $new1(L, P, N) \leftarrow new1(L1, P1, N1),\ q(L, P, N, L1, P1, N1)$

7.19* $q([H, K|T], [H|P1], N, [K|T], P1, N1) \leftarrow$
$\qquad plus(K, 1, H),\ not\_div\_any(H, P1),\ plus(N1, 1, N)$

7.20* $q([H, K|T], P1, N, [K|T], P1, N) \leftarrow plus(K, 1, H), div\_some(H, P1)$

---

The correctness of this rewriting follows from the fact that by unfolding the equalities in clause 7.21 we get clause 7.14 and by unfolding $q$ in clause 7.18* we get clauses 7.15 and 7.16 which are equivalent to clauses 7.18, 7.19, and 7.20.

The main cause of nondeterminism in the above program is clause 7.18*, where backtracking steps may be generated during the evaluation of the call $q(L, P, N, L1, P1, N1)$ *after* the evaluation of the recursive call of $new1$. In order to reduce nondeterminism we want to anticipate the computation of $q$, so that only nonfailing recursive calls to $new1$ are made. This goal can be achieved by obtaining a tail recursive version of clause 7.18*. In other words, since we are assuming a left-to-right evaluation order of the atoms in a goal, we would like to get a clause equivalent to 7.18* in which the recursive call to $new1$ is the rightmost one.

The reader should notice that, in general, we cannot get an efficient tail recursive Prolog program by simply rearranging the order of the atoms in the body of the clauses. Indeed, in the case of clause 7.18* by moving the atom $q(L, P, N, L1, P1, N1)$ to the left of the atom $new1(L1, P1, N1)$ we get a tail recursive clause, but the nondeterminism of the program is increased, at least for the class of goals we consider. The formal study of program properties which depend on the form of the goals, such as nondeterminism, can be done by using abstract interpretations (see, for instance, [Jones-Søndergaard 87]).

We will now transform clause 7.18* into a tail recursive clause of the following form:

7.22 $new1(L, P, N) \leftarrow B[L, P, N, \overline{X}],\ New2[L, P, N, \overline{X}]$

where: i) the notation $G[\ldots]$ is used here and in the sequel for denoting that a given goal (or atom) $G$ depends on some of the variables occurring in the list $[\ldots]$, ii) $B$ is a goal made out of atoms with predicate symbols $q$ and '=', iii) $New2$ is an atom with a possibly new predicate symbol, and iv) $\overline{X}$ is an $n$-tuple of variables.

We now apply the implicit definition strategy to compute the goal $B$ and the explicit definition, say $Expl2$, for the atom $New2$. According to that strategy, we perform the following four steps:

*Step* 1. We consider a new predicate $new3$ and two different definition clauses for it:

$D_1$. $new3(L, P, N) \leftarrow new1(L1, P1, N1), q(L, P, N, L1, P1, N1)$

whose body is identical to that of clause 7.18*, and

$D_2$. $new3(L, P, N) \leftarrow B[L, P, N, \overline{X}],\ New2[L, P, N, \overline{X}]$

whose body is identical to that of clause 7.22.

*Step* 2. By unfolding $new1$ in clause $D_1$ and then folding using clause $D_1$ itself, we get the following recursive definition of $new3$:

7.23 $new3(L, P, N) \leftarrow L1 = [2],\ P1 = [2],\ N1 = 1,\ q(L, P, N, L1, P1, N1)$

7.24 $new3(L, P, N) \leftarrow new3(L2, P2, N2),\ q(L, P, N, L2, P2, N2)$

*Step* 3. We then unfold $new3$ in clause 7.24 by using clause $D_2$. Thus, we replace clause 7.24 by the following one:

7.25 $new3(L, P, N) \leftarrow B[L2, P2, N2, \overline{X}],\ New2[L2, P2, N2, \overline{X}],$
$\qquad q(L, P, N, L2, P2, N2)$

*Step* 4. We now assume that clauses 7.23 and 7.25 are obtained by unfolding only the atom $New2$ in clause $D_2$ using the set $Expl2$ of unknown clauses. By this assumption we will be able to determine the goal $B$ and the set $Expl2$ of clauses which explicitly define the atom $New2$, as we now indicate.

Since we assume that $B[L,P,N,\overline{X}]$ is not unfolded, an occurrence of it should be in the body of clause 7.23. Among many possible choices we may take $B[L,P,N,\overline{X}]$ to be the goal $(L1=[2], P1=[2], N1=1)$, thereby instantiating $\overline{X}$ to the triplet of variables $\langle L1, P1, N1\rangle$. Thus, clauses $D_2$ and 7.25 can be rewritten as follows:

7.26 $new3(L,P,N) \leftarrow L1=[2], P1=[2], N1=1, New2[L,P,N,L1,P1,N1]$

7.27 $new3(L,P,N) \leftarrow L1=[2],\ P1=[2],\ N1=1,$
$New2[L2,P2,N2,L1,P1,N1],\ q(L,P,N,L2,P2,N2)$

According to our hypotheses, by unfolding $New2$ in clause 7.26 we should obtain clauses 7.23 and 7.27. Therefore, the clauses which define the atom $New2$ with the new predicate symbol $new2$ are:

7.28 $new2(L,P,N,L1,P1,N1) \leftarrow q(L,P,N,L1,P1,N1)$
7.29 $new2(L,P,N,L1,P1,N1) \leftarrow new2(L2,P2,N2,L1,P1,N1),$
$q(L,P,N,L2,P2,N2)$

Clauses 7.28 and 7.29 constitute the set $Expl2$ which we wanted to compute.

As a final step of the application of the implicit definition strategy, we replace clause 7.18* by the following tail recursive clause:

7.30 $new1(L,P,N) \leftarrow L1=[2], P1=[2], N1=1, new2(L,P,N,L1,P1,N1)$

Thus, the version of the program *Primes* we have derived so far, is made out of the following clauses:

---

7.17 $primes(N,P) \leftarrow new1(L,P,N)$
7.21 $new1(L,P,N) \leftarrow L=[2],\ P=[2],\ N=1$
7.30 $new1(L,P,N) \leftarrow L1=[2], P1=[2], N1=1, new2(L,P,N,L1,P1,N1)$
7.28 $new2(L,P,N,L1,P1,N1) \leftarrow q(L,P,N,L1,P1,N1)$
7.29 $new2(L,P,N,L1,P1,N1) \leftarrow new2(L2,P2,N2,L1,P1,N1),$
$q(L,P,N,L2,P2,N2)$

---

together with the clauses for $q$, *not_div_any*, *div_some*, and *plus*.

Part 4. *Further use of the implicit definition strategy for reducing nondeterminism.*

Unfortunately, clause 7.29 introduced during the transformation of clause 7.18* into tail recursive form, is *not* tail recursive. Thus, we apply again the implicit definition strategy for transforming clause 7.29 into one of the form:

$$7.31\quad new2(L,P,N,L1,P1,N1) \leftarrow C[L,P,N,L1,P1,N1,\overline{Y}],\; New4[L,P,N,L1,P1,N1,\overline{Y}]$$

where $C$ is a conjunction of atoms with predicates $q$ and '$=$', while $New4$ is an atom with a possibly new predicate symbol. In order to determine the form of the goal $C$ and the explicit definition of the atom $New4$, we consider a new predicate $new5$ and two different definitions for it, corresponding to the bodies of clauses 7.29 and 7.31, respectively:

$$E_1.\quad new5(L,P,N,L1,P1,N1) \leftarrow new2(L2,P2,N2,L1,P1,N1),\; q(L,P,N,L2,P2,N2)$$

$$E_2.\quad new5(L,P,N,L1,P1,N1) \leftarrow C[L,P,N,L1,P1,N1,\overline{Y}],\; New4[L,P,N,L1,P1,N1,\overline{Y}]$$

A derivation similar to the derivation shown above when constructing the explicit definition $Expl2$ of $new2$, allows us to rewrite clause $E_2$ as follows:

$$7.32\quad new5(L,P,N,L1,P1,N1) \leftarrow q(L2,P2,N2,L1,P1,N1),\; new4(L,P,N,L2,P2,N2)$$

with the following explicit definition of $new4$:

$$7.33\quad new4(L,P,N,L1,P1,N1) \leftarrow q(L,P,N,L1,P1,N1)$$

$$7.34\quad new4(L,P,N,L1,P1,N1) \leftarrow new4(L2,P2,N2,L1,P1,N1),\; q(L,P,N,L2,P2,N2)$$

The clauses for $new4$ are identical (up to the name of the predicate) to the clauses for $new2$. Therefore, we may forget about clauses 7.33 and 7.34, and we may replace the body of clause 7.29 by the body of clause 7.32 with $new2$ written instead of $new4$. Thus, instead of clause 7.29, we get the following tail recursive clause:

$$7.35\quad new2(L,P,N,L1,P1,N1) \leftarrow q(L2,P2,N2,L1,P1,N1),\; new2(L,P,N,L2,P2,N2)$$

The current program version is tail recursive and it is made out of the following clauses:

---

7.17 $primes(N, P) \leftarrow new1(L, P, N)$

7.21 $new1(L, P, N) \leftarrow L = [2],\ P = [2],\ N = 1$

7.30 $new1(L, P, N) \leftarrow L1 = [2], P1 = [2], N1 = 1,\ new2(L, P, N, L1, P1, N1)$

7.28 $new2(L, P, N, L1, P1, N1) \leftarrow q(L, P, N, L1, P1, N1)$

7.35 $new2(L, P, N, L1, P1, N1) \leftarrow q(L2, P2, N2, L1, P1, N1),$
$new2(L, P, N, L2, P2, N2)$

---

together with the clauses for $q$, *not_div_any*, *div_some*, and *plus*.

A straightforward transformation, consisting of the fusion of clauses 7.28 and 7.35, and the unfolding of the predicates $new1$, $new2$, $q$, and '$=$', allows us to get the following program:

---

7.10 $primes(1, [2]) \leftarrow$

7.36 $primes(N, P) \leftarrow new6(L, P, N, [3, 2], [3, 2], 2)$

7.37 $new6(L, P, N, L, P, N) \leftarrow$

7.38 $new6(L, P, N, [K|T], P1, N1) \leftarrow plus(K, 1, H), plus(N1, 1, N2),$
$not\_div\_any(H, P1),$
$new6(L, P, N, [H, K|T], [H|P1], N2)$

7.39 $new6(L, P, N, [K|T], P1, N1) \leftarrow plus(K, 1, H), div\_some(H, P1),$
$new6(L, P, N, [H, K|T], P1, N1)$

---

Part 5. *Elimination of unnecessary variables by applying the tupling strategy.*

The binding computed for the variable $L$ in clause 7.36 is not explicitly used for describing the relation $primes(N, P)$, because $L$ does not occur in the head. To avoid the occurrence of that variable, we apply the tupling strategy for $n = 1$, that is, we tuple one atom only. Thus, we introduce a new predicate $new7$ defined as follows:

7.40 $new7(P, N, L1, P1, N1) \leftarrow new6(L, P, N, L1, P1, N1)$

By folding clause 7.36 using clause 7.40 and then deriving the recursive definition of $new7$ we eliminate the unnecessary variable $L$. We get the following final program:

---

7.10 $primes(1, [2]) \leftarrow$

7.41 $primes(N, P) \leftarrow new7(P, N, [3, 2], [3, 2], 2)$

7.42 $new7(P, N, L, P, N) \leftarrow$

7.43 $new7(P, N, [K|T], P1, N1) \leftarrow plus(K, 1, H), not\_div\_any(H, P1),$
$plus(N1, 1, N2),$
$new7(P, N, [H, K|T], [H|P1], N2)$

7.44 $new7(P, N, [K|T], P1, N1) \leftarrow plus(K, 1, H), div\_some(H, P1),$
$new7(P, N, [H, K|T], P1, N1)$

---

This program keeps a list $P1$ of the prime numbers generated so far together with its length $N1$ (see the fourth and fifth arguments of $new7$, respectively), and it terminates when $N1$ reaches the desired value $N$ (see clause 7.42). The list $P1$ of prime numbers is updated by considering a number at a time and testing its divisibility by the primes generated so far. Thus, this final program performs $O(K \times N)$ divisibility tests, where $K$ is the largest generated prime number and $N$ is the number of generated primes. □

The following final example shows the use of the generalization strategy.

**Example 8 (Minimal Leaf Replacement)** Suppose that we are given a binary tree, say *InTree*, whose leaves are labelled by numbers. We want to obtain another tree, say *OutTree*, of the same shape with all its leaves replaced by their minimal value.

A simple two-visit algorithm can be given as follows. We first compute the minimal leaf value, say *Min*, of *InTree*, and then we visit again that tree for replacing its leaves by *Min*. A logic program which realizes this algorithm is as follows:

---

8.1 $mintree(InTree, OutTree) \leftarrow minleaves(InTree, Min),$
$replace(Min, InTree, OutTree)$

8.2 $minleaves(tip(N), N) \leftarrow$

8.3 $minleaves(tree(L, R), Min) \leftarrow minleaves(L, MinL),\ minleaves(R, MinR),$
$min(MinL, MinR, Min)$

8.4 $replace(M, tip(N), tip(M)) \leftarrow$

8.5 $replace(Min, tree(InL, InR), tree(OutL, OutR)) \leftarrow$
$replace(Min, InL, OutL),\ replace(Min, InR, OutR)$

---

where $min(M1, M2, M)$ holds iff $M$ is the minimum number of $M1$ and $M2$.

We would like to derive a program which traverses *InTree* only once. To this aim we may apply the tupling strategy to the predicates *minleaves* and *replace* which share the argument *InTree*. As in the first application of the tupling strategy in the previous Example 7, a new definition corresponding to the body of clause 8.1 is not necessary. We only need to look for the recursive definition of the predicate *mintree*. After some unfolding steps, we get:

8.6 $mintree(tip(N), tip(N)) \leftarrow$

8.7 $mintree(tree(InL, InR), tree(OutL, OutR)) \leftarrow$
$minleaves(InL, MinL), minleaves(InR, MinR),$
$min(MinL, MinR, Min), replace(Min, InL, OutL),$
$replace(Min, InR, OutR)$

Now, looking for a fold of the goal:

'$minleaves(InL, MinL), replace(Min, InL, OutL)$'

using clause 8.1, we realize that no matching is possible because this goal is not an instance of '$minleaves(InTree, Min), replace(Min, InTree, OutTree)$'. Thus, we apply the generalization strategy and we introduce the following clause:

$G.$ $genmintree(InTree, M1, M2, OutTree) \leftarrow minleaves(InTree, M1),$
$replace(M2, InTree, OutTree)$

where $bd(G)$ is the most concrete generalization of the two goals to be folded, that is, '$minleaves(InL, MinL), replace(Min, InL, OutL)$' and the body of clause 8.1. By folding clause 8.1 we get:

8.8 $mintree(InTree, OutTree) \leftarrow genmintree(InTree, Min, Min, OutTree)$

We are now left with the problem of finding the recursive definition of the predicate *genmintree*. This is an easy task, because we can perform the unfolding steps corresponding to those leading from clause 8.1 to clauses 8.6

and 8.7, and then we can use clause $G$ for folding. After those steps we get the following final program, which performs the desired tree transformation in one visit.

---

8.8 $\quad mintree(InTree, OutTree) \leftarrow genmintree(InTree, Min, Min, OutTree)$

8.9 $\quad genmintree(tip(N), N, M, tip(M)) \leftarrow$

8.10 $\quad genmintree(tree(InL, InR), M1, M2, tree(OutL, OutR)) \leftarrow$
$\qquad genmintree(InL, M1L, M2, OutL),$
$\qquad genmintree(InR, M1R, M2, OutR),$
$\qquad min(M1L, M1R, M1)$

---

Let us now consider an execution of the derived program for evaluating a goal of the form '$mintree(t, T)$', where $t$ is a ground binary tree and $T$ is an unbound variable. During the visit of the input tree $t$, the predicate *genmintree* both computes the minimal leaf value $M1$ and replaces the leaves using the *unbound* variable $M2$. The instantiation of $M2$ to the minimal leaf value is performed by the unification of the variables $M1$ and $M2$ due to the first clause of our final program.

When writing a similar program in an imperative language, in order to realize the leaf replacement in one visit we should use pointers to the location which at the end holds the minimal leaf value. As already seen in Example 6 of Section 2, the derivation of programs which use pointers in a disciplined way can be done by applying the Lambda Abstraction strategy. □

## 4 Other Techniques for Transforming Functional and Logic Programs

It is difficult to exhaustively report on the various techniques and methodologies which have been proposed in the literature for deriving functional or logic programs by transformation. As an introduction the reader may refer to [Feather 87, Partsch 90, Möller et al. 93, Pettorossi-Proietti 94].

Below we will present the following techniques:

1. the *Schemata Approach* (one may refer, for instance, to [Partsch 90] for a

survey in the functional case, and to [Brough-Hogger 91, Deville-Burnay 89, Fuchs-Fromherz 92, Marakakis-Gallagher 94, Seki-Furukawa 87] in the logic case),

2. the *Partial Evaluation* (or *Mixed Computation*) technique [Ershov 82, Jones et al. 93], and

3. the *Finite Differencing* technique [Paige-Koenig 82], and compare them with the strategies we have considered above.

For other methods, such as: i) combinatory techniques [Turner 79], ii) supercombinator methods [Hughes 82], iii) local recursions [Bird 84a], iv) promotion strategies [Bird 84b], v) lambda liftings [Johnsson 85], and vi) lambda hoistings [Takeichi 87], we suggest to look at the original papers.

There are some other techniques which have been proposed in the literature for transforming functional and logic programs. They are related to higher order features, in the sense that *continuations* and *communications* among function calls and predicate evaluations, are created for the derivation of very efficient programs. The reader may refer to [Wand 80, Pettorossi-Skowron 82] for the functional case and to [Tarau-Boyer 90] for the logic case.

For an introductory view of some methods for the transformation and the derivation of parallel and/or concurrent programs the reader may refer to [Paige et al. 93].

### 4.1 Schemata Approach

In the *schemata approach* to program transformation the programmer is given a *dictionary*, that is, a set of pairs of program schemata. If a program $P$ matches a schema $S_1$ by a substitution $\theta$ and $\langle S_1, S_2 \rangle$ is a pair in the dictionary, we can replace $P$ by $S_2\ \theta$. The dictionary ensures that $P$ and $S_2\ \theta$ are semantically equivalent, and also that $S_2\ \theta$ is an improvement over $P$ w.r.t. some complexity measure.

We presented some instances of these schemata transformations in the case of functional languages in Examples 2, 3, 5, and 6 above (see Figures

2, 4, 5, and 6, respectively). Similar transformations can also be applied in the case of logic languages. Moreover, in [Seki-Furukawa 87] it is shown that by schemata transformations one can reduce the nondeterminism of generate-and-test programs.

An advantage of the schemata approach over the 'rules + strategy' approach is that the application of a schema transformation is not computationally expensive, because it only requires the application of a substitution. However, the choice of a suitable schema transformation in the given dictionary of the available transformations may be time consuming, because it also requires to look for the existence of a matching substitution.

A drawback of the schemata approach is the impossibility of performing any kind of transformation when the program at hand is not an instance of any schema in the dictionary.

As an example of the schemata approach we want to mention the *invertion of the flow of computation* [Boiten 90]. This technique can be applied when the initial program is defined in terms of operators which have inverses or satisfy some suitable properties, like associativity or commutativity. By applying this technique, repeated evaluations of functions are often avoided and efficiency is improved. For instance, by inverting the flow of computation of the initial program:

$$\mathit{fib}(n) = \textbf{if } n \leq 1 \textbf{ then } 1 \textbf{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2) \qquad \text{for } n \geq 0$$

which computes the Fibonacci function in exponential time, we get the linear time program:

$$\begin{array}{ll} \mathit{fib}(n) = D(0, n, 1, 1) & \text{for } n \geq 0 \\ D(m, n, a, b) = \textbf{if } m = n \textbf{ then } b \textbf{ else } D(m+1, n, a+b, a) & \text{for } m, n \geq 0. \end{array}$$

The flow of computation has been inverted in the sense that the computation of the initial program is top-down, while the computation of the final program is bottom-up.

The above transformation can be performed in one step only, by applying a suitable schema equivalence, after having checked the validity of the required conditions on the operators [Boiten 90]. The correctness of that transformation is based on the fact that the successor function is the inverse of the predecessor function and for $n \geq 0$ the unfolding of $\mathit{fib}(n)$ eventually generates the calls of $\mathit{fib}(0)$ and/or $\mathit{fib}(1)$ only.

Obviously, in many cases, as in this fibonacci example, instead of applying a schema equivalence, one can derive the final program via the application of the transformation rules listed in Section 2.

### 4.2 Partial Evaluation

*Partial Evaluation* (related to *Program Division* [Jones 87] and *Variable Splitting* [Bjørner et al. 88]) is defined via a function $\mathit{PEval} : \mathit{Programs} \times \mathit{Data}_1 \rightarrow \mathit{Programs}$, which given a program $P$ in *Programs* and a data $d_1$ in the domain $\mathit{Data}_1$, produces a new program, called the *residual program*, as we now specify. Suppose that the domain of $P$ is isomorphic to $\mathit{Data}_1 \times \mathit{Data}_2$, where $\mathit{Data}_2$ is a suitable domain. Then for every data $d_2$ in $\mathit{Data}_2$ the residual program $\mathit{PEval}(P, d_1)$ of $P$ w.r.t. $d_1$ is a program which satisfies the following equation:

$P\langle d_1, d_2\rangle = (\mathit{PEval}(P, d_1))\, d_2$.

Partial evaluation is used for improving program performance when $d_1$, called the *static* data, is known at compile-time. The residual program is obtained by 'importing' the information relative to the static data $d_1$ into the program $P$. The data $d_2$ is known only at run-time and it is called the *dynamic* data.

Partial evaluation is also used for automatically producing compilers from interpreters [Futamura 71].

Partial evaluation can be viewed as the inverse of Lambda Abstraction in the sense that the latter is defined by the function *Lambda*: $\mathit{Programs} \rightarrow (\mathit{Programs} \times \mathit{Data})$, which 'exports' information from a given program and, for any program $P$ and data $d$, satisfies the following equation [Pettorossi-Proietti 87]:

$P\, d = P_1\langle d_1, d\rangle$ where $\mathit{Lambda}(P) = \langle P_1, d_1\rangle$.

The partial evaluation technique has also been widely applied in the logic programming context, where it has been first studied in [Komorowski 82]. In this context, the partial evaluation (also called *partial deduction*) of a program $P$ w.r.t. a goal $G$ produces a program $P_G$ such that, every ground goal $G\theta$ is true in $M(P)$ iff $G\theta$ is true in $M(P_G)$.

The basic theoretical results about the correctness of partial evaluation of logic programs, as well as the extension of this technique to the case of general logic programs with negation, can be found in [Lloyd-Shepherdson 91]. For a tutorial introduction to partial evaluation of logic programs the reader may look at [Gallagher 93].

In the following example we illustrate the use of the partial evaluation technique for the derivation of an efficient acceptor for a regular expression given at compile-time, starting from a 'universal acceptor' which works for any regular expression given at run-time.

In the example we will use the transformation rules and strategies presented in Section 3, instead of ad hoc techniques for partial evaluation such as those considered in [Lloyd-Shepherdson 91, Gallagher 93].

**Example 9 (Acceptors for Regular Expressions)** Let us consider the following logic program, called *Accepts*, for testing whether or not a word $W$ in $\{a, b\}^*$ is accepted by the regular expression $R$ [Mogensen-Bondorf 93]. This test is performed by a goal of the form $accepts(R, W)$, where the word $W$ is represented as a (possibly empty) list of $a$'s and $b$'s, and $R$ is a term denoting a regular expression in the set $Reg$ defined as follows:

$Reg ::= void \mid a \mid b \mid (Reg; Reg) \mid (Reg + Reg) \mid Reg^*$,

where *void* is the regular expression which does not accept any word.

---

$accepts(R, [\,]) \leftarrow accepts\_empty(R)$
$accepts(R, [S|W]) \leftarrow next(R, S, R1), accepts(R1, W)$

$accepts\_empty((R1; R2)) \leftarrow accepts\_empty(R1), accepts\_empty(R2)$
$accepts\_empty((R1 + R2)) \leftarrow accepts\_empty(R1)$
$accepts\_empty((R1 + R2)) \leftarrow accepts\_empty(R2)$
$accepts\_empty(R^*) \leftarrow$

$next(R, S, R1) \leftarrow transf(R, TR), deriv(TR, S, L), make\_regexpr(L, R1)$

$transf(void, void) \leftarrow$
$transf(S, S) \leftarrow symbol(S)$
$transf((void; R), void) \leftarrow$

$transf((S;R),(S;R)) \leftarrow symbol(S)$
$transf(((R1;R2);R3),R4) \leftarrow transf((R1;(R2;R3)),R4)$
$transf(((R1+R2);R3),R4) \leftarrow transf(((R1;R3)+(R2;R3)),R4)$
$transf((R1^*;R2),R3) \leftarrow transf((R2+(R1;(R1^*;R2))),R3)$
$transf((R1+R2),(R3+R4)) \leftarrow transf(R1,R3), transf(R2,R4)$
$transf(R1^*,(void^*+R2)) \leftarrow transf((R1;R1^*),R2)$
$deriv(void,S,[\,]) \leftarrow$
$deriv(void^*,S,[\,]) \leftarrow$
$deriv(S,S,[void^*]) \leftarrow symbol(S)$
$deriv(S1,S2,[\,]) \leftarrow symbol(S1), symbol(S2), S1 \neq S2$
$deriv((S;R),S,[R]) \leftarrow symbol(S)$
$deriv((S1;R),S2,[\,]) \leftarrow symbol(S1), symbol(S2), S1 \neq S2$
$deriv((R1+R2),S,L) \leftarrow deriv(R1,S,L1), deriv(R2,S,L2), append(L1,L2,L)$
$make\_regexpr([R],R) \leftarrow$
$make\_regexpr([R1,R2|Rs],(R1+T)) \leftarrow make\_regexpr([R2|Rs],T)$
$symbol(a) \leftarrow$
$symbol(b) \leftarrow$

---

where:

- $accepts\_empty(R)$ holds iff the regular expression $R$ accepts the empty word [ ];

- $next(R,S,R1)$ holds iff $S$ is a symbol in $\{a,b\}$ and for every word $W$ in $\{a,b\}^*$, $W$ is accepted by the regular expression $R$ iff there exists a word $Z$ accepted by the regular expression $R1$ such that $W = S;Z$. Thus, $R1$ is the *derivative* of $R$ w.r.t. $S$ [Brzozowski 64]. $R1$ is computed as the following three steps specify.

  i) First the predicate $transf(R,TR)$ transforms the regular expression $R$ into an equivalent regular expression $TR$ in the set $TReg$ defined as follows: $TReg ::= void \mid void^* \mid a \mid b \mid (a;Reg) \mid (b;Reg) \mid (TReg+TReg)$, where $void^*$ is the regular expression which accepts the empty word only. This transformation makes it very simple to compute the leftmost symbol of any word accepted by a given regular expression $R$.

ii) Then the predicate $deriv(TR, S, L)$ computes the list $L$ of all regular expressions $U$ such that $(S; U)$ is a *summand* of $TR$, where, given any regular expression $E = E_1 + \ldots + E_n$, for $n \geq 1$ and any possible parenthesization of $E_1 + \ldots + E_n$, a summand of $E$ is $E_i$ for any $i = 1, \ldots, n$. The predicate $deriv(TR, S, L)$ also includes the regular expression $void^*$ in $L$ iff $S$ is a summand of $TR$.

iii) Finally, the predicate $make\_regexpr(L, R1)$ transforms the list $L$ made out of the regular expressions $E_1, \ldots, E_{n-1}, E_n$ into the regular expression $R1$ which is $(E_1 + (\ldots + (E_{n-1} + E_n) \ldots))$.

For simplicity reasons, during partial evaluation, i) we also write terms without their outermost parentheses, ii) we assume the right associativity of ';' and '+', and iii) we assume that '$*$' binds more than ';' and ';' binds more than '+'. Thus, for instance, the term $(R1; (R2; R3))$ may also be written as $R1; R2; R3$.

From the above program *Accepts*, we would like to derive a residual program for testing whether or not a given word $W$ in $\{a, b\}^*$ is accepted by the regular expression $(a + b)^*; a; b; a$. This can be done by partially evaluating the program w.r.t. the goal $accepts(((a + b)^*; a; b; a), W)$.

We start off by introducing, via the definition rule, a new clause whose body is the goal w.r.t. which we want to perform the partial evaluation. Thus, in our case, we introduce the following clause:

9.1 $\quad new0(W) \leftarrow accepts(((a + b)^*; a; b; a), W)$

We then look for a set of clauses which define the predicate $new0$, and whose bodies do not contain any instantiated atom. Indeed, if there are instantiated atoms we may continue the partial evaluation process for taking full advantage of the information available. By repeatedly applying the unfolding rule and the clause deletion rule, we get from clause 9.1 the following clauses:

9.2 $\quad new0([a|W]) \leftarrow accepts((b; a + (a + b)^*; a; b; a), W)$

9.3 $\quad new0([b|W]) \leftarrow accepts(((a + b)^*; a; b; a), W)$

Clause 9.3 can be folded using clause 9.1, thereby getting:

9.4 $\quad new0([b|W]) \leftarrow new0(W)$

whose body does not contain any instantiated atom.

We now continue the derivation from clause 9.2. After a few unfolding steps and clause deletion steps we get:

9.5 $\quad new0([a, a|W]) \leftarrow accepts((b; a + (a+b)^*; a; b; a), W)$

9.6 $\quad new0([a, b|W]) \leftarrow accepts((a + (a+b)^*; a; b; a), W)$

Since the body of clause 9.5 is a variant of the body of clause 9.2 we may apply the loop absorption strategy. Thus, we introduce a new predicate, called *new*1, defined by the following clause whose body is equal to the body of clause 9.2:

9.7 $\quad new1(W) \leftarrow accepts((b; a + (a+b)^*; a; b; a), W)$

We then fold clause 9.2 using the definition clause 9.7, and we get the following clause:

9.8 $\quad new0([a|W]) \leftarrow new1(W)$

Now the clauses 9.4 and 9.8 defining the predicate *new*0 do not contain any instantiated atom.

Our partial evaluation process continues by considering clause 9.7 whose body contains an instantiated atom. By replaying the unfolding steps and the clause deletion steps which lead from clause 9.2 to clauses 9.5 and 9.6, and then performing a folding step using clause 9.7, we get the following two clauses for the predicate *new*1:

9.9 $\quad new1([a|W]) \leftarrow new1(W)$

9.10 $\quad new1([b|W]) \leftarrow accepts((a + (a+b)^*; a; b; a), W)$

Now in order to get clauses with no instantiated atoms in their bodies, we need to transform clause 9.10. Similarly to what we have done above starting from clause 9.2, we first apply some unfolding steps and clause deletion steps, and we then perform, when it is possible, some folding steps using definition clauses already introduced. When folding steps are not possible, we apply the loop absorption strategy and we introduce new definition clauses. By doing so we get the following final program:

---

9.11 $\quad accepts(((a+b)^*; a; b; a), W) \leftarrow new0(W)$

9.8 $\quad new0([a|W]) \leftarrow new1(W)$

9.4 $new0([b|W]) \leftarrow new0(W)$
9.9 $new1([a|W]) \leftarrow new1(W)$
9.12 $new1([b|W]) \leftarrow new2(W)$
9.13 $new2([a|W]) \leftarrow new3(W)$
9,14 $new2([b|W]) \leftarrow new0(W)$
9.15 $new3([\,]) \leftarrow$
9.16 $new3([a|W]) \leftarrow new1(W)$
9.17 $new3([b|W]) \leftarrow new2(W)$

---

where clause 9.11 has been added to the program for comparing the evaluations of the initial program and the evaluations of the residual program w.r.t. the same set of goals, each of which is of the form: '$accepts(((a+b)^*; a; b; a), w)$', for some given word $w$ in $\{a, b\}^*$.

From the correctness of the transformation rules it follows that, for every ground goal $G$ of the form '$accepts(((a+b)^*; a; b; a), w)$', we have that $G$ is true in the least Herbrand model of the initial program iff $G$ is true in the least Herbrand model of the final program.

Our final program is equal to the one derived by the Logimix partial evaluator [Mogensen-Bondorf 93], and it defines the deterministic finite automaton corresponding to the given regular expression, by considering the initial state 0 and the final state 3, and by taking any clause of the form $new\mathrm{h}([s|W]) \leftarrow new\mathrm{k}(W)$ as defining the transition from state h to state k with label $s$, for h and k in $\{0, 1, 2, 3\}$ and $s$ in $\{a, b\}$. □

Similarly to Partial Evaluation, *Staging Transformation* [Jørring-Scherlis 86] is a technique whereby the computation of a given function, say $f(x, y)$, is performed 'in stages', that is, $f(x, y)$ is computed as the value of the expression $f_2(f_1(x), y)$, where $f_1$ and $f_2$ are suitable auxiliary functions such that:

$f(x, y) = f_2(f_1(x), y)$.

The idea behind the Staging Transformation is the assumption that the argument $x$ is known 'before' $y$ (for instance, $x$ is known at compile-time and $y$ is known at run-time) and that, having computed the value of $f_1(x)$, we can then very efficiently evaluate $f_2$.

Staging Transformation can be viewed as a particular generalization from functions to functions by implicit definition. Indeed, one may apply twice that form of generalization strategy for obtaining from the equation $f(x, y) = expr$ the functions $f_1$ and $f_2$ such that $f(x, y) = f_2(f_1(x), y)$ holds.

### 4.3 Finite Differencing

*Finite Differencing* [Paige-Koenig 82] is a program transformation technique similar to the compiling technique called 'reduction of the operator strength'. We now illustrate the basic idea of finite differencing by way of an example.

**Example 10 (Finite Differencing via Generalization: Accumulation over Sequences in Monoids)** Let us consider the sequence $e_0, \ldots, e_{N+1}$ for some $N \geq 0$, and let us assume that we want to compute the sum of the elements of that sequence. That sum can be computed as the value of $f(N+1)$, defined by the following recursive equations:

$$\begin{array}{ll} f(0) = e_0 & \\ f(n+1) = e_{n+1} + f(n) & \text{for } 0 \leq n \leq N \end{array}$$

Let us assume that: i) there exists a function $\delta$ such that for any $n$, with $0 \leq n \leq N$, the element $e_n$ of the given sequence is equal to $\delta(e_{n+1}, n)$, and ii) the computation of the function $\delta$ is very efficient. Then, the application of the finite differencing technique generates the following program for the computation of the function $f$:

$$\begin{array}{ll} f(0) = e_0 & \\ f(n+1) = h(0, e_{n+1}, n) & \text{for } n \geq 0 \\ h(acc, z, 0) = acc + z + e_0 & \\ h(acc, z, n+1) = h(acc + z, \delta(z, n+1), n) & \text{for } n \geq 0, \end{array}$$

where the argument $acc$ plays the role of an accumulator.

More generally, let us consider the following program:

10.1 $f(0) = e(0)$

10.2 $f(n+1) = g(e(n+1), f(n))$ for $n \geq 0$,

where: i) $g(x, y)$ is an associative operation with neutral element $\eta$ (thus, $g$ determines a *monoid*), and ii) for any $n \geq 0$, $e(n)$ can be efficiently evaluated from the value of $e(n+1)$ by computing $\delta(e(n+1), n)$ for some given function $\delta$, while the direct evaluation of $e(n)$ from the value of $n$ is computationally expensive. Thus, for $n \geq 0$ we have that $f(n+1) = g(e(n+1), \ldots g(e(1), e(0)) \ldots)$. The value of $f(n+1)$ is said to be the *accumulation* of $g$ over the sequence '$e(0), e(1), \ldots, e(n+1)$'.

In these hypotheses finite differencing, as formalized by the schema transformation proposed in [Partsch 90], produces the following tail recursive program, which is a straightforward generalization of the program given above:

10.1 $f(0) = e(0)$
10.3 $f(n+1) = H(\eta, e(n+1), n)$ for $n \geq 0$
10.4 $H(acc, z, 0) = g(g(acc, z), e(0))$
10.5 $H(acc, z, n+1) = H(g(acc, z), \delta(z, n+1), n)$ for $n \geq 0$.

Thus, finite differencing basically works by performing the evaluation of an argument, $e(n+1)$ in our case, of a recursive function call by using the value, $e(n)$ in our case, of that argument in the previous function call. This fact may in some cases increase program efficiency.

Now we will show that the efficiency improvements due to finite differencing can be obtained via suitable generalization and folding steps. The corresponding transformation process can be done in two steps as follows.

i) We first look for a tail recursive program. By doing so, the computations to be performed from one recursive call to the next, are only the ones related to the evaluation of a substitution, say $\theta$, which computes the new recursive arguments from the old arguments.

ii) We then generalize suitable subexpressions, so that the substitution $\theta$ can be efficiently evaluated.

We start off by considering the initial program made out of the Equations 10.1 and 10.2 with the assumptions on $g$ and $\delta$ mentioned above. Then we unfold Equation 10.2 and we get for $n > 0$:

$$f(n+1) = g(e(n+1), f(n)) = \{\text{unfolding}\} = g(e(n+1), g(e(n), f(n-1))).$$

To derive a tail recursive program, the expression $g(e(n+1), g(e(n), f(n-1)))$ should be an instance of the r.h.s. of Equation 10.2 via the substitution $\{n = n-1\}$ which relates the recursive calls of the function $f$.

Thus, we first use the associativity property of $g$ for allowing the matching of the subexpression $f(n-1)$, and we get:

10.6 $f(n+1) = g(g(e(n+1), e(n)), f(n-1))$.

Then, in order to fold the r.h.s. of Equation 10.6 using Equation 10.2, we need to generalize the subexpression $g(e(n+1), e(n))$, which does *not* match $e(n+1)$, to a variable, say $y$, and we define the function:

10.7 $\quad G(y, n) = g(y, f(n)) \qquad\qquad$ for $n \geq 0$.

Thus, we get the following program:

---

10.1 $\quad f(0) = e(0)$

10.8 $\quad f(n+1) = \{\text{folding 10.2 using 10.7}\} = G(e(n+1), n) \quad$ for $n \geq 0$

10.9 $\quad G(y, 0) = g(y, e(0))$

10.10 $G(y, n+1) = \{\text{by 10.7}\} = g(y, f(n+1)) = \{\text{unfolding}\} =$

$= g(y, g(e(n+1), f(n))) = \{\text{associativity of } g\} =$

$= g(g(y, e(n+1)), f(n)) = \{\text{folding using 10.7}\} =$

$= G(g(y, e(n+1)), n) \qquad\qquad$ for $n \geq 0$.

---

This program is tail recursive, but unfortunately, it is *not* efficient because by Equation 10.10 the folding substitution $\{y = g(y, e(n+1)), \quad n+1 = n\}$, which computes the new arguments of $G$ from the old ones, requires the expensive evaluation of $e(n+1)$. We can avoid this drawback by unfolding Equation 10.10 and performing a generalization step as follows:

$$\begin{aligned} G(y, n+1) &= G(g(y, e(n+1)), n) = \{\text{unfolding using 10.10}\} = \\ &= G(g(g(y, e(n+1)), e(n)), n-1). \end{aligned}$$

This last expression $G(g(g(y, e(n+1)), e(n)), n-1)$ is an instance of the previous expression $G(g(y, e(n+1)), n)$ according to the substitution $\{y = g(y, e(n+1)), \;\; e(n+1) = e(n), \;\; n = n-1\}$, which can be efficiently computed if the values of $y$, $e(n+1)$, and $n$ are known. In particular, given $e(n+1)$ and $n$ we can compute $e(n)$ by using the function $\delta$.

The values of $y$, $e(n+1)$, and $n$ will be known during the recursive evaluation of $G$, if we promote $e(n+1)$ to be an argument of a new function, say

$H$, defined by generalization from functions to functions applied to the r.h.s. $G(g(y, e(n+1)), n)$ of Equation 10.10. Thus, $H$ should satisfy the following:

10.11 $H(y, e(n+1), n) \;=\; G(g(y, e(n+1)), n) \qquad$ for $n \geq 0$.

For the function $H$ we have:

10.12 $H(y, e(1), 0) = \{\text{by } 10.11\} = G(g(y, e(1)), 0) = \{\text{by } 10.7 \text{ and } 10.1\} =$
$= g(g(y, e(1)), e(0))$

10.13 $H(y, e(n+2), n+1) = \{\text{by } 10.11\} = G(g(y, e(n+2)), n+1) =$
$= \{\text{by } 10.10\} =$
$= G(g(g(y, e(n+2)), e(n+1)), n) =$
$= \{\text{folding using } 10.11\} =$
$= H(g(y, e(n+2)), e(n+1), n) =$
$= \{\text{by } e(n+1) = \delta(e(n+2), n+1)\} =$
$= H(g(y, e(n+2)), \delta(e(n+2), n+1), n)$
for $n \geq 0$.

The definition of the function $H$ is obtained by replacing in Equations 10.12 and 10.13 the second argument by a variable, say $z$. As a result, we get the following final program:

---

10.1 $\quad f(0) = e(0)$

10.14 $f(n+1) = G(e(n+1), n) = \{\text{neutrality of } \eta\} = G(g(\eta, e(n+1)), n) =$
$= H(\eta, e(n+1), n) \qquad$ for $n \geq 0$

10.12* $H(y, z, 0) = g(g(y, z), e(0))$

10.13* $H(y, z, n+1) = H(g(y, z), \delta(z, n+1), n) \qquad$ for $n \geq 0$.

---

This program is equal to the one made out of Equations 10.1, 10.3, 10.4, and 10.5 which is produced by the finite differencing technique. Notice that our derivation avoids the insights which are often required by the finite differencing technique, like, for instance, the understanding that the first argument of $H$ is an accumulator variable which holds the value of $g(\ldots g(g(\eta, e(n+1)), e(n)), \ldots, e(k))$ for some $k$, with $0 \leq k \leq n+1$, during the computation of $f(n+1)$.

Now, if we want to efficiently compute the following function $ss$ (see [Partsch 90], page 295) for computing the sum of squares:

$$
\begin{aligned}
&ss(0) = 0 && \\
&ss(n+1) = (n+1)^2 + ss(n) && \text{for } n \geq 0,
\end{aligned}
$$

we may use Equations 10.1, 10.14, 10.12*, and 10.13*, where we have that: i) $g(x, y) = x+y$, ii) $e(0) = \eta = 0$, iii) $e(n) = n^2$, and iv) $\delta(x, y) = x-2y-1$. This expression for the function $\delta$ is derived from the fact that, given $x = (n+1)^2$ and $y = n$, the value of $n^2$ can be efficiently computed by evaluating $x - 2y - 1$. Thus, we get the following program which avoids repeated squaring operations:

$$
\begin{aligned}
&ss(0) = 0 && \\
&ss(n+1) = H(0, (n+1)^2, n) && \text{for } n \geq 0 \\
&H(y, z, 0) = y + z && \\
&H(y, z, n+1) = H(y+z, z-2(n+1)-1, n) && \text{for } n \geq 0. \qquad \square
\end{aligned}
$$

## 5 Related Techniques for Program Development and Final Discussion

Due to space limitations we are not able to present here some other program transformation techniques which have been proposed in the literature (see, for instance, [Arsac-Kodratoff 82, Berry 76, Huet-Lang 78, Meertens 87, Pepper 84, Pepper 87, Wegbreit 76]).

In this section we would like to briefly indicate some techniques which have been developed in the fields of Theoretical Computer Science and Software Engineering and can also be used for assisting the programmer during the program transformation process.

First of all we want to mention the technique for proving program properties called *abstract interpretation* [Abramsky-Hankin 87, Cousot-Cousot 77, Debray 92]. By that technique one may acquire detailed knowledge about various program properties, like strictness of functions, types of arguments, data flow analysis, destructiveness of updatings, termination of evaluations, groundness of terms, functionality of predicates, etc., and those properties are then used for driving the program transformation process.

Notice, however, that in general only a 'partial knowlegde' about those properties is achievable, because of undecidability limitations. Notice also that the computation relative to the abstract interpretation analysis should always be terminating. Thus, one has to choose the domain of the abstraction with particular care so to get the maximum amount of information from the minimum amount of computation.

Particular relevance for the transformation process is the abstract interpretation called 'binding-time analysis', which tells us which expressions should be unfolded when part of the input data is known at compile-time [Launchbury 91, Mycroft 81, Nielson-Nielson 88, Jones et al. 93].

A general framework for the use of abstract interpretations when transforming logic programs is presented in [Boulanger-Bruynooghe 93].

As an example of use of abstract interpretations we now present the so-called *Matrix of Lengths method* for the derivation of on-line programs.

Given a program which produces a string as output, this method allows us to derive an equivalent program with *on-line behaviour*, that is, a program which produces the output string incrementally, one element at a time in a given order, and the production of one more output element takes *constant* time and *constant* space after the production of the preceding elements.

Actually, in our example we will allow for logarithmic time and logarithmic space w.r.t. the length $k$ of the output string, and we will say that a *pseudo on-line* behaviour has been achieved. For further details the interested reader may refer to [Pettorossi 87a].

**Example 11 (The Matrix of Lengths Method: Hilbert Curves)** Let us consider the free monoid $H^*$ of sequences of elements, also called moves, generated by the set $H = \{N, S, E, W\}$, where $N$, $S$, $E$, and $W$ denote the elementary moves towards North, South, East, and West, respectively. Let *skip* be the empty move and '::' denote the concatenation of moves in $H^*$. Let us consider the following program for computing the function *Hilbert*: $N \to H^*$, that is, the Hilbert Curve of any given order $n \geq 0$ [Wirth 76]:

11.1 $\mathit{Hilbert}(n) = A(n)$

11.2 $A(0) = B(0) = C(0) = D(0) = \mathit{skip}$

Figure 7: A graphical representation of the string $Hilbert(2) = A(2)$. The moves 1, 2, and 3 are the three elements of $D(1) = S :: W :: N$.

11.3 $A(n+1) = D(n) :: W :: A(n) :: S :: A(n) :: E :: B(n)$
11.4 $B(n+1) = C(n) :: N :: B(n) :: E :: B(n) :: S :: A(n)$
11.5 $C(n+1) = B(n) :: E :: C(n) :: N :: C(n) :: W :: D(n)$
11.6 $D(n+1) = A(n) :: S :: D(n) :: W :: D(n) :: N :: C(n)$,

where all equations hold for $n \geq 0$. In Fig. 7 we show the string $Hilbert(2)$. In that figure we have represented each move, say the $m$-th one, as a segment which is assumed to be oriented from the $(m-1)$-st move to the $(m+1)$-st move. For instance, we have that the 10-th move (from the left) of $Hilbert(2)$ is $S$ (not $N$).

Now we show through an example that for any $m > 0$ and $n \geq 0$ we can compute the $m$-th move of $Hilbert(n)$ in logarithmic time, and thus, we will derive a pseudo on-line algorithm.

We first derive a linear recursive program for $Hilbert(n)$ by applying the tupling strategy as follows. We tuple together the four function calls $A(n)$, $B(n)$, $C(n)$, and $D(n)$ because they share common subcomputations. Indeed, $A(n)$, $B(n)$, $C(n)$, and $D(n)$ all share $A(n-2)$.

We introduce the function: $Z(n) = \langle A(n), B(n), C(n), D(n)\rangle$, and for $n \geq 0$ we have:

11.7 $Hilbert(n) = \pi_1(Z(n))$
11.8 $Z(0) = \langle skip, skip, skip, skip\rangle$
11.9 $Z(n+1) = \langle d :: W :: a :: S :: a :: E :: b,\quad c :: N :: b :: E :: b :: S :: a,$
$\qquad b :: E :: c :: N :: c :: W :: d,\quad a :: S :: d :: W :: d :: N :: c\rangle$
$\qquad\qquad$ **where** $\langle a, b, c, d\rangle = Z(n)$ for $n \geq 0$.

Now we apply the Matrix of Length method as follows. As already mentioned, this method is based on a particular application of the *abstract interpretation* technique in which we consider the lengths of the strings in $H^*$, instead of the strings themselves.

Thus, with reference to the function $Z(n)$, we consider the lengths of its components and we introduce the function $L(n) = \langle L_A(n), L_B(n), L_C(n), L_D(n)\rangle$, where $L_A(n), \ldots, L_D(n)$ are the lengths of the strings $A(n), \ldots, D(n)$, respectively.

From Equations 11.7 we have that the length of *Hilbert*$(n)$ is $L_A(n)$ for any $n \geq 0$. By performing the abstract interpretation of the Equations 11.8 and 11.9 we get the following equations defining a linear recurrence relation:

11.10 $L(0) = \langle 0, 0, 0, 0\rangle$

11.11 $L(n+1) = \langle 2L_A + L_B + L_D + 3,\ L_A + 2L_B + L_C + 3,$
$L_B + 2L_C + L_D + 3,\ L_A + L_C + 2L_D + 3\rangle$
**where** $\langle L_A, L_B, L_C, L_D\rangle = L(n)$ for $n \geq 0$.

Then, in order to find a move of *Hilbert*$(n)$, we construct the Matrix of Lengths, say $M$, from column $j = 0$ up to column $j = n$. For $i = A, B, C, D$ and $j = 0, 1, \ldots, n$ the entry of $M$ at row $i$ and column $j$, denoted by $M(i, j)$, is $L_i(j)$. This matrix can be used to efficiently compute a move of *Hilbert*$(n)$ as we now show through an example.

Suppose that we want to compute the 10-th move of *Hilbert*$(2)$. In this case we have $n = 2$, and by applying Equations 11.10 and 11.11 we have the following matrix $M$:

| | $j=0$ | $j=1$ | $j=2$ |
|---|---|---|---|
| $i=A$ | 0 | 3 | 15 |
| $i=B$ | 0 | 3 | 15 |
| $i=C$ | 0 | 3 | 15 |
| $i=D$ | 0 | 3 | 15 |

The application of our method continues by looking at one column at a time, from column $j = n\,(= 2)$ down to column $j = 0$ until we find, as we will indicate, the desired move. For each column $j$ we compute a triplet of values, say $\langle i, j, k\rangle$, where $i$ is an element of $\{A, B, C, D\}$, denoting the component of $Z(j)$ whose $k$-th move we want to compute.

We start from the triplet $\langle A, 2, 10\rangle$, because we want to compute the 10-th move of *Hilbert*(2), that is, $A(2)$ (see Equation 11.1). Then, for any column $j > 0$ we compute from the triplet $\langle i, j, m\rangle$ for column $j$ the new triplet $\langle newi, j-1, newm\rangle$ for column $j-1$, as follows. We first consider the equation defining $i(j)$, that is, $A(2)$ in our case:

11.12 $A(2) = D(1) :: W :: A(1) :: S :: A(1) :: E :: B(1).$

We then apply to the r.h.s. of this equation the abstract interpretation which considers the length of the strings, and thus, in our case, we associate with $D(1) :: W :: A(1) :: S :: A(1) :: E :: B(1)$ the list $[3, 1, 3, 1, 3, 1, 3]$, because $L_A(1) = L_B(1) = L_D(1) = 3$. From this list we compute the values of *newi* and *newm* by considering that the length of the substring $D(1) :: W :: A(1) :: S$ is $8\ (= 3+1+3+1)$. Hence, the 10-th move of $A(2)$ is a move of $A(1)$, actually, it is the second move (from the left) of $A(1)$ (because $10 - 8 = 2$). Thus, we get the triplet relative to column $j = 1$. It is $\langle A, 1, 2\rangle$. $A(1)$ gives us the two components $A$ and 1, while the second move gives us the third component 2.

Finally, we compute the triplet relative to column $j = 0$ and we do so by considering the following equation:

11.13 $A(1) = D(0) :: W :: A(0) :: S :: A(0) :: E :: B(0)$

and the associated list of lengths: $[0, 1, 0, 1, 0, 1, 0]$. Hence, the second move of $A(1)$ is $S$, because the lengths of $D(0)$, $A(0)$, and $B(0)$ are all 0.

It remains to be proved that for the computations we have indicated above we only need at most a logarithmic amount of time and space with respect to $n$.

First of all, let us notice that if the abstract interpretation of the given equations gives rise to linear recurrence relations with constant coefficients, we need only a fixed number of columns of the Matrix of Lengths $M$ for computing one more column. We do not need to keep the entire matrix in memory, and we only need a constant number of memory cells (four, in our case, to store one column).

We also need a logarithmic amount of time to compute the elements of a new column of $M$ from the elements of the stored columns, because linear recurrence relations with constant coefficients can be evaluated in logarithmic time (see Example 5). Moreover, if the linear recurrence relations for $L(n)$ have solutions which grow exponentially with the parameter $n$, then when looking for the $m$-th move of the output string, the number of columns in the matrix $M$ is proportional to $\log(m)$, and thus, we only need $O(\log(m))$ time to compute the $m$-th move. □

One may apply the Matrix of Lengths method also to the Towers of Hanoi problem presented in Section 2. In that case the computation of the $m$-th move can be performed in constant time (if we assume that the binary digits of the given number $m$ can be computed in constant time) [Pettorossi 87a].

When transforming programs by the 'rules + strategies' approach we cannot change the programming language. This limitation may be overcome by the so-called *program annotation* technique, which has been proposed in [Schwarz 82]. In the annotation approach the improvement of performances is obtained by deriving from the given program written in the language $L$, the corresponding 'annotated program' written in the language $L_a$, for which we can provide an efficient evaluator. Annotations often refer to the use of memo-functions, the call-by-value or call-by-need mode of evaluation, etc., and they can be determined by automatic procedures.

Program transformation techniques are also related to *program synthesis* techniques, in the sense that the initial program can be considered as the initial specification, and the final program can be considered as the implementation of that specification. We may refer the reader to [Manna-Waldinger 91, Deville-Lau 94]. However, it is usually understood that the techniques for program synthesis are much more powerful than the simple manipulations which are allowed within the usual transformational approach. If we allow the use of very sophisticated *laws* or *properties* during program transformation (much more sophisticated than the associativity of *append* used in Example 6) then it is very difficult to separate the areas of program synthesis and program transformation.

A particular issue for which program transformation is related to program synthesis is the one concerning the *invention of data structures.* Indeed, when deriving programs following the 'rule + strategies' approach, often the data structures are *dynamically* invented while the transformation process progresses, and no fixed choice is made at the beginning of the transformation process [Wirth 76]. For instance, in Example 6 the introduction of the non-homogeneous array $R(l, y)$ consisting in one function and one list, is determined by the need-for-folding.

There are also techniques by which program synthesis may be viewed as part of program transformation. Among those techniques we would like to mention the so called *completion techniques* [Dershowitz 85]. In that paper it is shown how a new program can be derived from an old one by successive applications of semantics-preserving rewriting steps each of which can be viewed as a program trasformation step.

Recent developments have indicated a new interesting way of considering the process of program development (or transformation) from specifications. This approach is based on *Constructive Type Theories* (see, for instance, [Bates-Constable 85, Burstall et al. 91, Coquand-Huet 88, Galmiche 91, Sintzoff et al. 89]) where a program can be derived from the constructive proof of the existence of an element in a suitable domain type.

The transformational methods for developing programs are also closely related to methods for *program construction* (see, for instance, [Deville 90, Sterling-Kirschenbaum 93]), where complex programs are developed by enhancing and composing together simpler programs. However, the basic ideas and objectives of program construction are different from those of program transformation. In particular, the starting point of the methods for program construction is not a complete program, but a possibly incomplete and not fully formalized specification. Moreover, the main objective of program construction is the improvement of the efficiency in software production, rather than the improvement of the efficiency of derived programs.

Other techniques related to the transformation strategies we have described, such as *program slicing*, *program integration*, and *program enhance-*

*ment*, have been described in [Lakhotia-Sterling 88, Reps 90].

The presentation of rules and strategies for program derivation and transformation could have been done using *other formalisms* or frameworks, like, for instance, the ones proposed in [Backus 78, Bauer et al. 85, Bird-Meertens 87]. In those formalisms strategies take the form of theorems and equations which are used to rewrite programs. We could have also considered the case of Pascal-like programs [Illsley 88, Smith 85], where strategies resemble more closely the ones often used in Theorem Proving, such as divide-and-conquer, global search, local search, and pruning.

We have considered the transformation rules and strategies in the case of functional languages and then, separately, in the case of logic languages. In the literature, however, there are approaches for the integration of the two language paradigms, as for instance, [Bosco et al. 87, Darlington-Guo 79, van Emden-Yukawa 87, Fribourg 85, Goguen-Meseguer 86, Lincoln et al. 94, Reddy 87, Subrahmanyam-You 86], but no sets of rules have been proposed for transforming programs written using those integrated formalisms.

Some work has been done in the direction of translating logic programs into functional programs [Gardarin et al. 89, Reddy 84] and vice versa [Togashi-Noguchi 87]. These translations may allow improvements of efficiency, because one can exploit the functionality of some predicates and one can also use very efficient techniques for the evaluation of recursive equations. The translation of functional programs into logic programs has the advantage that one can 'invert' the computation of functions by deriving the inputs corresponding to given outputs, via the relational calculus provided by the logic programming setting.

Concerning the limitation of the transformation approach, the reader should notice that, in general, transformation strategies *cannot* be complete in the sense that their ability of deriving new programs from old programs is limited by the fact that the semantics equivalence of programs is undecidable. Thus, there is no algorithm which for any given initial program explores in finite time all equivalent programs which are more efficient than the initial program.

Moreover, the transformation methodology based on the kind of rules we have presented, such as unfolding, folding, etc., has some intrinsic complexity limitations in the sense that there exists a bound on the efficiency improvements that can be obtained via those rules. This has been shown in [Zhu 94].

A final issue is about the *mechanization* of the transformation rules and strategies for program development. The description of some systems which have been already implemented and are under development can be found in [Alexandre et al. 92, Aerts-Van Besien 91, Bauer et al. 87, Boyle 84, Cai-Paige 90, Feather 82, Feather 87, Krieg-Brückner 89, Partsch-Steinbrüggen 83, Smith 93, Wile 82].

## Acknowledgements

We would like to thank our colleagues of the IFIP Working Group 2.1 on *Algorithmic Languages and Calculi.* Their encouragement throughout the years of our involvement in the area of Program Transformation and Derivation is greatly appreciated. Particular thanks go to Prof. B. Möller and Prof. H. Partsch for their suggestions and comments, and to Prof. A. Haeberer for his invitation to the 'Formal Program Development' Seminar in Rio de Janeiro, Brazil, in 1992. For that occasion we wrote a preliminary version of this survey. We thank T. Mogensen for discussions concerning Example 9.

This work has been partially supported by the 'Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo' of the CNR, Italy, under grant n. 89.00026.69, MURST 40%, and Esprit Compulog II.

## References

[Abramsky-Hankin 87] Abramsky, S., Hankin, C. (eds.): Abstract Interpretation of Declarative Languages. Ellis Horwood (1987)

[Aerts-Van Besien 91] Aerts, K., Van Besien, D.: Autolap: A System for Transforming Logic Programs. Department of Electronics, Report University of Rome Tor Vergata, Rome, Italy (1991)

[Alexandre et al. 92] Alexandre, F., Bsaïes, K., Finance, J. P., Quéré, A.: Spes: A System for Logic Program Transformation. In Proceedings of the International Conference on Logic Programming and Automated Reasoning, LPAR '92, Lecture Notes in Computer Science **624**, Springer Verlag (1992) 445–447

[Aravindan-Dung 93] Aravindan, C., Dung, P. M.: On the Correctness of Unfold/Fold Transformation of Normal and Extended Logic Programs. Technical Report, Computer Science Division, Asian Institute of Technology, Bangkok, Thailand (1993)

[Arsac-Kodratoff 82] Arsac, J., Kodratoff, Y.: Some Techniques for Recursion Removal From Recursive Functions. ACM Toplas **4** (2) (1982) 295–322

[Aubin 79] Aubin, R.: Mechanizing Structural Induction: Part I and II. Theoretical Computer Science **9** (3) (1979) 329–362

[Backus 78] Backus, J.: Can Programming Be Liberated From The Von Neumann Style? Communications of the ACM **21** (8) (1978) 613–641

[Bates-Constable 85] Bates, J. L., Constable, R. L.: Proofs as Programs. ACM Toplas **7** (1) (1985) 113–136

[Bauer et al. 85] Bauer, F. L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., Wössner, H.: The CIP Language Group. The Munich Project CIP. Vol. I. The Wide Spectrum Language CIP-L. Lecture Notes in Computer Science **183**, Springer Verlag (1985)

[Bauer et al. 87] Bauer, F. L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O., Pepper, P.: The CIP System Group. The Munich Project

CIP. Vol. II. The Program Transformation System CIP-S. Lecture Notes in Computer Science **292**, Springer Verlag (1987)

[Berry 76] Berry, G.: Bottom-Up Computation of Recursive Programs. RAIRO Informatique Théorique **10** (3) (1976) 47–82

[Bird 80] Bird, R. S.: Tabulation Techniques for Recursive Programs. ACM Computing Surveys **12** (4) (1980) 403–418

[Bird 84a] Bird, R. S.: Using Circular Programs to Eliminate Multiple Traversal of Data. Acta Informatica **21** (1984) 239–250.

[Bird 84b] Bird, R. S.: The Promotion and Accumulation Strategies in Transformational Programming. ACM Toplas **6** (4) (1984) 487–504

[Bird-Meertens 87] Bird, R. S., Meertens, L. G. L. T.: Two Exercises Found in a Book on Algorithmics. TC2 WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz (Germany) North Holland (1987) 451–457

[Bjørner et al. 88] Bjørner, D., Ershov, A. P., Jones, N. D. (eds.): Partial Evaluation and Mixed Computation. IFIP TC2 Workshop on Partial and Mixed Computation, Gammel Avernæs (Denmark) North Holland (1988)

[Boiten 90] Boiten, E. A.: Views of Formal Program Development. Ph. D. Thesis, Catholic University of Nijmegen, Nijmegen, The Netherlands (1990)

[Bosco et al. 87] Bosco, P. G., Giovanetti, E., Levi, G., Moiso, C., Palamidessi, C.: A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions. In Proc. 4th Symposium on Logic Programming, San Francisco, U.S.A. IEEE Computer Society Press (1987) 1–27

[Bossi-Cocco 93] Bossi, A., Cocco, N.: Basic Transformation Operations for Logic Programs which Preserve Computed Answer Substitutions. Journal of Logic Programming, Special Issue on Partial Deduction **16** (1993) 47–87

[Bossi-Cocco-Etalle 92] Bossi, A., Cocco, N., Etalle, S.: Transforming Normal Programs by Replacement. In: Pettorossi, A. (ed.), Proc. 3rd International Workshop on Meta-Programming in Logic (Meta '92), Uppsala, Sweden, Lecture Notes in Computer Science **649**, Springer Verlag (1992) 265–279

[Boulanger-Bruynooghe 93] Boulanger, D., Bruynooghe, M.: Deriving Unfold/Fold Transformations of Logic Programs Using Extended OLDT-based Abstract Interpretation. Journal of Symbolic Computation **15** (1993) 495–521

[Boyle 84] Boyle, J. M.: Lisp to Fortran – Program Transformation Applied. In: Pepper, P. (ed.): Proc. Workshop on Program Transformation and Programming Environments. Nato ASI Series F, Computer and Systems Sciences **8**, Springer Verlag (1984) 291–298

[Boyer-Moore 75] Boyer, R. S., Moore, J. S.: Proving Theorems About LISP Functions. JACM **22** (1) (1975) 129–144

[Brough-Hogger91] Brough, D. R, Hogger, C. J.: Grammar-Related Transformations of Logic Programs. New Generation Computing **9** (1) (1991) 115–134

[Bruynooghe et al. 89] Bruynooghe, M., De Schreye, D., Krekels, B.: Compiling Control. Journal of Logic Programming **6** (1 & 2) (1989) 135–162

[Brzozowski 64] Brzozowski, J. A.: Derivatives of Regular Expressions. JACM **11** (4) (1964) 481–494

[Burstall et al. 91] Burstall, R. M. et al.: The Lego Project. Computer Science Department, Edinburgh University, Edinburgh (Scotland) (1991)

[Burstall-Darlington 75] Burstall, R. M., Darlington, J.: Some Transformations for Developing Recursive Programs. Proceedings of the International Conference on Reliable Software, Los Angeles USA (1975) 465–472

[Burstall-Darlington 77] Burstall, R. M., Darlington, J.: A Transformation System for Developing Recursive Programs. JACM, **24** (1) (1977) 44–67

[Cai-Paige 90] Cai, J., Paige, R.: The RAPTS Transformation System. Computer Science Department, New York University, New York USA (1990)

[Chatelin 76] Chatelin, P.: Program Manipulation: to Duplicate is not to Complicate. Report CNRS Laboratoire d'Informatique. Université de Grenoble (France) (1976)

[Chin 90] Chin, W.-N.: Automatic Methods for Program Transformation. Ph.D. Thesis, Imperial College of Science, Technology and Medicine, University of London, London, U.K. (1990)

[Chin 92] Chin, W.-N.: Safe Fusion of Functional Expressions. Proceedings of ACM SIGPLAN Symposium on Lisp and Functional Programming, San Francisco, Calif., U.S.A., ACM Press (1992) 11–20

[Chin 93] Chin, W.-N.: Towards an Automated Tupling Strategy. Proc. ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark, ACM Press (1993) 119–132

[Coquand-Huet 88] Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation **76** (1988) 95–120

[Courcelle 90] Courcelle, B.: Recursive Applicative Program Schemes. In: van Leuveen, J. (ed.): Handbook of Theoretical Computer Science. Vol. B, Elsevier (1990) 459–492

[Cousot-Cousot 77] Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. 4th POPL (1977) 238–252

[Darlington 72] Darlington, J.: A Semantic Approach to Automatic Program Improvement. Ph.D. Thesis, Department of Machine Intelligence, Edinburgh University, Edinburgh, U.K. (1972)

[Darlington 81] Darlington, J.: An Experimental Program Transformation System. Artificial Intelligence **16** (1981) 1–46

[Darlington-Guo 79] Darlington, J., Guo, Y.: The Unification of Functional and Logic Languages, towards Constraint Functional Programming. Technical Report, Imperial College, London, U.K. (1979)

[Debray 88] Debray, S. K.: Unfold/Fold Transformations and Loop Optimization of Logic Programs. Proceedings Sigplan 88, Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA, Sigplan Notices **23** (7) (1988) 297–307

[Debray 92] Debray, S. K. (ed.): Special Issue of the Journal of Logic Programming on Abstract Interpretation. Vol. **12** (2 & 3), Elsevier (1992)

[Debray-Warren 88] Debray, S. K., Warren, D. S.: Automatic Mode Inference for Logic Programs. Journal of Logic Programming **5** (1988) 207–229

[Dershowitz 85] Dershowitz, N.: Synthesis by Completion. Proc. 9th International Joint Conference on Artificial Intelligence **1** (1985) 208–214

[Deville 90] Deville, Y.: Logic Programming: Systematic Program Development. Addison-Wesley (1990)

[Deville-Burnay 89] Deville, Y., Burnay, J.: Generalization and Program Schemata. Proceedings NACLP '89, MIT Press (1989) 409–425

[Deville-Lau 94] Deville, Y., Lau, K.-K.: Logic Program Synthesis. Journal of Logic Programming, **19** & **20** (1994) 321–350

[van Emden-Yukawa 87] van Emden, M., Yukawa, K.: Logic Programming with Equations. Journal of Logic Programming, **4** (1987) 265–288.

[Ershov 82] Ershov, A. P.: Mixed Computation: Potential Applications and Problems for Study. Theoretical Computer Science **18** (1982) 41–67

[Feather 82] Feather, M. S.: A System for Assisting Program Transformation. ACM Toplas **4** (1) (1982) 1–20

[Feather 87] Feather, M. S.: A Survey and Classification of Some Program Transformation Techniques. Proc. TC2 IFIP Working Conference on Program Specification and Transformation, Bad Tölz (Germany) North Holland (1987) 165–195

[Fribourg 85] Fribourg, L.: SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. Proceedings of the IEEE International Symposium on Logic Programming, Boston, IEEE Computer Society Press (1985) 172–184

[Fuchs-Fromherz 92] Fuchs, N. E., Fromherz, M. P. J.: Schema-based Transformations of Logic Programs. In: Clement, T., Lau, K.-K. (eds.): Logic Program Synthesis and Transformation, Proceedings LOPSTR '91, Manchester, U.K., Springer-Verlag (1992) 111–125

[Futamura71] Futamura, Y.: Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. Systems, Computers, Controls **2** (5) (1971) 45–50

[Gallagher 93] Gallagher, J. P.: Tutorial on Specialization of Logic Programs. In: Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark, ACM Press (1993) 88–98

[Galmiche 91] Galmiche, D.: Proofs and Program Development in AF2. Notes of the IFIP W.G. 2.1 Meeting, Louvain-la-Neuve (Belgium) (1991)

[Gardarin et al. 89] Gardarin, G., Guessarian, I., de Maindreville, C.: Translation of Logic Programs into Functional Fixpoint Equations. Theoretical Computer Science **63** (1989) 253–274

[Gardner-Shepherdson 91] Gardner P. A., Shepherdson J. C.: Unfold/Fold Transformations of Logic Programs. In: Lassez, J.-L., Plotkin, G. (eds.): Computational Logic, Essays in Honor of Alan Robinson. MIT Press (1991) 565–583

[Goguen-Meseguer 86] Goguen J., Meseguer J.: Eqlog: Equality, Types, and Generic Modules for Logic Programming. In: De Groot, D., Lindstrom, G. (eds.): Logic Programming: Functions, Relations, and Equations. Prentice-Hall (1986) 295–363. An earlier version appears in Journal of Logic Programming, **1** (2) (1984) 179–210.

[Hogger 81] Hogger C. J.: Derivation of Logic Programs. JACM **28** (2) (1981) 372–392

[Huet-Lang 78] Huet, G., Lang, B.: Proving and Applying Program Transformation Expressed with Second-Order Patterns. Acta Informatica **11** (1978) 31–55

[Hughes 82] Hughes, R. J. M.: Super-combinators a New Implementation Method for Applicative Languages. Proc. 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, PA, USA (1982) 1–10

[Illsley 88] Illsley, M.: Transforming Imperative Programs. Ph.D. Thesis, Computer Science Department, Edinburgh University, Edinburgh (Scotland) (1988)

[Johnsson 85] Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. Functional Programming Languages and Computer Architecture, Nancy (France) Lecture Notes in Computer Science **201**, Springer Verlag (1985) 190–203

[Jones 87] Jones, N. D.: Automatic Program Specialization: A Re-Examination From Basic Principles. In: Bjørner, D., Ershov, A. P. (eds.): Proc. IFIP TC2 Working Conference on Partial and Mixed Computation. Ebberup (Denmark) (1987) 225–282

[Jones et al. 93] Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)

[Jones-Søndergaard 87] Jones, N., Søndergaard, H.: A Semantics-Based Framework for the Abstract Interpretation of Prolog. In: Abramsky, S., Hankin, C. (eds.): Abstract Interpretation of Declarative Languages. Ellis Horwood (1987) 123–142

[Jørring-Scherlis 86] Jørring, U., Scherlis, W. L.: Compilers and Staging Transformations. Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburgh Beach, Florida, USA (1986) 86–96

[Kanamori-Horiuchi 87] Kanamori, T., Horiuchi, K.: Construction of Logic Programs Based on Generalized Unfold/Fold Rules. Proceedings of the Fourth International Conference on Logic Programming, The MIT Press (1987) 744–768

[Kanamori-Maeji 86] Kanamori, T., Maeji, M.: Derivation of Logic Programs from Implicit Definition. Technical Report TR-178, ICOT, Tokyo (Japan) (1986)

[Kawamura-Kanamori 88] Kawamura, T., Kanamori, T.: Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo (Japan) (1988) 413–422

[Kott 78] Kott, L: About Transformation System: A Theoretical Study. 3ème Colloque International sur la Programmation, Dunod, Paris (France) (1978) 232–247

[Kott 82] Kott, L: The McCarthy's Induction Principle: 'Oldy' but 'Goody'. Calcolo **19** (1) (1982) 59–69

[Krieg-Brückner 89] Krieg-Brückner, B. (ed.): COMPASS, A COMPrehensive Algebraic Approach to System Specification and Development. ESPRIT Basic Research Working Group 3264, Objectives, State of the Art, References. University of Bremen, Germany, Bericht no. 6/89 (1989)

[Lakhotia-Sterling 88] Lakhotia, A., Sterling, L.: Composing Recursive Logic Programs with Clausal Join. New Generation Computing **6** (2 & 3) (1988) 211–226

[Launchbury 91] Launchbury, J.: Projection Factorisations in Partial Evaluation. Distinguished Dissertations in Computer Science, Cambridge University Press (1991)

[Lincoln et al. 94] Lincoln, P., Marti-Oliet, N., Meseguer, J.: Specification, Transformation, and Programming of Concurrent Systems in Rewriting Logic. In: Blelloch, G., Chandy, K.M., Jagannathan, S. (eds.): Proc. of the DIMACS Conference on Specification of Parallel Algorithms, Princeton, New Jersey, American Mathematical Society (1994) 309–339.

[Lloyd 87] Lloyd, J. W.: Foundations of Logic Programming. Springer Verlag, 2nd edition (1987)

[Lloyd-Shepherdson 91] Lloyd, J. W., Shepherdson, J. C.: Partial Evaluation in Logic Programming. Journal of Logic Programming **11** (1991) 217–242

[Maher 93] Maher, M. J.: A Transformation System for Deductive Database Modules with Perfect Model Semantics. Theoretical Computer Science **110** (1993) 377–403

[Manna 74] Manna, Z.: Mathematical Theory of Computation. MacGraw-Hill (1974)

[Manna-Waldinger 91] Manna, Z., Waldinger, R.: Fundamentals of Deductive Program Synthesis. In: Bauer, F. L. (ed.): Logic, Algebra, and Computation. Springer-Verlag (1991) 41–107

[Marakakis-Gallagher 94] Marakakis E., Gallagher, J. P.: SchemaBased Tod-Down Design of Logic Programs Using Abstract Data Types. Proceedings of LOPSTR '94, Pisa, Italy, Lecture Notes in Computer Science 883, Springer Verlag (1994) 138–153

[Meertens 87] Meertens, L. G. L. T. (ed.): Program Specification and Transformation. Proc. IFIP TC2 W.G. 2.1 Working Conference on Program Specification and Transformation, Bad Tölz (Germany) North Holland (1987)

[Mogensen-Bondorf 93] Mogensen, T., Bondorf, A.: Logimix: A Self-Applicable Partial Evaluator for Prolog. In: Lau, K.-K., Clement, T. (eds.): Logic Program Synthesis and Transformation, Proceedings LOP-STR '92, Manchester, U.K., Springer-Verlag (1993) 214–227

[Möller 91] Möller, B.: Relations as a Program Development Language. Proc. IFIP TC2 Working Conference on Constructing Programs from Specifications, Pacific Grove, California USA, North Holland (1991)

[Möller 93] Möller, B.: Derivation of Graph and Pointer Algorithms. In: Möller, B., Partsch, H., Schuman, S. (eds.): Formal Program Development. Lecture Notes in Computer Science **755**, Springer Verlag (1993) 123–160

[Möller et al. 93] Möller, B., Partsch, H., Schuman, S. (eds.): Formal Program Development. Lecture Notes in Computer Science **755**, Springer Verlag (1993)

[Moreno-Rodriguez 92] Moreno-Navarro, J. J., Rodriguez-Artalejo, M.: Logic Programming with Functions and Predicates: The Language Babel. Journal of Logic Programming **12**, Elsevier (1992) 191–223

[Mycroft 81] Mycroft, A.: Abstract Interpretation and Optimizing Transformations for Applicative Programs. Ph.D. Thesis CTS 15–81. University of Edinburgh, Scotland (1981)

[Nielson-Nielson 88] Nielson, F., Nielson, H. R.: Automatic Binding Time Analysis for a Typed $\lambda$-calculus. Science of Computer Programming **10** (1988) 139–176

[Paige-Koenig 82] Paige, R., Koenig, S.: Finite Differencing of Computable Expressions. ACM Toplas **4** (3) (1982) 402–454

[Paige et al. 93] Paige, R., Reif, J., Wachter, R. (eds.): Parallel Algorithm Derivation and Program Transformation. Proc. Workshop at Courant Institute of Mathematical Sciences, New York USA, Kluwer Academic Publishers (1993)

[Partsch 90] Partsch, H. A.: Specification and Transformation of Programs. Springer Verlag, New York (1990)

[Partsch-Steinbrüggen 83] Partsch, H. A., Steinbrüggen, R.: Program Transformation Systems. ACM Computing Surveys **15** (1983) 199–236

[Paterson-Hewitt 70] Paterson, M. S., Hewitt, C. E.: Comparative Schematology. Conference on Concurrent Systems and Parallel Computation Project MAC, Woods Hole, Mass. USA (1970) 119–127

[Pepper 84] Pepper, P. (ed.): Program Transformation and Programming Environments. Workshop Report, Nato ASI Series F **8**, Springer Verlag (1984)

[Pepper 87] Pepper, P. (ed.): A Simple Calculus for Program Transformation (Inclusive of Induction). Science of Computer Programming **9** (1987) 221–262

[Pettorossi 84] Pettorossi, A.: Methodologies for Transformations and Memoing in Applicative Languages. Ph. D. Thesis, Edinburgh University, Edinburgh, Scotland (1984)

[Pettorossi 87a] Pettorossi, A.: Derivation of Efficient Programs for Computing Sequences of Actions. Theoretical Computer Science **53** (1987) 151–167

[Pettorossi 87b] Pettorossi, A.: Derivation of Programs which Traverse Their Input Data Only Once. In: Cioni, G., Salwicki, A. (eds.): Proceedings of the School on Programming Methodologies, Roma, Academic Press (1987) 165–184

[Pettorossi-Burstall 82] Pettorossi, A., Burstall, R.M.: Deriving Very Efficient Algorithms for Evaluating Linear Recurrence Relations Using the Program Transformation Technique. Acta Informatica **18** Springer Verlag (1982) 181–206

[Pettorossi-Proietti 87] Pettorossi, A., Proietti, M.: Importing and Exporting Infomation in Program Development. Proc. IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernaes (Denmark) North Holland (1987) 405–425

[Pettorossi-Proietti 94] Pettorossi, A., Proietti, M.: Transformation of Logic Programs: Foundations and Techniques. Journal of Logic Programming **19**, **20** (1994) 261–320

[Pettorossi-Skowron 82] Pettorossi, A., Skowron, A.: Communicating Agents for Applicative Concurrent Programming. Proc. Intern. Symp. on Programming, Turin, Italy, Lecture Notes in Computer Science **137**, Springer Verlag (1982) 305–322

[Pettorossi-Skowron 87] Pettorossi, A., Skowron, A.: Higher Order Generalization in Program Derivation. Proc. Tapsoft '87, Pisa, Italy, Lecture Notes in Computer Science **250**, Springer Verlag (1987) 182–196

[Proietti-Pettorossi 90] Proietti, M., Pettorossi, A.: Synthesis of Eureka Predicates for Developing Logic Programs. Proc. ESOP '90, Copenhagen, Lecture Notes in Computer Science **432**, Springer Verlag (1990) 306–325

[Proietti-Pettorossi 91a] Proietti, M., Pettorossi, A.: Semantics Preserving Transformation Rules for Prolog. Proc. ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, New Haven USA, Sigplan Notices **26** 9 (1991) 274–284

[Proietti-Pettorossi 91b] Proietti, M., Pettorossi, A.: Unfolding-Definition-Folding, in this order, for Avoiding Unnecessary Variables in Logic Programs. In: Proceedings Third International Symposium on Programming Language Implementation and Logic Programming. PLILP '91, Passau, Germany. Lecture Notes in Computer Science **528**, Springer Verlag (1991) 347–358

[Proietti-Pettorossi 94] Proietti, M., Pettorossi, A.: Synthesis of Programs from Unfold/Fold Proofs. Proceedings LOPSTR '93, Louvain-la-Neuve, Belgium, 1993, Workshops in Computing Series, Springer Verlag (1994) 141–158

[Reddy 84] Reddy, U. S.: Transformation of Logic Programs into Functional Programs. Proceedings of the IEEE International Symposium on Logic Programming, IEEE Computer Society Press (1984) 187–197

[Reddy 87] Reddy, U. S.: Functional Logic Languages, Part I. In: Fasel, J. H., Keller, R. M. (eds.): Workshop Proceedings on Graph Rewriting. Lecture Notes in Computer Science **279**, Springer Verlag (1987) 401–425

[Reps 90] Reps, T.: Algebraic Properties of Program Integration. Proc. ESOP '90, Lecture Notes in Computer Science **432**, Springer Verlag (1990) 326–340

[Sands 94] Sands, D.: Total Correctness and Improvement in the Transformation of Functional Programs. Technical Report, DIKU, University of Copenhagen, Denmark (1994)

[Sato 92] Sato, T.: An Equivalence Preserving First Order Unfold/Fold Transformation System. Theoretical Computer Science **105** (1992) 57–84

[Scherlis 81] Scherlis, W. L.: Program Improvement by Internal Specialization. Proc. 8th ACM Symposium on Principles of Programming Languages, Williamsburgh, Va., USA (1981) 41–49

[Schwarz 82] Schwarz, J.: Using Annotations to Make Recursive Equations Behave. IEEE Transactions on Software Engineering SE **8** (1) (1982) 21–33

[Seki 91] Seki, H.: Unfold/Fold Transformation of Stratified Programs. Theoretical Computer Science **86** (1991) 107–139

[Seki 93] Seki, H.: Unfold/Fold Transformation of General Logic Programs for the Well-founded Semantics. Journal of Logic Programming, Special Issue on Partial Deduction **16** (1 & 2) (1993) 5–23

[Seki-Furukawa87] Seki, H., Furukawa, K.: Notes On Transformation Techniques for Generate and Test Logic Programs. Proceedings of the International Symposium on Logic Programming, San Francisco, USA, IEEE Press (1987) 215–223

[Sintzoff et al. 89] Sintzoff, M., Weber, M., de Groote, Ph., Cazin, J.: Definition 1.1 of the Generic Development Language Deva. Technical Report, Unité d'Informatique, Université Catholique de Louvain, Louvain-La-Neuve, Belgium (1989)

[Smith 85] Smith, D. R.: The Design of Divide and Conquer Algorithms. Science of Computer Programming **5** (1985) 37–58

[Smith 93] Smith, D. R.: Automatic Derivation of Algorithms. In: Möller, B., Partsch, H., Schuman, S. (eds.): Formal Program Development. Lecture Notes in Computer Science **755**, Springer Verlag (1993) 324–354

[Sterling-Kirschenbaum 93] Sterling, L., Kirschenbaum, M.: Applying Techniques to Skeletons. In: Jacquet, J.-M. (ed.): Constructing Logic Programs. Chapter 6, Wiley (1993) 127–140

[Subrahmanyam-You 86] Subrahmanyam, P. A., You, J. H.: FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming. In: De Groot, D., Lindstrom, G. (eds.): Logic Programming: Functions, Relations, and Equations. Prentice-Hall (1986) 157–198

[Takeichi 87] Takeichi, M.: Partial Parametrization Eliminates Multiple Traversals of Data Structures. Acta Informatica **24** (1987) 57–77

[Tamaki-Sato 84] Tamaki, H., Sato, T.: Unfold/Fold Transformation of Logic Programs. Proc. 2nd International Conference on Logic Programming, Uppsala (Sweden) (1984) 127–138

[Tarau-Boyer 90] P. Tarau, P., Boyer, M.: Elementary Logic Programs. In: Deransart, P., Małuszynski, J. (eds.): Proceedings PLILP '90, Springer Verlag, (1994) 159–173

[Togashi-Noguchi 87] Togashi, A., Noguchi, S.: A Program Transformation from Equational Programs into Logic Programs. Journal of Logic Programming **4** (1987) 85–103

[Turner 79] Turner, D. A: A New Implementation Technique for Applicative Languages. Software Practice and Experience **9** (1979) 31–49

[Wadler 85] Wadler, P. L.: Listlessness is Better than Laziness. Ph.D. Thesis, Computer Science Department, CMU-CS-85-171, Carnegie Mellon University, Pittsburgh, PA, USA (1985)

[Wand 80] Wand, M.: Continuation-based Program Transformation Strategies. JACM **27** (1) (1980) 164–180

[Wegbreit 76] Wegbreit, B.: Goal-directed Program Transformation. IEEE Transactions on Software Engineering SE **2** (1976) 69–79

[Wile 82] Wile, D.: POPART: Producer of Parser and Related Tools. Technical Report RR-82-21, ISI, 4676 Admiralty Way, Marina del Rey, California, USA (1982)

[Wirth 76] Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall, Inc. (1976)

[Zhu 94] Zhu, H.: How Powerful are Folding / Unfolding Transformations? Journal of Functional Programming **4** (1) (1994) 89–112