METHODOLOGIES FOR TRANSFORMATIONS AND MEMOING IN APPLICATIVE LANGUAGES

by

Alberto Pettorossi

Doctor of Philosophy
University of Edinburgh
1984

ABSTRACT

We study some methodologies for applicative language programming and we address the problem of improving efficiency in recursive equation programs by establishing suitable communications between subcomputations.

We first use the program transformation approach and we show that, using the generalization strategy and the tupling strategy, optimal algorithms may be obtained for the evaluation of linear recurrence relations in semiring structures. We also show that using the "tupling strategy" one may transform general recursion into linear recursion. This allows an efficient implementation of recursion without stacks and its direct translation into while-loop programs. Much other work in the area of program transformation turns out to be a special case of tupling strategy application.
We prove that for some classes of recursive equation programs it is impossible to avoid redundant computations by the use of the tupling strategy.

We also use the approach called "program annotation", by which we can dynamically specify communications between subcomputations. This method allows us to avoid redundant computations when the tupling strategy cannot work.
We apply the memo-function idea and we define the operational semantics of an applicative language with memo-functions providing their first formal treatment.

We also study the relationship between formal theories and operational semantics definitions and we describe the implemention of a deductive system for the concurrent evaluation of expressions using such definitions.

We finally say something about other methods for denoting communications in applicative languages and we study how efficiency improvements can be made by introducing concurrent computing agents and allowing messages among them.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Rod Burstall, for his helpful suggestions and continuous guidance throughout the period of my study. I am also very grateful to Robin Milner for acting as supervisor during Rod Burstall's absence, Gordon Plotkin at Edinburgh and Andrzej Skowron at Warsaw University for many stimulating conversations. My thanks go also to the colleagues at the Department of Computer Science at Edinburgh University: they, and many other friends, made my work very enjoyable and fruitful. Thanks to Eleanor Kerse for her excellent typing.

I acknowledge the financial support of the Consiglio Nazionale delle Ricerche (Italy). I was also supported by an Edinburgh University studentship. I would like to thank Lucio Bianco, Director of IASI-CNR in Rome, who allowed me to study at Edinburgh and granted me the necessary leaves of absence.

I dedicate this thesis to my parents.

DECLARATION

This thesis was composed by myself under the guidance of my supervisor Rod Burstall.
Sections 1.1 to 1.6 of Chapter 1 are an expanded version of [Pettorossi and Burstall 82]. Some of the results presented in Sections 1.7 and 1.8 were published in [Pettorossi 80a] and [Pettorossi 84]. Section 2.1 is an improved version of a part of [Plotkin 81]. Sections 2.2 - 2.3 have been submitted for publication.

Table of Contents

List of Figures

INTRODUCTION

One of the major problems in the technology of the software production is the specification and the verification of programs. A lot of research efforts are directed towards the definition of specification methods so that program development can be done without great difficulties and program correctness can easily be shown. In order to achieve those goals, among the various suggestions, applicative or functional programming has been advocated [Backus 78] because:
- it allows for proof techniques which are particularly appealing due to their simple logic and algebraic nature, and
- it allows for parallel execution of programs. Thus it overcomes some of the constraints due to the sequentiality of the von Neumann based programming languages, which often forces an "over-specification" of the algorithms.

For functional programming the "specification languages" and the "execution languages" are not so far apart, and that fact tends to make the correctness proofs more transparent and simpler.

As an introduction to functional programming and its applications, the reader may refer to [Henderson 80, Darlington, Henderson and Turner 82]. Various applicative languages have been proposed over the last two decades. Among them we recall LISP [McCarthy et al. 62], PROLOG [Kowalski 74], LUCID [Ashcroft and Wadge 77], FP systems [Backus 78], ML [Gordon, Milner and Wadsworth 79], SASL [Turner 76], KRC [Turner 81a], NPL [Burstall 77] and HOPE [Burstall, MacQueen and Sannella 80].

We will focus our attention to the ones which are more related to the

recursive equations language as defined in [Burstall and Darlington 77]. For that class of languages, which include NPL, HOPE and KRC, the operational semantics can be given in terms of rewriting rules (though some problems arise due to the nondeterminism and the possible overlapping of rules) [Boudol 83, Huet and Lévy 79].
A model for the computation of recursive functions based on rewriting systems is described already in [Kleene 67] Chapter 11, where the "replacement rule" and the "substitution rule" are used for rewriting "old" terms into "new" ones.
More recently, various people have been studying computation processes with equations and rewritings (see, for instance, [O'Donnell 77, Hoffmann and O'Donnell 82, Dershowitz 83]). Those computation processes seem to have interesting features and allow for short and elegant proofs of correctness.

However, in this style of programming, we have often to solve an efficiency problem. Indeed it may be difficult to specify efficient computations using equations. The program transformation methodology [Burstall and Darlington 77] offers a solution to that difficulty. We investigate that methodology and how we can get through it very efficient algorithms (at least for specified classes of programs).
Using transformations we can derive from ridiculous programs some reasonable programs [Burstall and Darlington 77] and also from reasonable programs some clever and very efficient ones:

$$\begin{tikzcd}\text{ridiculous programs} \arrow[r, "\text{Burstall and Darlington}"'] & \text{reasonable programs} \arrow[r, "\text{this thesis}"'] & \text{clever programs}\end{tikzcd}$$

In Section 1.1 - 1.7 of Chapter 1 we derive optimal algorithms for the evaluation of linear recurrence relations. We do so by introducing new generalization strategies and applying the "tupling strategy" (whose definition is given in Section 1.2) [Pettorossi 77] to make subcomputations to interact and cooperate as desired. Our results can be viewed as an answer to Professor Dijkstra's scepticism in the use of the transformation approach for the development of very efficient algorithms [Dijkstra 82]. Similar work has been done in [Reif and Scherlis 82] where efficient graph algorithms, as Tarjan's strong connectivity algorithm, are derived by transformation.

We look at the tupling strategy as a first approach to communications in applicative languages, in the sense that several functions can communicate among each other (thereby allowing for greater efficiency) when they are components of a function which is the tuple of all of them. The various examples given in this thesis will elucidate that idea.
Tupling also allows the parallel evaluation of expressions, because we can compute the components of a tupled function in a concurrent way.

When programming with equations, another possible cause of inefficiency is their recursive structure. That structure, while allowing straightforward proofs of correctness [Burstall 69], may not be desirable because implementing recursion is in general expensive. In Sections 1.8 - 1.9 we deal with that problem and we see that using symbolic evaluation and tupling strategy, one may transform recursive equations programs into new programs which have a particular form of recursion, called linear recursion. That kind of recursion is flowchartable (and sometimes with good efficiency [Chandra 73]), and therefore those programs can easily be rewritten into while-loop programs. Such translation can be done using a bounded number of data locations and that number is equal to the number of the functions which need to be tupled together.
Much other work done in program transformation can be viewed as a special case of application of the tupling strategy (see in particular [Cohen 83]).

Other causes of inefficiency in recursive equations programs already studied in the literature include:

- the elimination of intermediate data structures using the composition strategy [Feather 79, Wadler 81, Scherlis 80] and
- the avoidance of multiple traversal of data structures [Pettorossi 77, Feather 79, Wadler 81]. We investigate in some detail this last issue in Section 1.10.

We conclude the first chapter of the thesis with a result (see Section 1.8) concerning the limitations of the tupling strategy: we show that there exists a class of computations for which the elimination of repeated evaluations of recursively defined functions cannot be obtained by applying the tupling strategy. (We use techniques similar to the ones described in [Paterson and Hewitt 70].) That negative result is the reason for the study presented in the second part of the thesis. There we adopt another approach to efficiency improvements.

We consider a computer architecture in which programs can run in a parallel and concurrent way [Darlington and Reeve 81] and they can cooperate using some communication facilities.
Parallelism alone is not a good solution for gaining efficiency, because it may require an exponential amount of resources (as the familiar example of the Fibonacci function shows). Therefore, together with parallelism, we also need some communications among programs.
As a first step in this direction we use memo-functions for improving efficiency by remembering already computed values.
In Sections 2.1 - 2.4 we define two simple applicative languages, the first without memo-functions and the second with memo-functions. We give their structural operational semantics [Plotkin 81] (without using labelled transition systems [Plotkin 82]) and we prove their "consistency", thereby showing that

the evaluator for memo-functions improves efficiency while preserves correctness.

The use of memo-functions in recursive equations programs can also be considered in the framework of the "program annotation methodology" as first examined in [Schwarz 82]. In that methodology the process of increasing efficiency is factorized in two phases:

1. the definition of an evaluator for an "extended" recursive equations language where one can specify via annotations how recursive programs are evaluated.

2. the synthesis of the relevant annotations for improving the performances of the given programs.

In Section 2.5 we analyze some properties of the given operational semantics definitions and we study various ways in which it can be embedded in formal theories. Those embeddings show how the computational process can be viewed as a theorem proving activity in suitable theories.
In Section 2.6 we describe a Prolog implementation of those operational semantics definitions.

Finally we say something about other ways of allowing communications and parallelism in recursive equations programs. We introduce the notion of computing agents as entities which perform the evaluation of expressions, and we examine various methods of denoting communications among them. The aim is, as usual, the increase of efficiency through agents cooperation. This work is only briefly introduced at the end of the thesis, and the interested reader may refer to [Pettorossi 81a, Pettorossi and Skowron 82a, Pettorossi and Skowron 82b, Pettorossi and Skowron 83]. Related work is done in [Dennis 74, Kahn and MacQueen 77, Hewitt and Baker 78, Hoare 78, Milner 80].

. . . quia per facilia ad difficilia oportet devenire.

(. . . for difficult things ought to be reached by way of easy ones.)

"De Modo Studendi" (Thomas Aquinas. 1225–1274)

Chapter 1

DERIVING EFFICIENT APPLICATIVE PROGRAMS : COMMUNICATIONS VIA TUPLING STRATEGY

1.1 Introduction

In this part of the thesis we will analyse the program transformation technique à la Burstall-Darlington [Burstall and Darlington 77] for deriving correct and efficient programs.

We will consider a particular class of programs concerning the evaluation of linear recurrence relations, and we will try to derive them by applying the transformation technique. In the course of the derivation we will discuss the power of the transformation strategies and we will compare the efficiency of the programs we obtained with that of already known algorithms. We will also be concerned with the comparison of the program transformation technique à la Burstall-Darlington [Burstall and Darlington 77] with the stepwise refinement technique advocated by Dijkstra and Wirth [Wirth 71].

The choice of a particular class of problems is motivated by the need of a concrete test-case, so that we could compare the features of the program transformation technique in a domain where other techniques have been found very valuable. A similar approach has already been taken in the past and, for instance, in [Darlington 78] we can find a detailed study of the derivation of sorting algorithms. We think that those results and ours show some very good features of the program transformation technique and give interesting insights on the power of the transformation strategies.

The particular case of the Fibonacci function is first considered and a

comparison with the conventional matrix exponentiation algorithm is made. Then we generalize the derivation of the program for computing the Fibonacci function to the case of linear recurrence relations with constant coefficients. It turns out that the rules and the strategies one uses for Fibonacci are powerful enough to lead in the general case also, to an efficient algorithm in a quite straightforward way.
The choice of the evaluation of linear recurrence relations was suggested to us by Professor Dijkstra, who considered it to be a challenging one for exploring the practical interest of the program transformation technique, especially in contrast with the use of the stepwise refinement technique.

1.2 Preliminary considerations and definitions

Transforming recursive equations is a good methodology for writing correct and efficient programs [Burstall and Darlington 77]. Following this methodology the programmer is first asked to be concerned with program correctness only and then, at later stages, with efficiency considerations. The original version of the program, which he may easily prove correct, is transformed (perhaps in several steps) into a program which is still correct, because the rules used for the transformation preserve correctness, and it is more efficient because the evoked computations save time and/or space. Several papers have been written about this methodology concerning:
(i) systems for transforming programs [Bauer et al. 77], [Darlington and Burstall 76], [Feather 78], [Scherlis 80], [Darlington 81], [Feather 82];
(ii) various rules for making such transformations [Arsac and Kodratoff 82], [Chatelin 77], [Pettorossi 78], [Pettorossi 77], [Schwarz 78], [Manna and Waldinger 79], [Scherlis 80], [Wand 80]; and
(iii) some theories for proving the correctness of the transformations [Bauer et al. 78], [Huet and Lang 78], [Kott 78].
The given lists of references are not to be considered as exhaustive. For a

more extensive bibliography one may refer to [Burstall and Feather 77], [Partsch and Steinbrüggen 81] and [Partsch and Steinbrüggen 83].

Unfortunately for the program transformation methodology a general framework in which one can prove that transformations improve efficiency while preserving correctness, is not fully available yet. First steps in this direction were done in [Wegbreit 76, Scherlis 80].
One might wonder in fact whether it is important to develop such a general framework at all: it might be the case that using the transformation methodology one can derive algorithms which have a limited degree of efficiency only, and that if one wanted to write very efficient programs one would be forced to adopt other programming techniques, as for example, the stepwise refinement technique [Dahl et al. 72], [Wirth 71]. But this does not seem to be the case, at least for particular classes of algorithms, as demonstrated in this thesis for algorithms which evaluate linear recurrence relations. Working out the details of the program transformation, we show that very high levels of efficiency can be achieved.
In order to be able to reason about the program transformation approach in a precise way we will now introduce a suitable formal system. Our presentation is somehow inspired by Chapter XI of Kleene's book "Introduction to Metamathematics" [Kleene 67] and more recent work on the theory of formal languages (see for instance [Hopcroft and Ullman 79]). However the main inspiration is the recent work by G. Plotkin on a structural approach to operational semantics definitions [Plotkin 81].
We first define the language in which we write our recursive-equations programs. We consider the following basic sets:

1. Numbers $m \in N$ $N = \{0,1,2,...\}$

2. Truthvalues $t \in T$ $T = \{tt, ff\}$

3. Individual Variables $x, y, z, \ldots \in$ IVar

4. Functional Variables (with arities) $f, g, \ldots \in \text{FVar} = \bigcup_i \text{FVar}_i$, where FVar_i is the set of functional variables with arity i.

5. Basic Operators (with arities) $\text{bop}_i \in \text{Bop} = \bigcup_i \text{Bop}_i$, where Bop_i is the set of basic operators with arity i.

The set Var = IVar ∪ FVar is a (possibly infinite) denumerable set of symbols.

Notice that the set Numbers and possibly other basic sets we may want to define, as for instance Lists, Trees, etc., can be introduced by using constructors, which will be considered as basic operators. We may in fact define the set Constr of constructors, ranged over by cons $\in$ Constr $\subseteq$ Bop. Elements of Constr have arities as all basic operators do.
For instance, using the constructors 0 (with arity zero) and succ (with arity one) we can define Numbers as follows:

N = {0, succ(0), succ(succ(0)), . . .}

Using the constructors "empty", "tip" and "node" we can define the set of Trees-of-numbers, as the follows:

- "empty" denotes the empty tree;
- "tip(n)" denotes the tree with one tip only whose number is n;
- "node(t1, t2)" denotes the tree with the two subtrees t1 and t2.

Starting from these basic sets we can define the following <u>derived sets</u>:

<u>Basic Expressions</u> be $\in$ BasExp

be ::= m | t | x | cons(. . . , be, . . .)

<u>Expressions</u> e $\in$ Exp

$e ::= be \mid bop_i(\ldots,e,\ldots) \mid$ <u>if</u> e_0 <u>then</u> e_1 <u>else</u> e_2

e <u>where</u> $d \mid f(\ldots,e,\ldots)$

<u>Definitions</u> $d \in Def$

$d ::= x=e \mid \langle x_1,\ldots,x_n \rangle = e$ where $n \geq 2$

The angle-brackets in $\langle x_1,\ldots,x_n \rangle$ denote the n-ary tupling operator. We assume that for any $k \geq 1$ there exists in Bop a k-ary tupling operator, and that the unary tupling operator is the identity function.

<u>Recursive Equations</u> $req \in Req$

$req ::= f(\ldots,be,\ldots) \Leftarrow e$

We may use = instead of $\Leftarrow$, when no confusion arises with the equality predicate.
(The notion of Basic Expressions, as particular kinds of Expressions, is needed only for the definition of the left-hand-sides of the Recursive Equations).

<u>Programs</u> are sets of recursive equations.
For example :

$f(x) \Leftarrow$ <u>if</u> $x=0$ <u>then</u> 1 <u>else</u> $x \cdot f(x-1)$

is a recursive equation and it denotes a program which is made out of that recursive equation only. Sometimes we do not use the <u>if</u>-<u>then</u>-<u>else</u> construct and we break a recursive equation into several ones. For instance the following two equations:

$f(0) \Leftarrow 1$

$f(succ(n)) \Leftarrow succ(n) \cdot f(n)$

are equivalent to the one given above. Notice that here we assumed $n \in IVar$. We also omitted (as we will often do) the curly brackets to group together recursive equations to form a program.

Moreover, we like to avoid recursive equations such as:

$f(n) \Leftarrow m \cdot n$ because the variable m is not defined

or $f(n) \Leftarrow a^2+n^2$ where $z=3n$ because z does not occur in a^2+n^2.

In the Appendix A we will give the static semantics for recursive equations (this terminology comes from [Plotkin 81]), so that we may restrict ourselves to well-formed recursive equations only, and we may avoid the anomalous cases given above.

We should complete the presentation of our language for recursive equations programs by giving also its dynamic operational semantics, i.e. the rules for evaluating expressions.

However, in the second part of the thesis we will define the operational semantics of a simple applicative language L, which could be considered as an extension of the recursive equations language introduced here. Therefore we ask the reader to refer to the definitions given there.
For the time being we hope that he may be satisfied by knowing that expressions are evaluated by the use of a deductive system (similar to the one used in [Kleene 67] Chapter XI), which replaces instances of left hand sides of recursive equations by the corresponding instances of right hand sides (in any context they might occur) and performes the operations denoted by the basic operators.

Now we will introduce some more concepts for reasoning about transformations of recursive equations programs.
"New" programs can be derived from "old" programs, by using rules and/or strategies.
Here are the most usual rules one encounters in the literature.

1. Definitions Rule. It consists in the introduction of a new recursive equation, say $f(\ldots) \Leftarrow e$, so that the "left hand side" $f(\ldots)$ is not an instance of any left hand side of equations already existing in the program.
We say that an expression e_1 is an instance of an expression e_2 iff there is a substitution σ so that $\sigma(e_2)=e_1$; in that case we may say that e_1 matches e_2 (one-way unification).
Here are some examples: f(succ(n)) is an instance of f(m), where σ(m)=succ(n). g(succ(n),m) is not an instance of g(p,succ(q)).

2. Instantiation Rule. This rule consists in the introduction of an instance of an existing equation. This rule already occurs in [Kleene 67] where it is called "substitution rule".
For instance g(n,0) $\Leftarrow$ 0 can be obtained via instantiation from g(n,m) $\Leftarrow$ m · fact(n) .

3. Unfolding Rule. In an already existing equation, an occurrence of an instance of the l.h.s. of an equation is replaced by the corresponding instance of the associated r.h.s., thereby producing a new equation.
Here is an example. Suppose we are given the following program:

```
f(0)          ⇐ 1
f(succ(n))    ⇐ n·f(n).
```

Then we could derive by instantiation the equation:

```
f(succ(succ(n)) ⇐ succ(n) · f(succ(n))
```

and then by unfolding we get:

```
f(succ(succ(n)) ⇐ succ(n) · (n · f(n)).
```

□

This rule has been already considered in [Kleene 67] Chapter 11, where it is called "replacement rule". It is also the rule used in term rewriting systems (see for instance [Huet and Oppen 80]) for defining computations using rewrite rules.

4. Folding Rule (as the unfolding rule, interchanging l.h.s. and r.h.s.)

5. Where-Abstraction Rule. Given a recursive equation $f(\ldots) \Leftarrow e$ such that $e \equiv C[e']$ where C is a context denoting that the subexpression e' occurs in e, we introduce the new equation $f(\ldots) \Leftarrow C[z]$ where $z=e'$ provided that z is a "new" variable symbol, i.e. a variable symbol not occurring in $f(\ldots) \Leftarrow e$.

6. Laws Rule. We can define a new equation by transforming a subexpression using some rules which hold in the algebra of the basic operators. For instance $f(succ(n)) \Leftarrow n \cdot f(n)$ can be transformed into $f(succ(n)) \Leftarrow f(n) \cdot n$ because $\cdot$ is commutative.

Strategies for transforming programs are ways in which new function definitions may be derived. New derived functions may be introduced in the given program using the "definition rule".
Well-known strategies are the following ones [Burstall and Darlington 77]:

1. Composition. If a subexpression $f(g(x))$ occurs in an expression e, we define $h(x)=f(g(x))$ and replace $f(g(x))$ by $h(x)$ in e.
Using this strategy one may derive a new program which, hopefully, is more efficient. For instance, one may avoid the construction of intermediate data structures, which are the output of g and the input for f. Some examples are in [Burstall and Darlington 77, Feather 79, Scherlis 80].
The composition strategy is also very useful when the output of f does not depend on the entire output of g. In that case we can obtain some of the advantages of the lazy evaluation mechanism.

2. Tupling. (In the literature this strategy has been also called pairing, when only two functions are considered)([Burstall and Darlington 77] page 49).
If in an expression e, the functions $f_1(x)$, $f_2(x)$, ..., $f_n(x)$ occur, we may define $h(x)=\langle f_1(x), f_2(x), \ldots, f_n(x)\rangle$. Then in e we will use $\pi i(h(x))$ instead of $f_i(x)$, where πi denotes the i-th projection function, for $i=1,\ldots,n$.

The use of the tupling strategy is very useful if the functions $f_1(x), \ldots, f_n(x)$ have all to access the same data structure x (and no other function is using x). The use of x will then be restricted to the computation of h(x) and the store used by x can be released as soon as h(x) has been computed. This allows great improvements in "time×memory" efficiency while executing recursive equations programs [Pettorossi 77].
The use of the tupling strategy is also very effective when common subexpressions occur in evaluating several functions. Tupling those functions together results in an improvement of the performances. Various examples of that fact will be given later in the thesis.

3. Generalization. It could be of two kinds.
i) Generalization from expressions to functions.
If an expression e occurs in a program and the variables $x_1, \ldots, x_n$ occur free in it, then we may define a new function f via the equation $f(x_1, \ldots, x_n) \Leftarrow e$ and replace e by $f(x_1, \ldots, x_n)$ itself [Burstall and Feather 77]. Generalizations of this kind are somewhat similar to those in [Aubin 79] and [Boyer and Moore 75].
ii) Generalization from functions to functions.
It can be done in two different ways. The new function can be introduced either by cases or by implicit definition [Pettorossi and Burstall 82].

Definition by cases.
Let us consider the following program:
$\{ f(be_{11}, \ldots) \Leftarrow e_1, \ldots, f(be_{n1}, \ldots) \Leftarrow e_n \}$
which defines the function f and let us suppose that some constants, say $c_1, \ldots, c_m$, occur in the expressions $e_1, \ldots, e_n$. We can generalize those constants to variables, say $y_1, \ldots, y_m$, and define the same function f as follows:
$\{ f(\ldots) \Leftarrow f2(\ldots, c_1, \ldots, c_m),$
$f2(be_{11}, \ldots, y_1, \ldots, y_m) \Leftarrow e'_1, \ldots, f2(be_{n1}, \ldots, y_1, \ldots, y_m) \Leftarrow e'_n \}$

where $e'_1, \ldots, e'_n$ are expressions which can easily be derived from $e_1, \ldots, e_n$ using the fact that $f(\ldots) = f2(\ldots, c_1, \ldots, c_m)$.
For example, from:

$\{ f(0)=1,\ f(n+1)=(n+1)\cdot f(n) \}$ we get:

$\{ f(n)=f2(n,1),\ f2(0,y)=y,\ f2(n+1,y)=(n+1)\cdot f2(n,y) \}$.

<u>Implicit definition</u>.
Given the program:

$\{ f(be_{11}, \ldots) \Leftarrow e_1, \ \ldots, \ f(be_{n1}, \ldots) \Leftarrow e_n \}$

which defines the function f, we can introduce the new functions $g_1(\ldots), \ \ldots, \ g_m(\ldots)$ by implicit definition, using the following recursive equation:

$$f(\ldots) \Leftarrow F(\ldots, g_1(\ldots), \ldots, g_m(\ldots)),$$

where F is a functional which we have to invent and which may depend on the arguments of f as well.
For example, given the program:

$\{ G(a,b,0)=a,\ G(a,b,1)=b,\ G(a,b,n+2)=G(a,b,n)+G(a,b,n+1) \}$

we may introduce by implicit definition the new functions p(n) and q(n) using the equation:

$G(a,b,n) = F(a,b,n,p,q) = p(n)\cdot a + q(n)\cdot b.$

The explicit definition of p(n) and q(n) gives us:

$p(n)=G(1,0,n)$ and $q(n)=G(0,1,n)$.

We will see some more examples of those generalization strategies in what follows.

The basic idea of the program transformation technique as introduced by Burstall and Darlington [Burstall and Darlington 77], is to derive from a given recursive equations program, another one which computes the same function in a more efficient way.

In such a derivation two major issues are to be considered: equivalence and efficiency.

The equivalence issue consists in the fact that the derived program, say P', should be equivalent to the given one, say P, in a strong sense, i.e. $\forall x$ if P(x) terminates and computes the value f(x) then P'(x) terminates and computes the same value f(x).
Sometimes one may encounter a "termination problem". Indeed, using the strategies and the rules we have listed above, the derived program P'(x) may fail to terminate for input values for which P(x) terminates.
Let us see the following example.
Given the program: 1. $f(0) \Leftarrow 0$

2. $f(succ(n)) \Leftarrow f(n)$.

we can get by instantiation from 2:

3. $f(succ(succ(n))) \Leftarrow f(succ(n))$

by unfolding from 2 and 3:

3'. $f(succ(succ(n)) \Leftarrow f(n)$

and eventually by folding f(n) in 2 using 3' we get:

2'. $f(succ(n)) \Leftarrow f(succ(succ(n)))$.

Now the derived program (equations 1 and 2') for the function f terminates only for the input 0, and therefore it does not compute the function computed by the given program (equations 1 and 2), which was the constant function yielding always the value 0. □
Work related with this equivalence issue was done in [Kott 78, Scherlis 80].
In order to make sure that termination is preserved, Kott suggested a method based on counting the number of foldings and unfoldings, while Scherlis adopted transformation rules which correspond to a restricted use of the ones we have presented above.

The efficiency issue is related to the time and/or memory requirements in running a derived program with respect to a given one.

In [Burstall and Darlington 77] the time requirement considered is the number of recursive calls necessary to compute the result. We will also use such a measure in the next sections when deriving efficient algorithms for evaluating linear recurrence relations.
Ways of improving memory requirements via program transformation or "destructiveness analysis" have been considered in [Pettorossi 78] and [Mycroft 81].

1.3 Deriving an algorithm for computing the Fibonacci function in logarithmic time

We start off from the familiar definition:

Program P1

1. $fib(0) = 1$
2. $fib(1) = 1$
3. $fib(n+2) = fib(n+1) + fib(n)$ for $n \geq 0$

and we look for a logarithmic running time program for computing the Fibonacci function. (In [Burstall and Darlington 77] a linear running time program is obtained.) We will use the transformation method and its strategies for deriving "new" (and more efficient) programs from "old" ones.

Obviously Program P1 is a well-formed program (see Appendix A).

As it has been the case for many other functions (see for instance, the factorial function in [Burstall and Feather 77]), in order to improve the running time efficiency we can apply the "generalization strategy" already described, by transforming the constant 1 in equation 1 into the variable a_0 and the constant 1 in equation 2 into the variable a_1.

These variables will be considered as extra arguments of a new function, as usually done when using the generalization strategy [Burstall and Feather 77]. Therefore we get the following generalized Fibonacci function:

(GENERALIZATION EUREKA)

4. $G(a_0, a_1, 0) = a_0$
5. $G(a_0, a_1, 1) = a_1$
6. $G(a_0, a_1, n+2) = G(a_0, a_1, n+1) + G(a_0, a_1, n)$ for $n \geq 0$

The word "eureka" which annotates the above generalization step, is used throughout this paper as in [Burstall and Darlington 77], for denoting unobvious steps in the transformations of our programs.

The function G has been obtained by applying the <u>generalization with definition by cases</u> (see Section 1.2).
It allows us to compute the Fibonacci function starting from any two initial values a_0 and a_1.

Obviously we can derive the following program for computing fib(n):

<u>Program P1.1</u>

7. $fib(n) = G(1,1,n)$ for $n \geq 0$
4. $G(a_0,a_1,0) = a_0$
5. $G(a_0,a_1,1) = a_1$
6. $G(a_0,a_1,n+2) = G(a_0,a_1,n+1) + G(a_0,a_1,n)$ for $n \geq 0$

Program P1.1 is well-formed.

Looking for a fast way of computing G, we need to relate $G(a_0,a_1,n+2)$ not to $G(a_0,a_1,n+1)$ and $G(a_0,a_1,n)$ as in equation 6, but to "more distant" values, say $G(a_0,a_1,n)$ and $G(a_0,a_1,n-1)$. This can be achieved by using the "unfolding rule". We get:

$$\begin{aligned} G(a_0,a_1,n+2) &= G(a_0,a_1,n+1) + G(a_0,a_1,n) \\ &= 2\cdot G(a_0,a_1,n) + G(a_0,a_1,n-1) \end{aligned}$$

by unfolding $G(a_0,a_1,n+1)$ using 6.

We can iterate this unfolding process and we get:

$$G(a_0,a_1,n+2) = 3\cdot G(a_0,a_1,n-1) + 2\cdot G(a_0,a_1 n-2)$$

by unfolding $G(a_0,a_1,n)$ in the previous expression.

Eventually we get:

$$G(a_0,a_1,n+2) = c_1\cdot G(a_0,a_1,0) = c_1\cdot a_1+c_2\cdot a_0,$$

where the two values c_1 and c_2 depend on n, i.e. the "distance" of $G(a_0,a_1,n+2)$ from $G(a_0,a_1,1)$ and $G(a_0,a_1,0)$.
This reasoning motivates the following "eureka step" (more "ad hoc" than the

previous generalization eureka), by which we generalize the constants c_1 and c_2 and we introduce the implicit definition of two new functions p(n) and q(n):

8. $G(a_0, a_1, n) = p(n) \cdot a_0 + q(n) \cdot a_1$ (LINEAR COMBINATION EUREKA)

This "eureka" is an application of the generalization by implicit definition as introduced in Section 1.2.

The program transformation process continues, as usual, by looking for an explicit definition of the newly introduced functions p(n) and q(n).
From 4 and 8 (for n=0) we get: $a_0 = p(0) \cdot a_0 + q(0) \cdot a_1$. Since this equality should hold for all values of a_0 and a_1, we have q(0) = 0 and hence p(0) = 1.
Analogously from 5 and 8 (for n=1) we get: $a_1 = p(1) \cdot a_0 + q(1) \cdot a_1$, which yields p(1) = 0 and q(1) = 1.
From 6 and 8 (for n+2 instead of n) we get:
$G(a_0, a_1, n+1) + G(a_0, a_1, n) = p(n+2) \cdot a_0 + q(n+2) \cdot a_1$
By 8 we have:
$p(n+1) \cdot a_0 + q(n+1) \cdot a_1 + p(n) \cdot a_0 + q(n) \cdot a_1 = p(n+2) \cdot a_0 + q(n+2) \cdot a_1$
and therefore, since this equality should hold for all values of a_0 and a_1, we get: p(n+2) = p(n+1) + p(n) and q(n+2) = q(n+1) + q(n).

By 4, 5 and 6 we get: p(n) = G(1, 0, n) and q(n) = G(0, 1, n).
Thus the LINEAR COMBINATION EUREKA can be rewritten as follows:
8′. $G(a_0, a_1, n) = G(1, 0, n) \cdot a_0 + G(0, 1, n) \cdot a_1$

Since the function $G(a_0, a_1, n)$ computes the Fibonacci function starting from the initial values a_0 and a_1, if we substitute in 8′ fib(n) for a_0 and fib(n+1) for a_1 we get that $G(a_0, a_1, n)$ is equal to fib(2n); thus:

8". fib(2n) = G(fib(n), fib(n+1), n)

$= G(1,0,n) \cdot fib(n) + G(0,1,n) \cdot fib(n+1)$ by 8′

This equation allows to derive a logarithmic running time algorithm for computing the Fibonacci function, because the argument 2n has been divided by 2 and because analogous equations can be derived for $G(1,0,n)$ and $G(0,1,n)$.

However, in order to obtain equation 8" using the program transformation technique, we need a second generalization step, by which we will obtain a general formula for computing $G(a_0,a_1,n)$ in logarithmic time for any value of the variables a_0, a_1 and n.

From equation 6 we can generalize the constant 2 occurring in $G(a_0,a_1,n+2)$ into a new variable, say k, which will be, as usual, an extra argument of the resulting function. Therefore we define the function $F(a_0,a_1,n,k)$ as follows:

9. $F(a_0,a_1,n,k) = G(a_0,a_1,n+k)$ (GENERALIZATION EUREKA)
(for the function F)

Now we look for the explicit definition of F, in terms of the argument k just introduced. We obtain:

$F(a_0,a_1,n,0) = G(a_0,a_1,n)$

$F(a_0,a_1,n,1) = G(a_0,a_1,n+1)$

$F(a_0,a_1,n,k+2) = G(a_0,a_1,n+k+2)$

$= G(a_0,a_1,n+k+1) + G(a_0,a_1,n+k)$ by unfolding (using 6)

$= F(a_0,a_1,n,k+1) + F(a_0,a_1,n,k)$ by folding (using 9)

Now it is possible to make for F the same transformation steps made for G, because F and G obey the same recurrence relation.

We can apply for F the "linear combination eureka" (see equation 8), and we get:

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

because $G(a_0, a_1, n)$, $G(a_0, a_1, n+1)$ and k play for F the role of a_0, a_1 and n respectively for G.

Looking for the explicit definition of r(k) and s(k) we obtain (by making the same derivations performed for the function G) that:

$r(0) = 1,\ r(1) = 0,\ r(k+2) = r(k+1) + r(k)$ and

$s(0) = 0,\ s(1) = 1,\ s(k+2) = s(k+1) + s(k)$.

Thus we have as before:

$r(k) = G(1, 0, k)$ and $s(k) = G(0, 1, k)$.

We get:

$F(a_0, a_1, n, k) = G(1, 0, k) \cdot G(a_0, a_1, n) + G(0, 1, k) \cdot G(a_0, a_1, n+1)$

and from it we obtain, using 9, the following equation:

9'. $G(a_0, a_1, n+k) = G(1, 0, k) \cdot G(a_0, a_1, n) + G(0, 1, k) \cdot G(a_0, a_1, n+1)$

(<u>EFFICIENCY EQUATION</u>)

Equation 9' allows us to achieve the logarithmic running time algorithm we were looking for. In fact, if we want to compute fib(2k) we need to compute G(1, 1, 2k) (by equation 7) and for computing G(1, 1, 2k) according to equation 9', we need only to know G(1, 0, k), G(1, 1, k), G(0, 1, k) and G(1, 1, k+1) which are values of the function G around the point k. As we already mentioned, this division by 2 of the relevant argument allows the desired efficiency.

From equation 9' we can derive the new program P2 for computing the Fibonacci function:

<u>Program P2</u>

10. $fib(0) = 1$ by 1

11. $fib(1) = 1$ by 2

12. $fib(2k) = G(1,1,2k)$ by 7

$= G(1,0,k) \cdot G(1,1,k) + G(0,1,k) \cdot G(1,1,k+1)$ by 9'

13. $fib(2k+1) = G(1,1,2k+1)$ by 7

$= G(1,0,k) \cdot G(1,1,k+1) + G(0,1,k) \cdot G(1,1,k+2)$ by 9'

14. $G(a_0,a_1,0) = a_0$ by 4

15. $G(a_0,a_1,1) = a_1$ by 5

16. $G(a_0,a_1,2k) = G(1,0,k) \cdot G(a_0,a_1,k) + G(0,1,k) \cdot G(a_0,a_1,k+1)$ by 9'

17. $G(a_0,a_1,2k+1) = G(1,0,k) \cdot G(a_0,a_1,k+1) + G(0,1,k) \cdot G(a_0,a_1,k+2)$ by 9'

Unfortunately, as one can see from fig. 1-1, program P2 may evoke computations with redundant evaluations. In order to guarantee a logarithmic running time for P2 one should avoid them (or, at least, prove them to be suitably bounded).
Notice that redundant evaluations can be detected by symbolic evaluation alone, using "unfolding": in our case, for example, when computing fib(2k), we will compute G(1,0,k/2) 4 times (see fig. 1-1).

But this difficulty can be easily solved by using the "tupling strategy" [Pettorossi 77]. The "tupling strategy" consists in defining a new function as the tuple of all functions which require common subcomputations. We will apply it by "tupling" together G(1,0,k), G(1,1,k), G(0,1,k) and G(1,1,k+1) because they all need the computation of G(1,0,k/2) (see fig. 1-1).

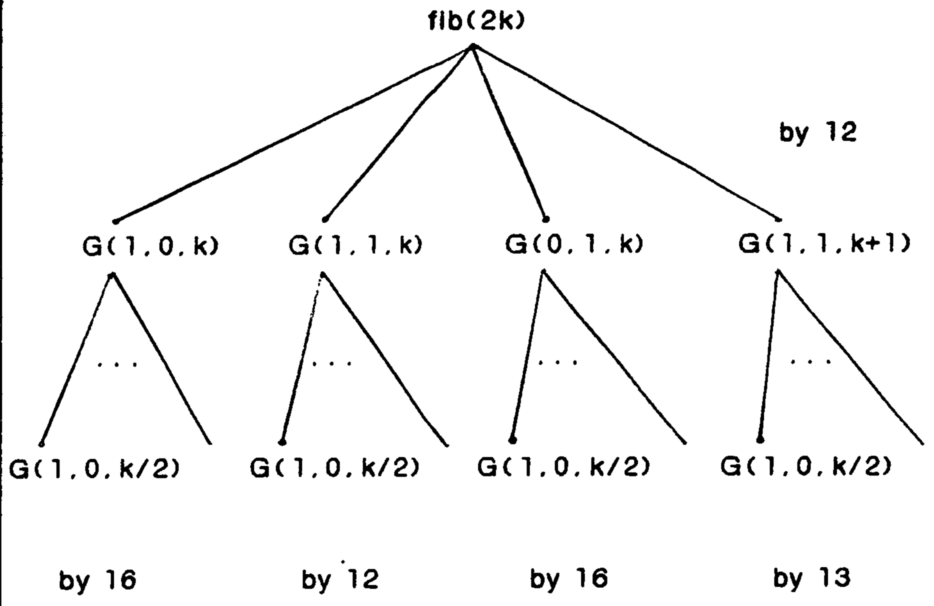


Figure 1-1: Redundant computations in program P2: G(1,0,k/2) occurs 4 times in evaluating fib(2k). (Suppose k even.)

Therefore we will define the following auxiliary function t(k):

18. $t(k) = \langle G(1,0,k), G(0,1,k), G(1,1,k), G(1,1,k+1)\rangle$ (<u>TUPLING EUREKA</u>)

The order in which the individual functions G's occur in t(k) is immaterial. The explicit definition of the function t(k) can easily be derived by standard applications of the "folding" and "unfolding" rules using equations 4, 5, 6 and 9' and we get:

19. $t(0) = \langle 1,0,1,1\rangle$

20. $t(2k) = \langle G(1,0,2k),\ G(0,1,2k),\ G(1,1,2k),\ G(1,1,2k+1)\rangle$

$= \langle G(1,0,k)\cdot G(1,0,k) + G(0,1,k)\cdot G(1,0,k+1),$

$G(1,0,k)\cdot G(0,1,k) + G(0,1,k)\cdot G(0,1,k+1),$

$G(1,0,k)\cdot G(1,1,k) + G(0,1,k)\cdot G(1,1,k+1),$

$G(1,0,k)\cdot G(1,1,k+1) + G(0,1,k)\cdot G(1,1,k+2)\rangle$

by 9'

$= \langle a^2+b^2,\ b(2a+b),\ ac+bd,\ ad+b(c+d)\rangle$ <u>where</u> $\langle a,b,c,d\rangle = t(k)$

by 6 and 9'

21. $t(2k+1) = \langle G(1,0,2k+1),\ G(0,1,2k+1),\ G(1,1,2k+1),\ G(1,1,2k+2)\rangle$
$= \langle 2ab+b^2,\ (a+b)^2+b^2,\ ad+b(c+d),\ a(c+d)+b(2d+c)\rangle$
<u>where</u> $\langle a,b,c,d\rangle = t(k)$

Therefore we can now obtain the following nonredundant program P3, which computes the Fibonacci function in logarithmic time.

<u>Program P3</u>

```
10.  fib(0)      = 1
11.  fib(1)      = 1
12'. fib(2k)     = ac+bd          where <a,b,c,d> = t(k)
13'. fib(2k+1)   = ad+b(c+d)      where <a,b,c,d> = t(k)
19.  t(0)        = <1,0,1,1>
20.  t(2k)       = <a²+b², b(2a+b), ac+bd, ad+b(c+d)>
                                  where <a,b,c,d> = t(k)
21.  t(2k+1)     = <2ab+b², (a+b)²+b², ad+b(c+d), a(c+d)+b(2d+c)>
                                  where <a,b,c,d> = t(k)
```

Notice that, instead of "tupling" the functions which cause redundant computations in equation 12, namely G(1,0,k), G(0,1,k), G(1,1,k) and G(1,1,k+1), as we did, we could have tupled the functions G(1,0,k), G(0,1,k), G(1,1,k+1) and G(1,1,k+2) which cause redundant computations in equation 13. The resulting program would have been equivalent to program P3. In Section 1.8 we will discuss that fact and we will see other interesting properties of the tupling strategy.

We also can simplify program P3 using the following equation which can be derived from equations 7 and 18:

22. $fib(n+1) = \pi3(t(n))$

where, in general, $\pi i(\langle e0,e1,\ldots,ek\rangle) = ei$ for $i=0,\ldots,k$.

Notice that πi denotes the $(i+1)$st projection function. It is not a usual convention, but it will shorten our notations in what follows. When no confusion arises, we will feel free to use πi to denote the i-th projection as well.

We obtain program P4.

Program P4

```
10. fib(0)     = 1
22. fib(n+1)   = π3(t(n))          where π3(<e0,e1,e2,e3>) = e3
19. t(0)       = <1,0,1,1>
20. t(2k)      = <a²+b², b(2a+b), ac+bd, ad+b(c+d)>
                                   where<a,b,c,d> = t(k)
21. t(2k+1)    = <2ab+b², (a+b)²+b(2), ad+b(c+d), a(c+d)+b(2d+c)>
                                   where <a,b,c,d> = t(k)
```

We could also have derived the equation:

22'. $fib(n) = \pi 2(t(n))$ for $n \geq 0$

and we could have used it in program P4 instead of equations 22 and 10. But, in evaluating $\pi 2(t(n))$ we should avoid computing all 4 components of $t(n)$, because otherwise we would also compute $\pi 3(t(n))$ which is equal to fib(n+1), and it seems awkward to compute fib(n+1) while computing fib(n). A solution to this inconvenience can be obtained by using a "call by need" evaluation mode for the projection function $\pi 2$ or by introducing suitable conditional expressions (see Example 9 Section 1.8.3).

In fig. 1-2 the reader may have a synoptic account of the program transformation steps we have made so far.

Program P1 (1, 2, 3): Definition. Exponential time.

↓

Generalization Eureka (4, 5, 6) defining $G(a_0, a_1, n)$

↓

Program P1. 1 (7, 4, 5, 6). Exponential time.

↓

Linear Combination Eureka for $G(a_0, a_1, n)$ (8′)

Generalization Eureka defining $F(a_0, a_1, n, k)$ (9)

Efficiency Equation (9′)

↓

Program P2 (from 10 to 17). "Almost" logarithmic time.

4 functions require common subcomputations.

↓

Tupling strategy with 4 functions (19, 20, 21)

↓

Program P3 (10, 11, 12′, 13′, 19, 20, 21). Logarithmic time.

As the 2×2 Matrix Exponentiation Method.

↓

Simplification strategy (22)

↓

Program P4 (10, 19, 20, 21, 22). Logarithmic time.

As program P3, but fewer equations.

Figure 1-2: Deriving logarithmic running time algorithms for computing the Fibonacci function. (We have annotated the transformation steps and the programs with the corresponding equation numbers.)

1.4 A comparison with other algorithms

Linear recurrence relations can also be evaluated in logarithmic time using the well-known matrix exponentiation method [Miller and Brown 66, Hoggatt 69]. We will recall it in the case of the Fibonacci function evaluation. Directly from the definition, we have:

$$\begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} fib(1) \\ fib(0) \end{bmatrix}$$

and, in general, we have:

$$\begin{bmatrix} fib(k+1) \\ fib(k) \end{bmatrix} = M^k \cdot \begin{bmatrix} fib(1) \\ fib(0) \end{bmatrix} = M^k \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{for } k \geq 0$$

where $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$.

Therefore, in order to compute fib(k+1) we have to compute M^k. This matrix can be computed in logarithmic time, because we can compute the values of $M, M^2, M^4, M^8, \ldots$ by successive squarings starting from the matrix M and then we can multiply together those matrices whose exponents contribute to a sum equal to k [Miller and Brown 66, Hoggatt 69, Steinbrüggen 77]. More formally, we have: $M^k = \prod_{i=0}^{n} M^{b_i 2^i}$, where the b_i's are defined by the binary expansion of k, i.e. $k = \sum_{i=0}^{n} b_i 2^i$.

Example 1. For computing fib(14), since 13 = 1+4+8, we have:

$$\begin{bmatrix} fib(14) \\ \\ fib(13) \end{bmatrix} = M^{13} \cdot \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix} = M \cdot M^4 \cdot M^8 \cdot \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 5 & 3 \\ \\ 3 & 2 \end{bmatrix} \cdot \begin{bmatrix} 34 & 21 \\ \\ 21 & 13 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 377 & 233 \\ \\ 233 & 144 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}$$

Thus fib(14) = 337 + 233 = 610. □

What kind of relationship exists between the matrix exponentiation method and our method as defined by program P4? The answer to this question is given by the following fact, which shows that the 4-tuple t(k) directly corresponds to M^k for any k≥0.

Fact 1 Let M be $\begin{bmatrix} 1 & 1 \\ \\ 1 & 0 \end{bmatrix}$, M^k be $\begin{bmatrix} c & b \\ \\ d & a \end{bmatrix}$ and t(k) be defined as in program P4. $\forall k \geq 0$ t(k) = <a, b, c, c+d>.

Proof By induction on k. □

It is very interesting to notice that the use of the tupling strategy, which we applied in the transformation approach only for avoiding redundant evaluations of common subexpressions, "rediscovered", as Fact 1 indicates, the matrix exponentiation approach, by satisfying the apparently unrelated requirement of improving efficiency. But the relationship between the two approaches can be shown to be even stronger, as we will now indicate.

The matrix exponentiation method can be improved because the following

Fact 2 allows us to derive the matrix M^{2k} from the matrix M^k, for any $k \geq 0$, by knowing only the 2 elements of the last column of M^k [Miller and Brown 66].

Fact 2. Under the hypotheses of Fact 1, $\forall k \geq 0$ $d=b$ and $c=a+b$, i.e. the matrices M^k's are symmetric and they have only two independent elements, namely a and b. Proof. By induction on k. □

From Fact 2 it follows that in order to derive M^{2k} from M^k it is enough to know only one row or one column of M^k. That property is indeed valid for any linear recurrence relation with constant coefficients, as we will see in the next section.

Can we "rediscover" the improved version of the matrix exponentiation method (in which we compute only two elements of the matrices M^k, for any $k \geq 0$, by using Fact 2) by applying the tupling strategy for avoiding redundant computations so that only two functions (instead of four as we did in the definition of t(k)) are tupled together?
The answer is "yes" and the corresponding program can be derived from program P2 as follows.
The technique used is an application of what we may call a simplification strategy. We can transform equations 12 and 13 for minimizing the number of distinct values occurring in them.

12". fib(2k) $= G(1,0,k) \cdot G(1,1,k) + G(0,1,k) \cdot G(1,1,k+1)$ by 12

$= G(1,0,k) \cdot (G(1,0,k) \cdot G(1,1,0) + G(0,1,k) \cdot G(1,1,1))$

$+ G(0,1,k) \cdot (G(1,0,k) \cdot G(1,1,1) + G(0,1,k) \cdot G(1,1,2))$

by 9′

$= (a+b)^2 + b^2$ where $a, b = G(1,0,k), G(0,1,k)$

by 4, 5 and 6

Analogously:

13". $fib(2k+1) = G(1,0,k) \cdot G(1,1,k+1) + G(0,1,k) \cdot G(1,1,k+2)$ by 13

$= (a+b)^2 + 2b(a+b)$ <u>where</u> $a, b = G(1,0,k),\ G(0,1,k)$ by 9'

Analogously we may transform equations 16 and 17 and we get:

$G(a_0, a_1, k) = G(1,0,k) \cdot a_0 + G(0,1,k) \cdot a_1$ by 8'

$G(a_0, a_1, k+1) = G(1,0,k) \cdot G(a_0, a_1, 1) + G(0,1,k) \cdot G(a_0, a_1, 2)$ by 9'

$= G(1,0,k) \cdot a_1 + G(0,1,k) \cdot (a_0+a_1)$ by 4, 5 and 6

$G(a_0, a_1, k+2) = G(a_0, a_1, k+1) + G(a_0, a_1, k)$ by 6

$= G(1,0,k) \cdot a_1 + G(0,1,k) \cdot (a_0+a_1) + G(a_0, a_1, k)$

by above equation for $G(a_0, a_1, k+1)$

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

$+ G(1,0,k)\ G(a_0, a_1, 0) + G(0,1,k)\ G(a_0, a_1, 1)$ by 9'

$= G(1,0,k)\ (a_0+a_1) + G(0,1,k)\ (a_0+2a_1)$ by 4 and 5

Therefore we get the following program P2.1 from Program P2:

<u>Program P2.1</u>

10. $fib(0) = 1$

11. $fib(1) = 1$

12". $fib(2k) = (a+b)^2+b^2$ <u>where</u> $a, b = G(1,0,k),\ G(0,1,k)$

13". $fib(2k+1) = (a+b)^2+2b(a+b)$ <u>where</u> $a, b = G(1,0,k),\ G(0,1,k)$

14. $G(a_0, a_1, 0) = a_0$

15. $G(a_0, a_1, 1) = a_1$

16'. $G(a_0, a_1, 2k) = G(1,0,k) \cdot G(a_0, a_1, k) + G(0,1,k) \cdot G(a_0, a_1, k+1)$ by 9'

$= p^2a_0+2a_1pq+(a_0+a_1)q^2$

<u>where</u> $p, q = G(1,0,k), G(0,1,k)$

17′. $G(a_0, a_1, 2k+1) = G(1,0,k) \cdot G(a_0, a_1, k+1) + G(0,1,k) \cdot G(a_0, a_1, k+2)$ by 9′

$= p^2 a_1 + 2(a_0+a_1)pq + (a_0+2a_1)q^2$

<u>where</u> $p, q = G(1,0,k), G(0,1,k)$

Program P2.1 still suffers from the presence of redundant computations (see fig. 1-3).

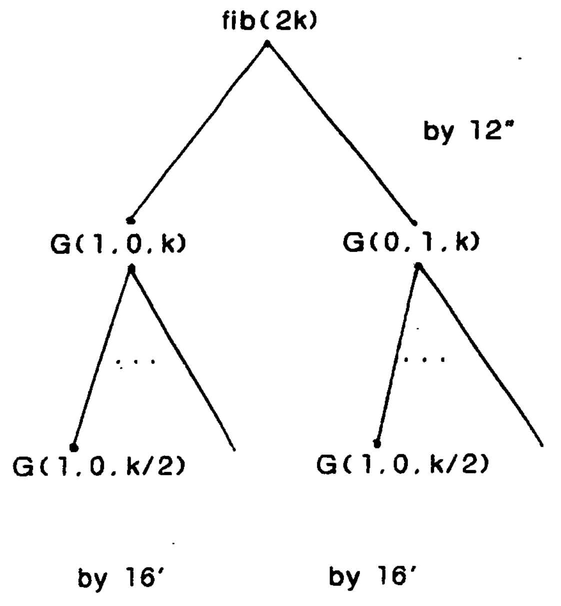


Figure 1-3: Redundant computations in program P2.1: G(1,0,k/2) occurs twice in evaluating fib(2k). (Suppose k even.)

They are caused by the functions G(1,0,k) and G(0,1,k) and therefore we will use the "tupling strategy" defining the following auxiliary function r(k):

23. $r(k) = \langle G(1,0,k), G(0,1,k) \rangle$ (<u>TUPLING EUREKA</u>)

24. $r(0) = \langle 1, 0 \rangle$

25. $r(2k) = \langle G(1,0,2k), G(0,1,2k) \rangle$

$= \langle p^2+q^2, 2pq+q^2 \rangle$ where $\langle p,q \rangle = r(k)$ by 16′

26. $r(2k+1) = \langle G(1,0,2k+1),\ G(0,1,2k+1) \rangle$

$= \langle 2pq+q^2,\ (p+q)^2+q^2 \rangle$ where $\langle p,q \rangle = r(k)$ by 17′

Therefore we can transform program P2.1 into program P3.1 where no redundant computations occur and, as required, we tupled only two functions. The running time for P3.1 is again logarithmic.

<u>Program P3.1</u>

```
10.  fib(0) = 1
11.  fib(1) = 1
12". fib(2k) = (a+b)^2+b^2               where <a,b> = r(k)
13". fib(2k+1) = (a+b)^2 + 2b(a+b)       where <a,b> = r(k)

24.  r(0) = <1,0>
25.  r(2k) = <a^2+b^2, 2ab+b^2>          where <a,b> = r(k)
26.  r(2k+1) = <2ab+b^2, (a+b)^2+b^2>    where <a,b> = r(k)
```

The straightforward correspondence between program P3.1 and the matrix exponentiation method is stated by the following fact:

<u>Fact 3</u> Let M be $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$, M^k be $\begin{bmatrix} c & b \\ d & a \end{bmatrix}$ and r(k) be defined as in program P3.1. $\forall k \geq 0 \quad r(k) = \langle a, b \rangle$.

<u>Proof</u> By induction on k. □

As we simplified Program P3 and we obtained program P4, we can simplify program P3.1 also, by using once more the efficiency equation 9′. For $n \geq 0$, we have:

7′. $fib(n) = G(1,1,n)$ by 7

$= G(1,0,n) \cdot G(1,1,0) + G(0,1,n) \cdot G(1,1,1)$ by 9'

$= G(1,0,n) + G(0,1,n)$ by 4, 5

$= a+b$ <u>where</u> $\langle a, b\rangle = r(n)$ by 23

Therefore we can get the following program P4. 1:

<u>Program P4. 1</u>

```
10.  fib(0) = 1
11.  fib(1) = 1
7'.  fib(n+2) = a+b       where <a, b> = r(n+2)    for n≥0
24.  r(0) = <1, 0>
25.  r(2k) = <a²+b², 2ab+b²>            where <a, b> = r(k)
26.  r(2k+1) = <2ab+b², (a+b)²+b²>      where <a, b) = r(k)
```

Notice that program P4. 1 has less equations than program P3. 1, but it determines exactly the same sequence of computations, as it can be proved by induction on k.
The following fig. 1-4 summarizes the last transformation steps we have performed.

Instead of using the simplification strategy and the tupling strategy, we can get program P3. 1 directly from program P1 by discovering an efficiency equation a bit more "clever" than 9' as follows. We could have thought of expressing $G(1,0,k)$ and $G(0,1,k)$ in 9' in terms of the Fibonacci function for reducing the number of different functions to be computed. This can be done by extending the definition of the Fibonacci function so that $fib(n+2) = fib(n+1) + fib(n)$ holds for $n \geq -2$. Therefore, if we want to have $fib(0) = 1$ and $fib(1) = 1$, we must have $fib(-1) = 0$ and $fib(-2) = 1$. Thus:

27. $G(1,0,k) = fib(k-2)$ by 4, 5 and 6

<u>Program P2</u> (from 10 to 17). "Almost" logarithmic time

↓

Simplification strategy (12", 13", 16', 17')

↓

<u>Program P2.1</u> (10, 11, 12", 13", 14, 15, 16', 17'). "Almost" logarithmic time.
2 functions require common subcomputations.

↓

Tupling strategy with 2 functions (24, 25, 26)

↓

<u>Program P3.1</u> (10, 11, 12", 13", 24, 25, 26). Logarithmic time.
As the 2×2 Matrix Exponentiation Method with 2 independent values.

↓

Simplification strategy (7')

↓

<u>Program P4.1</u> (10, 11, 7', 24, 25, 26). Logarithmic time.
As Program P3.1, but fewer equations.

Figure 1-4: Alternative derivation of logarithmic running time algorithms for the Fibonacci function. (Transformation steps and programs are annotated with the corresponding equation numbers.)

28. $G(0, 1, k) = fib(k-1)$ by 4, 5 and 6

We can then rewrite the <u>EFFICIENCY EQUATION</u> as follows:

9". $G(a_0, a_1, n+k) = fib(k-2) \cdot G(a_0, a_1, n) + fib(k-1) \cdot G(a_0, a_1, n+1)$
for $n \geq 0$, $k \geq 0$

Using 9" we get the following program for computing the Fibonacci function:

10. $fib(0) = 1$

11. $fib(1) = 1$

```
29. fib(2k)    = G(1,1,2k)                                         by 7
               = fib(k-2)·fib(k) + fib(k-1)·fib(k+1)               by 9" and 7
               = (a+b)^2 + b^2      where a,b = fib(k-2),fib(k-1)   by 3
30. fib(2k+1)  = G(1,1,2k+1)                                       by 7
               = fib(k-2)·fib(k+1) + fib(k-1)·fib(k+2)             by 9" and 7
               = (a+b)^2 + 2b(a+b)  where a,b = fib(k-2),fib(k-1)   by 3
```

By equations 23, 27 and 28 it turns out that equations 29 and 30 are the same as 12" and 13". Therefore applying the tupling strategy by defining r(n) = <fib(n-2),fib(n-1)> we can get again program P3.1.
It is very interesting to notice that the cleverness of discovering 9", starting from 9' (including also the extension of the Fibonacci function definition) can be obtained via the application of a very simple strategy, i.e. the simplification one. Therefore, in some particular cases, the basic strategies are indeed very powerful and they allow us to obtain results which can be otherwise derived only by "intelligent" reasoning.

Linear recurrence relations may also be solved using the <u>generating functions method</u> [Liu 68]. Applying that method it is possible to derive an explicit formula for the nth element of the series which satisfies a given recurrence relation. That formula contains in general an exponentiation to the nth power which allows a logarithmic running time algorithm. For example, for the Fibonacci relation, fib(n) is equal to $(A^{n+1}-B^{n+1})/\sqrt{5}$, where $A = (1+\sqrt{5})/2$ and $B = (1-\sqrt{5})/2$. Therefore as far as running time is concerned, the program we obtained using the transformation technique is not less efficient, at least asymptotically.

1.5 The general case of homogeneous linear recurrence relations

In this section we deal with the general case of homogeneous linear recurrence relations with constant coefficients (whose definition will be recalled later) and we show that analogous steps to those presented for deriving an efficient algorithm for computing the Fibonacci function can be performed in that case as well.

We will follow for the general case, the transformation steps indicated in fig. 1-2 (from program P1 to program P2) and the ones indicated in fig. 1-4 (from program P2 to program P3.1): in fact the generalization eurekas and the linear combination eurekas can be applied in the same manner, so that we can derive an algorithm for evaluating any homogeneous linear recurrence relation in logarithmic time.
Therefore, as the matrix exponentiation method is a general method for evaluating linear recurrence relations, so is our transformation method. This is an interesting fact because it shows the "uniform" power of the transformation strategies with respect to a given class of problems, just as "paradigms" [Floyd 79] in the stepwise refinement technique has a uniform power for solving problems in a given class.

Let us consider a general homogeneous linear recurrence relation of order r:

4.1
$$\left[\begin{array}{lll} h(0) & = h_0 & \\ h(1) & = h_1 & \\ \quad \dots & & \\ h(r-1) & = h_{r-1} & \\ h(n) & = b_0\, h(n-r) + \dots + b_{r-1}\, h(n-1) & \text{for } n \geq r \end{array}\right.$$

We will also write $h(n) = L(h(n-r), \dots, h(n-1))$ where the polynomial

$L(x_0,\ldots,x_{r-1}) = \sum_{i=0}^{r-1} b_i\, x_i$ is a linear polynomial in the variables $x_0,\ldots,x_{r-1}$

with the constant coefficients $b_0,\ldots,b_{r-1}$.

The generalization eureka introduces the following function H (analogous to the function $G(a_0,a_1,n)$):

4.2 $$\left[\begin{array}{ll} H(h_0,\ldots,h_{r-1},0) & = h_0 \\ \quad\ldots & \\ H(h_0,\ldots,h_{r-1},r-1) & = h_{r-1} \\ H(h_0,\ldots,h_{r-1},n) & = L(H(h_0,\ldots,h_{r-1},n-r),\ldots,H(h_0,\ldots,h_{r-1},n-1)) \\ & \qquad\qquad \text{for } n \geq r \end{array}\right.$$

The general form of the linear combination eureka (analogous to equation 8') is:

4.3 $$H(h_0,\ldots,h_{r-1},n) = H(1,0,\ldots,0,n)\cdot h_0 + \ldots + H(0,0,\ldots,1,n)\cdot h_{r-1}$$

$$= \sum_{i=0}^{r-1} H_i(n)\, h_i$$

where we denoted by $H_i(n)$ the term $H(\underbrace{0,\ldots,0}_{i},1,\underbrace{0,\ldots,0}_{r-1-i},n)$

for $i=0,\ldots,r-1$.

If we take $h_0,\ldots,h_{r-1}$ to be equal to r consecutive values of $H(h_0,\ldots,h_{r-1},n)$ for $n=k,k+1,\ldots,k+r-1$, then from the linear combination eureka, we can derive the following general form of the efficiency equation, as we did in deriving the equation 9':

4.4 $$H(h_0,\ldots,h_{r-1},m+k) = H(1,0,\ldots,0,m)\cdot H(h_0,\ldots,h_{r-1},k) + \ldots$$
$$+ H(0,0,\ldots,1,m)\cdot H(h_0,\ldots,h_{r-1},k+r-1)$$

$$= \sum_{i=0}^{r-1} H_i(m)\ H(h_0, \ldots, h_{r-1}, k+i)$$

which can be proved by induction on k using equation 4.3.

Therefore we have the following equations:

4.4.1 $h(2k) = H(_0, \ldots, h_{r-1}, k+k) = \sum_{i=0}^{r-1} H_i(k)\ H(h_0, \ldots, h_{r-1}, k+i)$

4.4.2 $h(2k+1) = H(h_0, \ldots, h_{r-1}, k+k+1) = \sum_{i=0}^{r-1} H_i(k)\ H(h_0, \ldots, h_{r-1}, k+i+1)$

which are analogous to the equations 12 and 13 of Section 1.3.

For avoiding redundant computations in computing h(2k) and h(2k+1) we need to apply the tupling strategy (as we did in deriving equation 18) to the following 2r functions: $H_0(k)$, ..., $H_{r-1}(k)$, $H(h_0, \ldots, h_{r-1}, k)$, ..., $H(h_0, \ldots, h_{r-1}, k+r-1)$. Out of these 2r values, only r are independent. In fact, as for the equations 16′ and 17′, we can express $H(h_0, \ldots, h_{r-1}, k+i)$ for $i=0, \ldots, r$ in terms of the $H_j(k)$'s for $0 \leq j \leq r-1$, using the equation 4.4. We have:

4.5 $H(h_0, \ldots, h_{r-1}, k+i) = \sum_{j=0}^{r-1} H_j(k)\ H(h_0, \ldots, h_{r-1}, i+j)$ for $0 \leq i \leq r$ by 4.4

Therefore we can rewrite the equations 4.4, 4.4.1 and 4.4.2 as follows:

4.4′ $H(h_0, \ldots, h_{r-1}, m+k) = \sum_{i=0}^{r-1} H_i(m)\ [\sum_{j=0}^{r-1} H_j(k)\ H(h_0, \ldots, h_{r-1}, i+j)\ ]$

4.4.1′ $h(2k) = \sum_{i=0}^{r-1} H_i(k)\ [\sum_{j=0}^{r-1} H_j(k)\ H(h_0, \ldots, h_{r-1}, i+j)]$

$$= \sum_{i=0}^{r=0} H_i(k)\ [\sum_{j=0}^{r-1} H_j(k) h_{i+j}\ ]$$

4.4.2′ $h(2k+1) = \sum_{i=0}^{r-1} H_i(k)\ [\sum_{j=0}^{r-1} H_j(k)\ H(h_0, \ldots, h_{r-1}, i+j+1)]$

$$= \sum_{i=0}^{r-1} H_i(k) \; [\sum_{j=0}^{r-1} H_j(k) \; h_{i+j+1}]$$

where h_i stands for $h(i)$ for $0 \leq i \leq 2r-1$.

The h_i's for $0 \leq i \leq 2r-1$ can be computed once and for all, using the equations 4.1. Then we can apply the tupling strategy, as we did in equation 23, for computing the r independent values by defining the following function $t(k)$:

4.6 $t(k) = \langle H(1,0,\ldots,0,k),\ H(0,1,\ldots,0,k),\ \ldots,\ H(0,0,\ldots,1,k)\rangle$

$= \langle H_0(k),\ H_1(k),\ \ldots,\ H_{r-1}(k)\rangle$ (<u>TUPLING EUREKA</u>)

Now we can derive the equations for computing $t(k)$ using equation 4.6 and the efficiency equation 4.4', in much the same way as we derived equations 24, 25 and 26.

4.7 $t(0) = \langle H_0(0), H_1(0), \ldots, H_{r-1}(0)\rangle$ by 4.6

$= \langle 1, 0, \ldots, 0\rangle$ by 4.2

...

4.8 $t(r-1) = \langle H_0(r-1), H_1(r-1), \ldots, H_{r-1}(r-1)\rangle$ by 4.6

$= \langle 0, 0, \ldots, 1\rangle$ by 4.2

4.9 $t(2k) = \langle H_0(k+k), \ldots, H_{r-1}(k+k)\rangle$ by 4.6

$$= \langle \sum_{i=0}^{r-1} H_i(k) \; [\sum_{j=0}^{r-1} H_j(k) H_0(i+j)], \ldots,$$

$$\sum_{i=0}^{r-1} H_i(k) [\sum_{j=0}^{r-1} H_j(k) \; H_{r-1}(i+j)] \rangle$$ by 4.4'

4.10 $t(2k+1) = \langle H_0(k+k+1), \ldots, H_{r-1}(k+k+1)\rangle$ by 4.6

$$= < \sum_{i=0}^{r-1} H_i(k) \cdot H_0(k+1+i), \ldots . \sum_{i=0}^{r-1} H_i(k) \cdot H_{r-1}(k+1+i) > \qquad \text{by } 4.4$$

$$= < \sum_{i=0}^{r-1} H_i(k) \; [\sum_{j=0}^{r-1} H_j(k) \;\; H_0(i+j+1)], \ldots .$$

$$\sum_{i=0}^{r-1} H_i(k) [\sum_{j=0}^{r-1} H_j(k) \;\; H_{r-1}(i+j+1)] > \qquad \text{by } 4.5$$

We have now completed the task of deriving a general program (given below) for evaluating any homogeneous linear recurrence relation of order r in logarithmic time.

Recursive program for evaluating linear recurrence relations

(as given by equations 4.1)

$h(0) = h_0$ by 4.1

. . .

$h(r-1) = h_{r-1}$ by 4.1

$$h(2k) = \sum_{i=0}^{r-1} a_i \sum_{j=0}^{r-1} a_j\, h(i+j)$$

where $\langle a_0, \ldots, a_{r-1}\rangle = t(k)$ by 4.4.1′ and 4.6

$$h(2k+1) = \sum_{i=0}^{r-1} a_i \sum_{j=0}^{r-1} a_j\, h(i+j+1)$$

where $\langle a_0, \ldots, a_{r-1}\rangle = t(k)$ by 4.4.2′ and 4.6

$t(0) = \langle 1, 0, \ldots, 0\rangle$ by 4.7

. . .

$t(r-1) = \langle 0, 0, \ldots, 1\rangle$ by 4.8

$$t(2k) = \langle \sum_{i=0}^{r-1} a_i \,[\, \sum_{j=0}^{r-1} a_j \cdot \pi 0(t(i+j))\,] \,\ldots\ldots\, \sum_{i=0}^{r-1} a_i \,[\, \sum_{j=0}^{r-1} a_j \cdot \pi r-1(t(i+j))\,] \rangle$$

where $\langle a_0, \ldots, a_{r-1}\rangle = t(k)$ by 4.9

$$t(2k+1) = \langle \sum_{i=0}^{r-1} a_i \,[\, \sum_{j=0}^{r-1} a_j \cdot \pi 0(t(i+j+1))\,] \,\ldots\ldots\, \sum_{i=0}^{r-1} a_i \,[\, \sum_{j=0}^{r-1} a_j \cdot \pi r-1(t(i+j+1))\,] \rangle$$

where $\langle a_0, \ldots, a_{r-1}\rangle = t(k)$ by 4.10

Remarks

1. The precomputation of the $h(i)$'s and the $t(i)$'s for $r \leq i \leq 2r-1$ is required and it can be done using equations 4.1 and definitions 4.2 and 4.6.
2. $\pi i(\langle a_0, \ldots, a_{r-1}\rangle) = a_i$ for $i=0, \ldots, r-1$. □

As a consequence of the program we have just derived one can say that, in general, any homogeneous linear recurrence relation of order r (in which a generic value depends on r preceding values) can be evaluated in logarithmic time if we compute r values simultaneously (see equation 4.6).

Example 2

Given the following linear recurrence relation:

$$\begin{cases} p(0) = 1 \\ p(n) = 2 \cdot p(n-1) \quad \text{for } n>0 \end{cases}$$

we have r=1 and therefore all summation operators occurring in the general program vanish.

Moreover, since r=2r−1=1 we have to compute only the value of p(1), which is equal to 2.

We have: $p(2k) = a_0 \cdot a_0 \cdot p(0) = a_0^2$, $p(2k+1) = a_0 \cdot a_0 \cdot p(1) = 2a_0^2$,

$t(k) = \langle a_0 \rangle$; $t(0) = \langle 1 \rangle$; $t(2k) = \langle a_0^2 \rangle$; $t(2k+1) = \langle 2a_0^2 \rangle$

and therefore we get (by forgetting the unit-tupling operator):

$$\begin{array}{lll} p(0) & = 1 & \\ p(2k) & = a_0^2 & \underline{\text{where}}\ a_0 = t(k) \\ p(2k+1) & = 2a_0^2 & \underline{\text{where}}\ a_0 = t(k) \\ t(0) & = 1 & \\ t(2k) & = a_0^2 & \underline{\text{where}}\ a_0 = t(k) \\ t(2k+1) & = 2a_0^2 & \underline{\text{where}}\ a_0 = t(k) \end{array}$$

Since p(k) and t(k) obey the same equations, p(k)=t(k). Thus the program we obtained can be further simplified as follows:

$$\begin{cases} p(0) = 1 \\ p(2k) = p^2(k) \\ p(2k+1) = 2 \cdot p^2(k) \end{cases}$$ □

Example 3

Let us consider the following recurrence relation:

$$\begin{cases} d(0) = 1 \\ d(1) = 2 \\ d(2) = 0 \\ d(n) = d(n-1) + 2d(n-3) \quad \text{for } n>2 \end{cases}$$

We have: $r=3$, $2r-1=5$. Thus we have to compute $d(i)$ for $i=3,4,5$ and we get: $d(3)=2$, $d(4)=d(5)=6$. We also have:
$t(0)=\langle 1,0,0\rangle$, $t(1)=\langle 0,1,0\rangle$, $t(2)=\langle 0,0,1\rangle$, $t(3)=\langle 2,0,1\rangle$, $t(4)=\langle 2,2,1\rangle$, $t(5)=\langle 2,2,3\rangle$ because for $i=3,4,5$ and $j=0,1,2$: $\pi j(t(i)) = \pi j(t(i-1)) + 2\cdot\pi j(t(i-3))$. (In fact the components of $t(i)$ satisfy the same recurrence relation given for $d(n)$, as one can see from equation 4.2 and 4.6). We can then derive:

$$d(2k) = a_0\cdot(a_0d(0) + a_1d(1) + a_2d(2)) + a_1(a_0d(1) + a_1d(2) + a_2d(3)) + a_2(a_0d(2) + a_1d(3) + a_2d(4))$$
$$= a_0^2 + 4a_0a_1 + 4a_1a_2 + 6a_2^2 \qquad \underline{\text{where}}\ \langle a_0, a_1, a_2\rangle = t(k)$$

and analogously:

$$d(2k+1) = 2a_0^2 + 2a_1^2 + 6a_2^2 + 4a_0a_2 + 12\,a_1a_2 \qquad \underline{\text{where}}\ \langle a_0, a_1, a_2\rangle = t(k)$$

We also have:

$$t(2k) = \langle a_0^2 + 4a_1a_2 + 2a_2^2,\ 2a_0a_1 + 2a_2^2,\ a_1^2 + 2a_2^2 + 2a_0a_2 + 2a_1a_2\rangle \qquad \underline{\text{where}}\ \langle a_0, a_1, a_2\rangle = t(k)$$

$$t(2k+1) = \langle 4a_0a_2+2a_1^2+4a_1a_2+2a_2^2,\ a_0^2+4a_1a_2+2a_2^2,\ 2a_0a_1+2a_0a_2+2a_1a_2+a_1^2+3a_2^2\rangle \qquad \underline{\text{where}}\ \langle a_0, a_1, a_2\rangle = t(k)$$

The resulting program is the following:

$$\begin{cases} d(0) = 1 \\ d(1) = 2 \\ d(2) = 0 \\ d(2k) = a^2 + 4ab + 4bc + 6c^2 \qquad \underline{\text{where}}\ \langle a, b, c\rangle = t(k) \end{cases}$$

$$
\begin{array}{lll}
d(2k+1) & = 2a^2 + 2b^2 + 6c^2 + 4ac + 12bc & \underline{\text{where}}\ \langle a,b,c\rangle = t(k) \\
t(0) & = \langle 1,0,0\rangle & \\
t(2k) & = \langle a^2 + 4bc + 2c^2,\ 2ab + 2c^2,\ b^2 + c^2 + 2ac + 2bc\rangle & \\
 & & \underline{\text{where}}\ \langle a,b,c\rangle = t(k) \\
t(2k+1) & = \langle 4ac + 2b^2 + 4bc + 2c^2,\ a^2 + 4bc + 2c^2, 2ab + 2ac + 2bc + b^2 + 3c^2\rangle & \\
 & & \underline{\text{where}}\ \langle a,b,c\rangle = t(k)
\end{array}
$$

Notice that once we computed the expressions for t(2k) and t(2k+1), since they hold for any $k \geq 0$, we can discard from the program the equations for t(1),....,t(r-1). This fact is a general property and it could also be applied for the d's values, so that we could have erased the equations for d(1) and d(2). □

At the end of this section we would like to derive a for-loop program equivalent to the general recursive program we have given. This for-loop program will be a generalization of the one presented in [Wilson and Shortt 80], which had to be proved correct at the expense of many theorems and long proofs and worked only for Fibonacci recurrence relations.

Unfortunately the recursion which occurs in the given general recursive program is not a tail-recursion and therefore its translation into a for-loop program is not immediate.

Nevertheless such a translation is possible [Walker and Strong 72] because of the equivalence between the following recursive schema S and the flowchart F in fig. 1-5. Suppose we are given the following recursive definition of the function g(k):

$$g(k) = \begin{cases} e & \underline{\text{if}}\ \text{zero}(k) \\ a(g(b(k))) & \underline{\text{if}}\ \text{even}(k) \\ c(g(d(k))) & \underline{\text{if}}\ \text{odd}(k) \end{cases} \qquad \text{(Schema S)}$$

where zero(k), even(k) and odd(k) test whether k is zero, even or odd, respectively.
It can easily be proved by induction on the depth of recursion that g(k) can be computed by the flowchart program (with one stack) given in fig. 1-5.

We can apply that equivalence result of the schema S and the flowchart F for computing the recursively defined function t(k) which occurs in our program. Since in that program the b and d operations correspond to integer divisions by 2, the content of the stack at the label L in the flowchart F is the binary expansion of k. We assume that the empty stack represents the binary expansion of 0.

Therefore we can obtain the following <u>for-loop</u> program for evaluating in logarithmic time any homogeneous linear recurrence relation of order r, as given by the equations 4.1.

(Flowchart F)

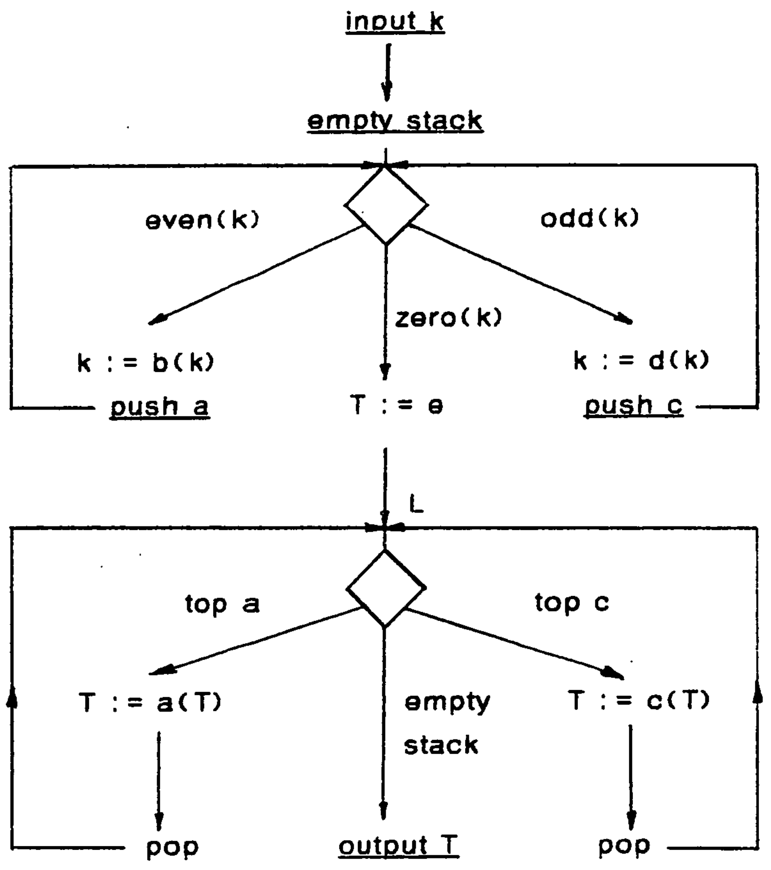


Figure 1-5: The flowchart F corresponding to the program schema S.

For-loop Program for Evaluating Linear Recurrence Relations

(as given by equations 4.1)

```
{ $b \geq 0$; $h(0)=h_0$; ...; $h(r-1)=h_{r-1}$; $h(n)=L(h(n-r),\ldots,h(n-1))$ for $n \geq r$ }

if n=0 then B(0), ℓ:=0,0
        else begin m,i,ℓ:=n,0,⌊log n⌋;                                              (1)
            while i≤ℓ do B(i),m,i:=rem(m/2),m/2,i+1 od end;

    { $n = \sum_{i=0}^{\ell} B(i)\cdot 2^i$  and  $\ell = \lfloor \log n \rfloor$ }

t(0)   := <1,0,...,0>;
  ...
t(r-1) := <0,0,...,1>;
for i=r to 2r-1 do h(i) := L(h(i-r),...,h(i-1));
                   t(i) :=<L(π0(t(i-r)),...,π0(t(i-1))),....,                       (2)
                           L(πr-1(t(i-r)),...,πr-1(t(i-1)))>
              od;
T:=t(0);
for p=ℓ downto 1 do
        if B(p)=0 then

        T:=< $\sum_{i=0}^{r-1}$ πi(T) [ $\sum_{j=0}^{r-1}$ πj(T)·π0(t(i+j))] .....,
             $\sum_{i=0}^{r-1}$ πi(T) [ $\sum_{j=0}^{r-1}$ πj(T)·πr-1(t(i+j))] >

        else                                                                        (3)

        T:=< $\sum_{i=0}^{r-1}$ πi(T) [ $\sum_{j=0}^{r-1}$ πj(T)·π0(t(i+j+1))] .....,
             $\sum_{i=0}^{r-1}$ πi(T) [ $\sum_{j=0}^{r-1}$ πj(T)·πr-1(t(i+j+1))] >
            od;
if B(0)=0 then H:= $\sum_{i=0}^{r-1} \pi i(T)\ [\sum_{j=0}^{r-1} \pi j(T)\cdot h(i+j))]$
          else H:= $\sum_{i=0}^{r-1} \pi i(T)\ [\sum_{j=0}^{r-1} \pi j(T)\cdot h(i+j+1))]$ ;
        { $H = h(n)$ }
```

(1) The binary expansion of n is stored in the array B(i) for $i=\ell,\dots,0$

(2) Computation of the values $h(r),\dots,h(2r-1),t(r),\dots,t(2r-1)$.

(3) Computation of T. $T = t(n/2)$ if n is even. $T = t((n-1)/2)$ if n is odd. □

The method presented here allows also a fast evaluation of non-homogeneous linear recurrence relations, because in that case the result is the sum of two terms: one corresponding to the associated homogeneous relation, which we can evaluate in logarithmic time, and the other one being a particular solution of the given non-homogeneous relation [Liu 68]. The method can also be used when we have a set of mutual depending linear recurrence relations (with constant coefficients). In fact it is always possible to reduce such a set, via substitutions, to a set of independent relations [Liu 68] and solve them simultaneously using the tupling strategy.

Furthermore, one can apply the general algorithms we have given in this section to recurrence relations holding in other algebraic structures, different from the one of the integers with usual addition and multiplication for which we presented it. This is what is stated in the following Fact 4. Let us first recall a few basic definitions.

A semiring is an algebra $(S,+,\cdot)$, where S is a set and + and $\cdot$ are two binary operations on S, such that:

i) $+$ is associative and commutative.

ii) $\cdot$ is associative.

iii) $\cdot$ distributes over $+$ on both sides, i.e. $a\cdot(b+c)=(a\cdot b)+(a\cdot c)$ and $(b+c)\cdot a=(b\cdot a)+(c\cdot a)$.

An element i of S is an identity for the operation op iff

$\forall x \in S \quad (x \text{ op } i) = (i \text{ op } x) = x$. (We used an infix notation for op).

An element a of S is an absorbent element for the operation op iff

$\forall x \in S \quad (x \text{ op } a) = (a \text{ op } x) = a$.

Notice that we could have written "the identity for +" instead of "an identity for +" because if there is an identity then it is unique. In fact, suppose there are 2 identities, 0 and $0'$ say. By definition we have:

$\forall x \in S \quad 0+x = x+0 = x$ and $\forall x \in S \quad 0'+x = x+0' = x$.

Instantiating the first for $x=0'$ and the second for $x=0$ we get $0+0'=0=0'$. Analogously one could show that the absorbent element for an operation is unique (if it exists).

A semiring $(S,+,\cdot)$ with identity elements 1 for $\cdot$ and 0 for $+$ is also denoted by $(S,+,\cdot,0,1)$.

Fact 4. The recursive program and the for-loop program we have given for the evaluation of linear recurrence relations with constant coefficients are valid in any semiring structure $(S,+,\cdot)$, which has

i) an identity element for $\cdot$ (let us call it 1), and

ii) an absorbent element for $\cdot$ (let us call it 0) which is an identity for $+$.

Proof. The various steps we made for going from the given equations 4.1 to the final programs are based on the efficiency equation 4.4. We have to show that equation 4.4 holds in any semiring structure satisfying the given hypotheses. We will prove that property by induction on $\langle k, m\rangle$.

For k=0 and m=0:

$$H(h_0,\dots,h_{r-1},0) = \sum_{i=0}^{r-1} H_i(0)\ H(h_0,\dots,h_{r-1},i)$$

$$= 1\cdot H(h_0,\dots,h_{r-1},0) + 0\cdot H(h_0,\dots,h_{r-1},1) + \dots$$

$$+ 0\cdot H(h_0,\dots,h_{r-1},r-1)$$

$$= H(h_0,\dots,h_{r-1},0)$$

Analogous equalities can be proved for k=0 and m=1,...,r−1 as follows. Suppose now that equation 4.4. is valid for k=0 and any $m \leqslant p$ with $p \geqslant r-1$. Let us show that it holds for m=p+1.

$$H(h_0,\dots,h_{r-1},p+1) = b_0\cdot H(h_0,\dots,h_{r-1},p+1-r)+\dots+b_{r-1}\cdot H(h_0,\dots,h_{r-1},p)$$

by 4.1

$$= b_0\cdot \sum_{i=0}^{r-1} H_i(p+1-r)\ H(h_0,\dots,h_{r-1},i)+\dots$$

$$+ b_{r-1}\cdot \sum_{i=0}^{r-1} H_i(p)\ H(h_0,\dots,h_{r-1},i)$$

by inductive hypothesis

$$= \sum_{i=0}^{r-1} H_i(p+1)\ H(h_0,\dots,h_{r-1},i)$$

by hypotheses on the semiring structure and equations 4.1.

Therefore equation 4.4 holds in a semiring structure for k=0 and any $m \geqslant 0$.

Now suppose that equation 4.4 holds for any $m \geqslant 0$ and any $k \leqslant p$ with $p \geqslant r-1$. Let us show that it holds for k=p+1.

$$H(h_0,\dots,h_{r-1},m+p+1) = b_0\cdot H(h_0,\dots,h_{r-1},m+p+1-r) + \dots$$

$$+ b_{r-1}\cdot H(h_0,\dots,h_{r-1},m+p)$$

by 4.1

$$= b_0\cdot \left[\sum_{i=0}^{r-1} H_i(m)\ H(h_0,\dots,h_{r-1},p+1-r+i)\right] + \dots$$

$$+ b_{r-1} \cdot \left[\sum_{i=0}^{r-1} H_i(m)\ H(h_0, \ldots, h_{r-1}, p+i) \right]$$

by inductive hypothesis

$$= \sum_{i=0}^{r-1} H_i(m)\ H(h_0, \ldots, h_{r-1}, p+1+i)$$

by hypotheses on the semiring structure and equations 4.1. □

The programs we have given can be used, for instance, in the semiring of truthvalues $B = (\{\underline{true}, \underline{false}\}, \vee, \wedge, \underline{false}, \underline{true})$, and the semiring M of the real numbers R with $+\infty$ and the usual minimum and addition operations $M = (R \cup \{+\infty\}, \min, +, +\infty, 0)$, and the subsemiring of M obtained by considering nonnegative real numbers only.

Example 4. Let us consider the following recurrence relation in M:

$$\begin{cases} d(0) = 0 \\ d(1) = 2 \\ d(2) = +\infty \\ d(n) = \min(d(n-1), d(n-3)-2) \quad \text{for } n>2 \end{cases}$$

We get the following logarithmic running time program for computing d(n):

$$d(0) = 0$$

$$d(2k) = \min(a + \min(a+0, b+2, c+\infty), b + \min(a+2, b+\infty, c-2), c + \min(a+\infty, b-2, c-2))$$

$$= \min(2a, a+b+2, b+c-2, 2c-2) \qquad \underline{\text{where}}\ \langle a, b, c\rangle = t(k)$$

and analogously:

$$d(2k+1) = \min(2a+2, a+c-2, 2b-2, b+c-2, 2c-2)\ \underline{\text{where}}\ \langle a, b, c\rangle = t(k)$$

$$t(0) = \langle 0, +\infty, +\infty\rangle$$

t(2k) = <min(2a, b+c−2, 2c−2), min(a+b, 2c−2), min(a+c, 2b, b+c, 2c)>

where <a, b, c>=t(k)

t(2k+1) = <min(a+c−2, 2b−2, b+c−2, 2c−2), min(2a, b+c−2, 2c−2),

min(a+b, a+c, 2b, b+c, 2c−2)> where <a, b, c>=t(k)

□

1.6 Comparing the transformation technique and the stepwise refinement technique

In this section we would like to compare the transformation technique [Burstall and Darlington 77] and the stepwise refinement technique [Wirth 71] making some general comments and discussing their features when these techniques are applied for writing programs which evaluate linear recurrence relations.

Both techniques indeed have their merits. For computing the solution of recurrence relations the stepwise refinement method uses structured concepts, like matrices and binary representations, but it seems to require more creative thoughts than the transformation technique. In particular in the stepwise refinement approach, the programmer has to be familiar with the fact that given two matrices:

$$A = \begin{bmatrix} a+b & b \\ b & a \end{bmatrix} \text{ and } A' = \begin{bmatrix} c+d & d \\ d & c \end{bmatrix} \text{ for some a, b, c and d}$$

we have that: $A \cdot A' = A' \cdot A = \begin{bmatrix} e+f & f \\ f & e \end{bmatrix}$,

where $e = ac+bd$ and $f = d(a+b)+bc$.

The knowledge of that fact is an a priori requirement for devising a suitable loop structure and, if the program is not organized using such a loop, it seems very hard, or even impossible, to achieve the desired logarithmic efficiency [Gries and Levin 80, Urbanek 80, Wilson and Shortt 80, Pettorossi 80a].

This situation fits very well with the analogy given in [Burstall and Feather

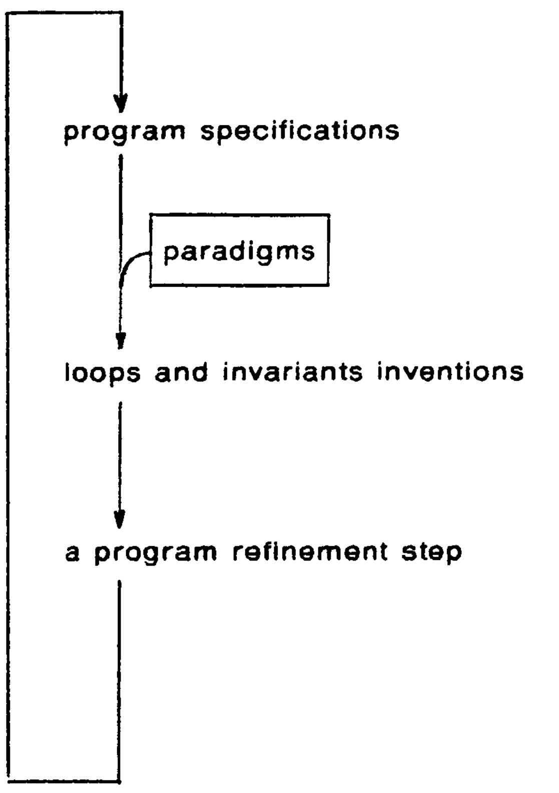
77], concerning the stepwise refinement technique and the transformation technique, which are respectively compared with sculpture and plasticine modelling. In sculpture, we all know that once the outline of a statue is carved in a rock, there is very little possibility for changing the basic idea of the artistic composition. In our program, using the stepwise refinement technique, the "outline of the statue" consists, in the outermost loop for performing the successive squarings of the matrix:

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

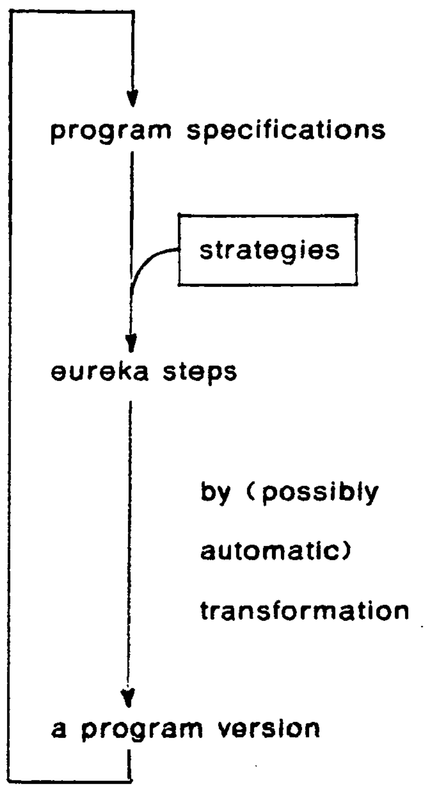
That loop has to be controlled by the binary representation of the value n for which we would like to compute the solution of the recurrence relation (see [Wilson and Shortt 80]). Different approches to the outermost loop could have made a logarithmic running time almost impossible to be achieved in a simple way.

On the contrary, we showed in the previous sections that using the transformation technique a very efficient program can be obtained, without a deep knowledge of recurrence relations or matrix theory. The transformation technique, via its strategies, led us to the desired algorithm without great effort.

Obviously, in transforming programs we have to make some clever steps, called "eurekas" [Burstall and Darlington 77], which are not always easy to make. However, we have strategies, such as composition, tupling, generalization, simplification, etc. [Burstall and Feather 77], which help us to produce eurekas and allow us to improve program performances. These strategies play the analogous role of the programming paradigms, which help the programmer in choosing the suitable loop structures, when using the stepwise refinement technique [Floyd 79]. The situation of program construction using the two techniques may be depicted as in fig. 1-6.



(i) The Stepwise Refinement Technique



(ii) The Transformation Technique

Figure 1-6: The Program Construction Cycle according to the Stepwise Refinement Technique and the Program Transformation Technique.

We are not claiming that the transformation method is better than the stepwise refinement method, but we would like to stress that in our examples some very interesting features of the program transformation technique with respect to the stepwise technique have been exploited. In particular we have seen that, using standard strategies, it was relatively simple to derive the various "eureka" definitions for our programs, while it seems difficult to directly formulate the complex loop invariant of an iterative program for solving recurrence relations in logarithmic time [Wilson and Shortt 80]. In particular the notion of matrix as grouping of values for a fast evaluation of recurrence relations has been derived by simple application of the tupling strategy, while that notion is in some sense "primitive" (i.e. not derivable) in the construction

of an efficient program using the stepwise refinement technique. It is interesting to notice that, in deriving our programs, the simultaneous evaluation of some expressions for avoiding redundant computations, automatically achieved the desired logarithmic performance. One might say that it was the efficiency requirement itself which, via the tupling strategy application, "rediscovered" for us the notion of matrix. Indeed it also "rediscovered" the notion of symmetric matrix, as we showed towards the end of section 1.4.
Another possible advantage of the program transformation approach is the fact that it can easily be used in the functional style of programming, advocated in [Backus 78].

In the last part of this section we would like also to stress the strong connection between the task of devising loops and the one of finding eurekas, already mentioned in [Burstall and Feather 77]. Related work in this direction has been done in [Broy and Krieg-Brückner 80].
The "eureka" steps are "intelligent" steps as the inventions of loops invariants are. For these inventions, creativity and programming experience play a crucial role. For example, the "linear combination eureka" has been as crucial as the idea successive squaring of matrices is for the stepwise refinement technique, but we think that such a eureka is, in some sense, less difficult.
One can often observe an interesting correspondence between the eureka steps and the loops invariants. This correspondence is not claimed in general, but it is valid when the recursive program has a "simple" kind of recursion (which includes tail-recursion) easily translatable into a loop. In order to clarify the ideas we will now give the following two examples.

Example 5 .

In [Burstall and Darlington 77] the following program for computing the Fibonacci function in linear time is given:

```
fib(0) = 1
fib(1) = 1
fib(n) = u+v        where <u,v> = g(n-1)
g(1) = <1,1>
g(n) = <u+v,u>      where <u,v> = g(n-1)
```

It is based on the eureka step which consists of defining the function $g(n) = \langle fib(n), fib(n-1)\rangle$ for $n \geq 1$. On the other hand, the iterative program for computing the same function in linear time may be written as follows:

```
{ n≥0 }
i,u,v := -1,1,0;
{ <u,v> = <fib(i+1),fib(i)> and i≤n-1 }
while i<n-1 do
          t:=u+v;  v:=u;   u:=t;  i:=i+1
          od
{ n≥0 and u = fib(n) }
```

One can easily see that the two auxiliary variables u and v of the iterative program are equal to the projections of the function g defined in the eureka step. We have in fact: $g(i+1) = \langle u, v\rangle$. □

Example 6 (From section 1.4)

Since $r(n) = \langle G(1,0,n), G(0,1,n)\rangle$ (see equation 23) using equations 27 and 28 we have: $r(n) = \langle fib(n-2), fib(n-1)\rangle$. Therefore we can transform program P3.1 into the following:

Program P3.2

```
fib(n)   = $\pi 1(r(n+1))$
r(0)     = $\langle 1,0\rangle$
r(2k)    = $\langle a^2 + b^2, 2ab + b^2\rangle$                where $\langle a,b\rangle = r(k)$
r(2k+1)  = $\langle 2ab + b^2, (a+b)^2 + b^2\rangle$       where $\langle a,b\rangle = r(k)$
where $\pi 1(\langle e0, e1\rangle) = e1$.
```

For this recursive program we can derive the following iterative version:

```
  { $n \geq 0$ and $\ell = \lfloor \log(n+1) \rfloor$ and $n+1 = \sum_{i=0}^{\ell} B(i) \cdot 2^i$ }
  u,v := 1,0;
  p := $\ell$;
I = { $\langle u,v\rangle = \langle fib(m-2), fib(m-1)\rangle =$
    $= \langle fib^2(k-2)+fib^2(k-1), 2fib(k-2)fib(k-1)+fib^2(k-1)\rangle$                if m=2k
    $= \langle 2fib(k-2)fib(k-1)+fib^2(k-1), (fib(k-2)+fib(k-1))^2+fib^2(k-1)\rangle$   if m=2k+1

  where if $\ell$=p then m=0 else $m = \sum_{i=p+1}^{\ell} B(i) \cdot 2^{i-p-1}$ }

  while p>0 do if B(p) = 0 then u,v := $u^2+v^2, 2uv+v^2$
                                else u,v := $2uv+v^2, (u+v)^2+v^2$;
               p:=p-1
          od
        { $n \geq 0$ and v = fib(n) }
```

This iterative program can easily be derived from the general for-loop program (actually only from the fragment for computing T) given in section 1.5 by recalling that:

i) T = t(n/2) if n is even and T = t((n-1)/2) if n is odd; and

ii) in order to compute fib(n) we need only to compute the second projection of t(n+1), because in the general for-loop program the function t plays the role of the function r in program P3.2 and in our case

$r(n+1) = \langle fib(n-1), fib(n) \rangle$. Therefore $t(n+1)$ can be obtained as the value of the variable T in the general program assuming that the array B contains the binary expansion of n+1 and the for-loop is performed for all digits in B.

To complete the derivation of our iterative program from the general one, it is enough to consider that in our case T is $\langle u, v \rangle$ and $t(0) = \langle 1, 0 \rangle$, $t(1) = \langle 0, 1 \rangle$, $t(2) = \langle 1, 1 \rangle$ and $t(3) = \langle 1, 2 \rangle$.

The expression for $\langle fib(m-2), fib(m-1) \rangle$ in the invariant I can be obtained from the equation 23 on which the recursive program P3.2 is based. In fact, since $r(m) = \langle G(1,0,m), G(0,1,m) \rangle = \langle fib(m-2), fib(m-1) \rangle$ using equations 25 and 26, we have:

$$\begin{aligned}
&\langle fib(m-2), fib(m-1) \rangle = \\
&= \langle fib^2(k-2) + fib^2(k-1),\ 2 \cdot fib(k-2) fib(k-1) + fib^2(k-1) \rangle \quad \underline{if}\ m=2k \\
&= \langle 2 \cdot fib(k-2) fib(k-1) + fib^2(k-1),\ (fib(k-2)+fib(k-1))^2 + fib^2(k-1) \rangle \quad \underline{if}\ m=2k+1
\end{aligned}$$

□

1.7 More on the comparison with other algorithms

One of the first papers which, to our knowledge, has been published on the mechanical evaluation of linear recurrence relations is [Miller and Brown 66].

Several other papers have been recently published on that topic during the last few years [Shortt 78, Gries and Levin 80, Urbanek 80, Wilson and Shortt 80, Pettorossi 80a] and we think that it is appropriate to relate those results with the ones we presented in the previous sections.

In [Miller and Brown 66] the key ideas for achieving an algorithm which requires an $O(k^3 \log n)$ multiplication for computing the n-th term of a linear recurrence relation of order k are already developed. In that paper the matrix multiplication method is used, and there are some insights for an $O(k^2 \log n)$ algorithm, when it is said that "we need to know only a single row or column of [the matrix] A_r". But then the authors (page 189) say that the matrices "are filled in by use of the recurrence relation" and this approach, if taken in a naive way requires k extra multiplications for each element of the matrices. Therefore they get an $O(k^3 \log n)$ algorithm.

Indeed in order to compute the product of two $k \times k$ matrices in semiring structures we need, in general, at least k^3 multiplications ([Jerrum and Snir 82]).

[Gries and Levin 80] (page 69) showed how to use only k^2 multiplications to "fill in the rest of [the matrices]". In order to do so the authors take advantage of the particular structure of the matrices involved. Such a structure, however, was already known to the authors of [Miller and Brown 66].

A last comment on [Miller and Brown 66] paper: some of the computation steps presented there are a bit awkward for the following two reasons: i) the multiplication by the matrix U^{-1} corresponds to an "opposite movement" in the recurrence relation to be evaluated in the sense that, for instance, in the sequence { f_i | $i \geq 0$ } they must compute f_k for computing f_{k-1}, and ii) in the final matrix multiplication, denoted there by BU_k, they need to know all the values of U_k, while in general a row of B and a column of U_k would suffice.

In [Shortt 78] the author gives an algorithm for computing Fibonacci numbers in O(log n) steps using only a fixed amount of storage. That algorithm corresponds to one requiring $O(k^2 \log n)$ multiplications (where k=2 since the Fibonacci relation is of order 2). It has been generalized to any order-k Fibonacci number in [Wilson and Shortt 80], where order-k Fibonacci numbers, denoted by $Fib^k(n)$, are defined as follows:

$$\begin{cases} Fib^k(1)=0, \;\dots,\; Fib^k(k-1)=0, \\ Fib^k(k)=1, \\ Fib^k(n)= \sum_{i=1}^{k} Fib^k(n-i) \qquad \text{for } n>k>1. \end{cases}$$

Although the authors of that last paper do not give a detailed complexity analysis of their algorithm, it is easy to see that it still requires $O(k^2 \log n)$ multiplications and a constant number of auxiliary memory cells (F(0),...,F(k-1),F2(0),...,F2(k-1)), apart from the cell(s) needed for the binary representation of the input number n for which $Fib^k(n)$ should be computed .

Let us recall the Wilson-Shortt algorithm, which we will write using the syntax adopted in [Wilson and Shortt 80]. (For the time being, the reader should not pay attention to the fact that some instructions are put within square brackets and marked by [1],...,[5].)

```
Given k and n, where $2<k<n$:
begin
[ for i=0 to k-2, F(i):=0;                  [1]
  F(k-1):=1; ]
  L:= $\lfloor \log_2 n \rfloor$;
  for i=0 to L, b(i):=ith bit of the binary representation of n;
                  /* the most significant bit is b(0) */
  for step = 1 to L, do:
      [ for d=0 to k-1,                                                  [2]

      $F2(d):=[\sum_{j=0}^{d} F(j)(\sum_{h=0}^{j} F(d-h))] + \sum_{j=d+1}^{k-1} F(j)\ [F(k+d-j) - \sum_{i=1}^{k-j-1} F(d+i)];$ ]

      if b(step)=0 then [ for d=0 to k-1, F(d):=F2(d); ]                  [3]
                   else [ do:
                          for d=0 to k-2, F(d):=F2(d+1);
                                                                         [4]
                          $F(k-1):=\sum_{i=0}^{k-1} F2(i)$;

                          end ];
                   end
  return [ F(0) ];          [5]
end
```

This algorithm is not completely satisfactory because of the following reasons:

i) it does not allow us to compute general order-k Fibonacci numbers
$F^k(n)$ for $n<k$ and

ii) it performs some unuseful computations.

Point i) can be easily remedied, realizing that:

$\forall d\ 0 \leq d \leq k-1$, F(d) gives indeed the d-th Fibonacci number.
For point ii) we notice that in the main loop of the Wilson-Shortt algorithm, driven by the index "step", a new k-tuple of Fibonacci numbers is computed from an old one, while the result to be returned at the end of the computation is a component only, namely F(0), of the last computed k-tuple. In particular when computing the nth Fibonacci number, that algorithm computes also the (n+1)th,....,(n+k-1)th numbers. Therefore a way of giving a solution to the problem of point ii) is the following: when computing the mth order-k Fibonacci number one should use the Wilson-Shortt algorithm with n initialized to m-(k-1) and then return as result F(k-1), instead of F(0).

As we already pointed out in [Pettorossi 80a], the Wilson-Shortt algorithm is quite interesting because it shows that for computing general order-k Fibonacci numbers in logarithmic time it is sufficient to know only k consecutive values of the Fibonacci sequence. This fact indeed can be generalized to any homogeneous linear recurrence relation (h.l.r.r. for short) of order k, as we have shown in Section 1.5.
A first step in this direction was made by [Urbanek 80], where an $O(k^3 \log n)$ algorithm for solving h.l.r.r. of order k is presented. There it is clearly stated that, for the initialization step, one could replace $k \times k$ matrices (as used in the matrix multiplication method) by column vectors of length k.
We can summarize the results presented in the papers we mentioned in the following table.

[Miller and Brown 66]:
Computation of order-k homogeneous linear recurrence relations.
Complexity: <u>Time</u>: $O(k^3 \log n)$ multiplications. Hints for an $O(k^2 \log n)$ algorithm.
<u>Space</u>: $O(k^2)$

[Shortt 78]:

Computation of order-2 Fibonacci numbers.

Complexity: Time: $O(k^2 \log n)$ multiplications.

Space: $O(k)$

[Wilson and Shortt 80]:

Computation of order-k Fibonacci numbers.

Complexity: Time: $O(k^2 \log n)$ multiplications.

Space: $O(k)$

[Urbanek 80]:

Computation of order-k homogeneous linear recurrence relations.

Complexity: Time: $O(k^3 \log n)$ multiplications.

Space: $O(k^2)$. Hints for $O(k)$.

[Gries and Levin 80]:

Computation of order-k homogeneous linear recurrence relations.

Complexity: Time: $O(k^2 \log n)$ multiplications.

Space: $O(k^2)$.

In those papers the correctness proof of the proposed algorithms was done via theorems.

In [Pettorossi 80a] we show that the [Wilson and Shortt 80] algorithm can be derived by transformations from the usual $k \times k$ matrix multiplication method if we use a concise representation of the matrices involved as vectors of length k. Indeed the usual algorithm based on matrix multiplications can be written as follows (for the time being do not consider the square brackets around some instructions):

The Matrix Multiplication Algorithm

```
Given k and n, where $2 \leq k < n$:
begin
[ P:=M; ]        [1]
  L:= $\lfloor \log_2 n \rfloor$;
  for i=0 to L, b(i):=ith bit of the binary representation of n;
                          /* the most significant is b(0) */
  for step = 1 to L, do:  [ M2:=P×P ; ]     [2]
                              if b(step) = 0 then [ P:=M2 ] else [ P:=M2×M ];
                                                      [3]               [4]
                          end;
  return  [ $p_{kk}$ ];     [5]
end
```

where $M = \begin{bmatrix} 1 & 1 & 1 & . & . & . & 1 & 1 \\ 1 & 0 & 0 & . & . & . & 0 & 0 \\ 0 & 1 & 0 & . & . & . & 0 & 0 \\ & & & . & . & . & & \\ 0 & 0 & 0 & . & . & . & 1 & 0 \end{bmatrix}$ is a k×k matrix and p_{kk} is the element

of the lower right corner of the k×k matrix P.

We proved that there exists an "extract" operation such that for any matrix computation above written within square brackets, the following diagram commutes:

$$\begin{tikzcd}
\text{matrices operands} \arrow[r, "\text{operation [i] of the Matrix Multiplication Algorithm within square brackets}"] \arrow[d, "\text{extract}"'] & \text{matrix result} \arrow[d, "\text{extract}"] \\
\text{vectors of values} \arrow[r, "\text{corresponding operation [i] within square brackets for Wilson–Shortt Algorithm}"'] & \text{vector of values}
\end{tikzcd}$$

The interested reader may see the details in [Pettorossi 80a].
In Section 1.5 we derived via transformations an algorithm whose features can be described as follows:

[Pettorossi and Burstall 82]:

Computation of order-k homogeneous linear recurrence relations over semirings.

Complexity: <u>Time</u>: $O(k^2 \log n)$ multiplications.

<u>Space</u>: $O(k)$

In Section 1.5 we also showed that for any sequence $\{G(i)\}$ of values defined by a homogeneous linear recurrence relation of order k with constant coefficients we can express $G(2m), G(2m+1), \ldots, G(2m+k-1)$ in terms of $G(m), G(m+1), \ldots, G(m+k-1)$. That means that "suitable" recurrence formulae, as defined in [Wilson and Shortt 80] page 69, exist for any homogeneous linear recurrence relation.

A last remark concerns the length of the correctness proof. Some pages of Theorems and Lemmas were necessary in [Wilson and Shortt 80] to show the correctness of the algorithm.
In Section 1.5 using the program transformation technique we are able to show, without much effort, the correctness of our iterative algorithm, which is a generalization of one by Wilson-Shortt, starting from the general definition of the linear recurrence relations. We obtained that derivation by a simple application of some strategies, well-known in the program transformation methodology.

1.8 On the application of the tupling strategy: the general method and some of its properties

In this section we will make some comments on the tupling strategy as we used it in Sections 1.3 and 1.5, and we will present some of its properties. A first comment concerns the fact that in the program P2 (see Section 1.3) the tupling strategy could have been applied in a different way. But, as we already mentioned, the resulting program would have been equivalent to the program P3 (see Section 1.3) ("equivalent" here means that the programs determine the same computations for any input). Indeed we will show that the elimination of redundant subcomputations evoked by equation 13 in program P2 avoids also the redundant subcomputations evoked by equation 12 and viceversa.

A second comment is related to the symbolic evaluation analysis needed for discovering common subcomputations. We will show that an interesting property, which we will call safety property, holds for the tupling strategy. It can be formulated as follows: if the analysis of the redundancy is not complete and we "tuple together" only some of the functions which determine redundant (sub)computations, then the "undiscovered" redundancy still remains and it can be avoided by applying again the tupling strategy itself.

We will then give some other examples of the application of the tupling strategy. We will also present a "limitation result" and we will exhibit a class of program schemata for which the tupling strategy cannot completely avoid redundancy in recursive calls. (The same result holds for any other method which makes use of a bounded number of memory cells.)

1.8.1 An alternative use of the tupling strategy in program P2

Looking at equation 13 of program P2 by symbolic evaluation we discover the common subcomputations as illustrated in fig. 1-7 (analogous to fig. 1-1).

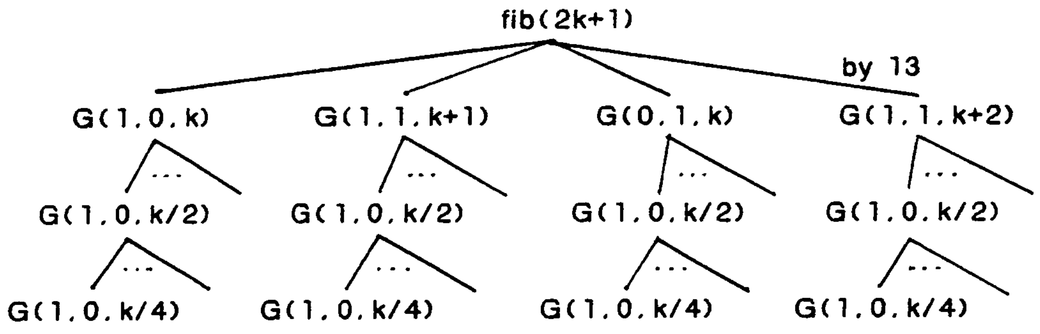


Figure 1-7: Redundant computations in program P2. G(1,0,k/4) occurs 4 times in evaluating fib(2k+1). (Suppose k/2 even).

We will use the tupling strategy by defining the following function z(k):

$z(k) = \langle G(1,0,k), G(0,1,k), G(1,1,k+1), G(1,1,k+2)\rangle$.

Using the equations 4., 5. and 6. of Section 1.3 we get the explicit definition for z(k), as follows:

$z(0) = \langle G(1,0,0), G(0,1,0), G(1,1,1), G(1,1,2)\rangle$

$= \langle 1,0,1,2\rangle$

$z(2k) = \langle G(1,0,2k), G(0,1,2k), G(1,1,2k+1), G(1,1,2k+2)\rangle$

$= \langle G(1,0,k)\cdot G(1,0,k)+G(0,1,k)\cdot G(1,0,k+1),$

$G(1,0,k)\cdot G(0,1,k)+G(0,1,k)\cdot G(0,1,k+1),$

$G(1,0,k)\cdot G(1,1,k+1)+G(0,1,k)\cdot G(1,1,k+2),$

$G(1,0,k)\cdot G(1,1,k+2)+G(0,1,k)\cdot G(1,1,k+3)\rangle$ by 9'

$= \langle a^2+b^2, b(2a+b), ac+bd, ad+b(c+d)\rangle$ <u>where</u> $\langle a,b,c,d\rangle = z(k)$

and analogously,

$z(2k+1) = \langle G(1,0,2k+1), G(0,1,2k+1), G(1,1,2k+2), G(1,1,2k+3)\rangle$

$= \langle 2ab+b^2, (a+b)^2+b^2, ad+b(c+d), a(c+d)+b(2d+c)\rangle$

where $\langle a, b, c, d\rangle = z(k)$

We also have:

$fib(2k) = G(1,1,2k)$

$= G(1,0,k) \cdot G(1,1,k) + G(0,1,k) \cdot G(1,1,k+1)$ by 9'

$= a(d-c)+bc$ where $\langle a, b, c, d\rangle = z(k)$

$fib(2k+1) = G(1,1,2k+1)$

$= G(1,0,k) \cdot G(1,1,k+1) + G(0,1,k) \cdot G(1,1,k+2)$ by 9'

$= ac+bd$ where $\langle a, b, c, d\rangle = z(k)$

and we can derive the following program for computing the Fibonacci function in logarithmic time:

Program P3.2

```
fib(0) = 1
fib(1) = 1
fib(2k)   = a(d-c)+bc                    where <a,b,c,d> = z(k)
fib(2k+1) = ac+bd                        where <a,b,c,d> = z(k)
z(0)      = <1,0,1,2>
z(2k)     = <a^2+b^2, b(2a+b), ac+bd, ad+b(c+d)>
                                         where <a,b,c,d> = z(k)
z(2k+1) = <2ab+b^2, (a+b)^2+b^2, ad+b(c+d), a(c+d)+b(2b+c)>
                                         where <a,b,c,d> = z(k)
```

Fact 5. Program P3.2 computes the same function computed by program P3 (see Section 1.3).

Proof. It is enough to prove that:

$z(k) = \langle a, b, d, c+d\rangle$ where $\langle a, b, c, d\rangle = r(k)$. Immediate by induction on k. □

The same alternative method of applying the tupling strategy in program P3 we have now shown, could have been used also in the program for evaluating linear recurrence relations, with the result of deriving a program equivalent to the one we presented in Section 1.5.

1.8.2 A "safety property" for the tupling strategy

We will study the "safety property" of the tupling strategy by first presenting an example of its use.
We take our example again from the evaluation of linear recurrence relations.
As we already mentioned the safety property is a guarantee against the incomplete analysis of redundancy.
Suppose that in the program P2 of Section 1.3 we discovered, through symbolic evaluation, that the functions G(1,0,k) and G(1,1,k) require the computation of G(1,0,k/2) (see equation 12.) and we did not realize that also functions G(0,1,k) and G(1,1,k+1) require it.
In that case we would have defined, instead of the function t(k) of equation 18., the following function p(k):

p(k) = <G(1,0,k),G(1,1,k)> (PARTIAL TUPLING EUREKA)

We can easily derive that:

p(0) = <1,1>

p(2k) = $<a^2+G^2(0,1,k),\ ac+G(0,1,k)\cdot G(1,1,k+1)>$
where <a,c> = p(k)

p(2k+1) = $<2aG(0,1,k)+G^2(0,1,k),\ aG(1,1,k+1)+G(0,1,k)(c+G(1,1,k+1))>$
where <a,c> = p(k)

Thus we get from program P2 the following program:

fib(0) = 1
fib(1) = 1
fib(2k) = ac+G(0,1,k)·G(1,1,k+1) where <a,c> = p(k)
fib(2k+1) = a·G(1,1,k+1)+G(0,1,k)·(c+G(1,1,k+1)) where <a,c> = p(k)
together with the above equations for p(0), p(2k) and p(2k+1) and
the equations 14, 15, 16 and 17 for $G(a_0,a_1,k)$ (see Section 1.3).

If we look for more redundancies we may discover (see fig. 1-8) that we need to tuple the functions p(k), G(0,1,k) and G(1,1,k+1).

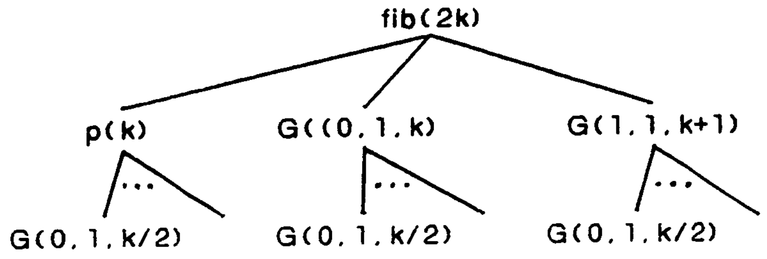


Figure 1-8: Redundant computations in evaluating fib(2k). G(0,1,k/2) occurs 3 times. (Suppose k even.)

Therefore, applying the tupling strategy again, we will define:

q(k) = <p(k), G(0,1,k), G(1,1,k+1)>   and we get:

$q(0) = \langle\langle 1, 1\rangle, 0, 1\rangle$

$q(2k) = \langle\langle a^2+b^2, ac+bd\rangle, b(2a+b), ad+b(c+d)\rangle$ <u>where</u> $\langle\langle a,c\rangle, b, d\rangle = q(k)$

$q(2k+1) = \langle\langle 2ab+b^2, ad+b(c+d)\rangle, (a+b)^2+b^2, a(c+d)+b(2d+c)\rangle$ <u>where</u> $\langle\langle a,c\rangle, b, d\rangle = q(k)$

Here is the resulting program for computing the Fibonacci function.

```
fib(0) = 1
fib(1) = 1
fib(2k)   = ac+bd              where <<a,c>,b,d> = q(k)
fib(2k+1) = ad+b(c+d)          where <<a,c>,b,d> = q(k)
together with the equations above for q(0), q(2k) and q(2k+1).
```

This program is equivalent to program P3 of Section 1.3 because we have that t(k) of program P3 is equal, by definition, to the 4-tuple $\langle\pi1(\pi1(q(k))), \pi2(q(k)), \pi2(\pi1(q(k))), \pi3(q(k))\rangle$.

Now we can generalize the result we have obtained in the above example. Suppose we are given a recursive equation program of the form:

$$\left[\begin{array}{l} f(\underline{k}) \Leftarrow r(g(p_1(\underline{k})), \ldots, g(p_n(\underline{k})), \underline{k}) \\ \qquad\qquad \ldots \qquad \ldots \\ g(\underline{h}) \Leftarrow s(g(q_1(\underline{h})), \ldots, g(q_m(\underline{h})), \underline{h}) \end{array}\right.$$

where f and g are two functional variables, $p_1, \ldots, p_n$, $q_1, \ldots, q_m$ are basic operators, and r and s are basic operators or if-then-else.

$\underline{k}$ and $\underline{h}$ denote tuples of arguments. If $\underline{k}$ is $\langle k_1, \ldots, k_r \rangle$ then $p(\underline{k})$ is $p(k_1, \ldots, k_r)$. Analogously for $\underline{h}$.

The results we will present here could be further generalized:

i) by considering more than two function definitions (not f and g only, as in our case),

ii) by allowing the p_i's and the q_i's to have different arities;

iii) by allowing r and s to be of the form: if-then-elseif-then-...-else.

Without loss of generality, let us assume that three different calls of the function g require the same subcomputation.

The situation can be depicted in fig. 1-9 where $1 \leq r,s,t \leq n$ and $1 \leq u,v,w \leq m$. In such a situation there are three function calls, namely $g(p_r(\underline{k}))$, $g(p_s(\underline{k}))$ and $g(p_t(\underline{k}))$, which all require the computation of $g(q_u(p_r(\underline{k})))$.

Therefore we have: $q_u(p_r(\underline{k})) = q_v(p_s(\underline{k})) = q_w(p_t(\underline{k}))$.

There are two cases depending on whether or not the analysis of the redundancy is complete.

Case 1. The analysis of the redundancy is not complete.

Without loss of generality, let us assume that in applying the tupling strategy we pair together only the functions $g(p_r(\underline{k}))$ and $g(p_s(\underline{k}))$. Therefore we define the function:

$a(\underline{k}) = \langle g(p_r(\underline{k})), g(p_s(\underline{k})) \rangle$.

We now assume (Uniqueness Assumption) that, for a given "decrement function" da, there is a unique way of expressing $a(\underline{k})$ as follows:

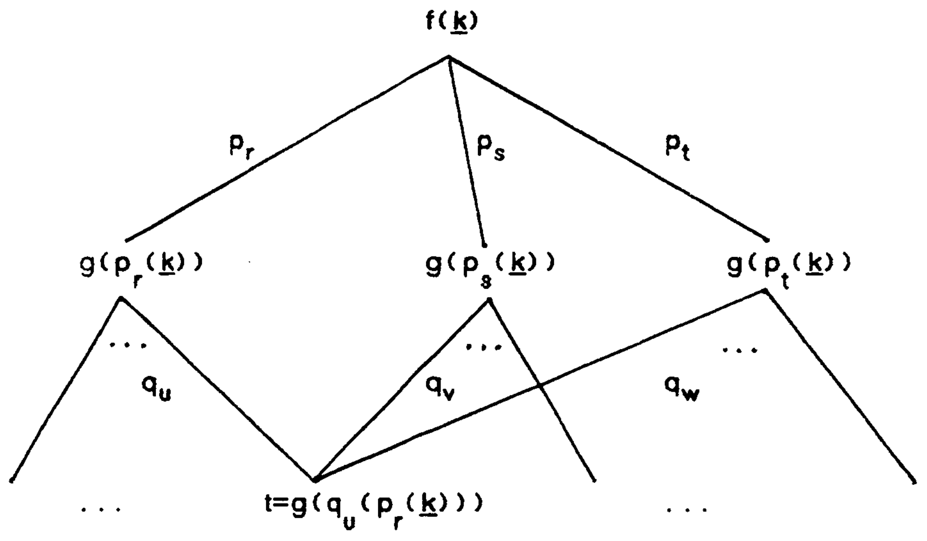


Figure 1-9: Computations which share the same subcomputation.

$a(\underline{k}) = \langle 1a[a_1, a_2, t, \underline{k}],\ 2a[a_1, a_2, t, \underline{k}]\rangle$ __where__ $t = ca[a_1, a_2, \underline{k}]$
__where__ $\langle a_1, a_2\rangle = a(da(\underline{k}))$

where

(i) $1a[a_1, a_2, t, \underline{k}]$ (i.e. the "first component" for a) and $2a[a_1, a_2, t, \underline{k}]$ (i.e. the "second component" for a) are two expressions with possible occurrences of the variables a_1, a_2, t and $\underline{k}$; and

(ii) $ca[a_1, a_2, \underline{k}]$ (i.e. the "common subterm" for a) is an expression (with possible occurrences of the variables a_1, a_2 and $\underline{k}$) equal to $g(q_u(p_r(\underline{k})))$.

For example, in the case of the Fibonacci evaluation, when we obtained the function p(k) (see this Subsection), we have:

$1a[a_1, a_2, t, \underline{k}]$: __if__ k=0 __then__ 1 __elseif__ even(k) __then__ $a_1^2+G^2(0, 1, k)$
__else__ $2 \cdot a_1 G(0, 1, k)+G^2(0, 1, k)$

$2a[a_1, a_2, t, \underline{k}]$: <u>if</u> k=0 <u>then</u> 1 <u>elseif</u> even(k) <u>then</u> $a_1a_2+G(0,1,k)\cdot G(1,1,k+1)$
<u>else</u> $a_1\cdot G(1,1,k+1)+G(0,1,k)\cdot(a_2+G(1,1,k+1))$

$ca[a_1, a_2, \underline{k}]$: a_1

$da(\underline{k})$: <u>if</u> even(k) <u>then</u> k/2 <u>else</u> (k−1)/2.

In most of the examples we have studied the Uniqueness Assumption does hold by allowing only "standard applications" of the tupling strategy. A formal definition of the "standard application" notion will not be given here, because it has only a technical relevance for expressing the conditions under which the Uniqueness Assumption is valid.

We only say that, when defining the tupled function $a(\underline{k})$, the standard application forces $ca[a_1, a_2, \underline{k}]$ to be a_1 (or a_2) if a_1 (or a_2) is the common subterm of the components of $a(\underline{k})$.

We also assume that for the decrement function a "termination condition" holds, in the sense that the sequence of recursive calls of $\{ a(da^n(\underline{k})) \mid n \geq 0 \}$ is well-founded.

<u>Case 2</u>. The analysis of the redundancy is <u>complete</u>.

The tupling strategy gives us the following function $c(\underline{k})$:

$c(\underline{k}) = \langle g(p_r(\underline{k})),\ g(p_s(\underline{k})),\ g(p_t(\underline{k}))\rangle$.

By the Uniqueness Assumption, given the "decrement function" dc, there is a unique way of expressing $c(\underline{k})$ as follows:

$c(\underline{k}) = \langle 1c[c_1, c_2, c_3, t, \underline{k}],\ 2c[c_1, c_2, c_3, t, \underline{k}],\ 3c[c_1, c_2, c_3, t, \underline{k}]\rangle$
<u>where</u> $t = cc[c_1, c_2, c_3, \underline{k}]$
<u>where</u> $\langle c_1, c_2, c_3\rangle = c(dc(\underline{k}))$

where

(i) 1c[...], 2c[...], 3c[...] and cc[...] are expressions with possible occurrences of the variables c_1, c_2, c_3, and $\underline{k}$, and

(ii) the expression cc[...] is equal to $g(q_u(p_r(\underline{k})))$, i.e. the "common subterm" shared by the projections of $c(\underline{k})$.

Fact 6. If $da(\underline{k}) = dc(\underline{k})$ then

$1a[a_1, a_2, t, \underline{k}] \equiv 1c[a_1, a_2, g(p_t(da(\underline{k}))), t, \underline{k}]$

$2a[a_1, a_2, t, \underline{k}] \equiv 2c[a_1, a_2, g(p_t(da(\underline{k}))), t, \underline{k}]$

$ca[a_1, a_2, \underline{k}] \quad \equiv cc[a_1, a_2, g(p_t(da(\underline{k}))), \underline{k}]$

Proof. Immediate consequence of the Uniqueness Assumption, because the common term t is the same for $a(\underline{k})$ and $c(\underline{k})$. □

Now the "safety property" for the tupling strategy can be defined as follows:

1. given a recursive equations program p, where all functions in the set $F = \{ f_1, \ldots, f_n \}$ share the computation of a term t, if the tupling strategy is applied to the functions in a subset S of F, thereby defining the new function a, then in the resulting program the functions in F - S and the tupled function a still share the computation of t;

2. if we apply again the tupling strategy to the functions in $\{ a \} \cup F - S$, then the resulting program is equivalent to the one obtainable from the program p by tupling together all functions in F.

Fact 7. With reference to the definitions given above, if $da(\underline{k}) = dc(\underline{k})$ then the safety property holds for the tupling strategy.

Proof. We have to show that we can pair together the functions $a(\underline{k})$ and $g(p_t(\underline{k}))$. They share the same subcomputation of the term $t = g(q_u(p_r(\underline{k})))$ because $q_u(p_r(\underline{k})) = q_w(p_t(\underline{k}))$, and therefore pairing is indeed possible.
We can then define the new function: $b(\underline{k}) = \langle a(\underline{k}), g(p_t(\underline{k})\rangle$. In order to complete the proof we need to show that:
$c(\underline{k}) = \langle \pi 1(a(\underline{k})), \pi 2(a(\underline{k})), g(p_t(\underline{k}))\rangle$. That equality can easily be shown by recursion induction, using the results of the previous Fact 6. □

Remark. The proof of the "safety property" has been greatly simplified by the Uniqueness Assumption. As it stands, our proof is essentially based on the isomorphism between the two n-tuples $\langle \ldots \langle\langle x_1, x_2\rangle, x_3\rangle, \ldots, x_n\rangle$ and $\langle x_1, x_2, x_3, \ldots, x_n\rangle$.

The result we have obtained can be generalized to any k-tuple of function calls (not triples only) and any number of intermediate steps (not two successive pairings only).
We also think that the safety property holds under much weaker hypotheses than the Uniqueness Assumption: we leave that study for elsewhere.

1.8.3 Application of the tupling strategy

Let us consider the class $\{ S_n \mid n \geq 0 \}$ of recursive program schemata each of which is of the following form:

S_n: $f(x) = a(x)$ <u>if</u> $p(x)$
$\quad f(x) = b(x, f(c_1(x)), \ldots, f(c_n(x)))$ <u>otherwise</u>

Without loss of generality we may restrict our attention to the case $n = 2$. Let us denote c_1 by c and c_2 by d. We have:

S_2: $f(x) = a(x)$ <u>if</u> $p(x)$
$\quad f(x) = b(x, f(c(x)), f(d(x)))$ <u>otherwise</u>

We will now study some properties of the tupling strategy for the schema S_2. Those properties can easily be extended to any schema S_n, for any n.

We will follow the terminology introduced in [Cohen 83].
We assume that the functions a, b, c, d and p are strict and their evaluation has no side effects. Under these hypotheses the call by value mechanism allows us to compute f(x) as the least fixed point of S_2. The parameter x stands for either a scalar or a k-tuple (for $k \geq 2$).

Definition 1: The <u>descent tree</u> for S_2 is a tree whose root is f(x) and if f(y) is one of its nodes, then f(c(y)) and f(d(y)) are the son-nodes of that node.

Sometimes we allow ourselves to use variants of descent trees, where we avoid writing the function symbol f and/or the argument symbol x and/or the parentheses.

Notice that the descent tree for S_2 is an infinite tree.

Figure 1-10 shows the "upper part" of the descent tree for S_2.

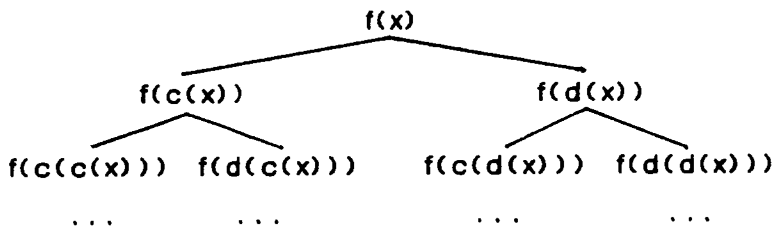


Figure 1-10: "Upper part" of the descent tree for S_2

In the descent tree some of the nodes may be identified if we assume that some conditions hold for the functions c and d.

For example, if c(d(x)) = d(c(x)) the "upper part" of the descent tree for S_2 will look like the directed acyclic graph (dag) in fig. 1-11.

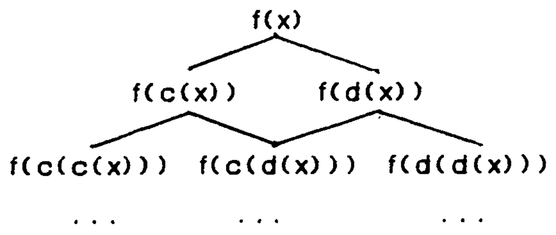


Figure 1-11: "Upper part" of the descent tree for S_2 if c(d(x)) = d(c(x))

A descent tree with some nodes which are identified is called a <u>compressed descent dag</u>.

Under a given interpretation, if a compressed descent dag does not have two distinct nodes which can be identified, then it is called a minimal compressed descent dag. (It is also called a dependency graph in [Bird 80].) It denotes the computation of f(x) without redundancy, if we assume that it is truncated in such a way that the arguments occurring in the leaves make the predicate p(x) to be true.

Let D be the domain where the function f is defined. It can be partitioned in two subdomains as follows:

$D = B \cup R$, s.t. $B \cap R = \emptyset$, where $\forall x \in B$. $p(x)$ = true and

$\forall x \in R$. $p(x)$ = false.

We say that B is the set of base cases and R is the set of recursive cases.

Let us consider a set of functions $S = \{ h_i \mid \text{for } i>0 \}$. We say that the frontier condition holds for S w.r.t. the function f iff

$\forall x \in B$. $\forall h_i \in S$. [$h_i(x)$ and $a(h_i(x))$ are defined and

if $f(h_i(x))$ is defined then $f(h_i(x)) = a(h_i(x))$].

In other words, $\forall x \in B$. $\forall h_i \in S$. if $p(x)$ = true then $p(h_i(x))$ = true, i.e. if x is a base case for f then $h_i(x)$ is a base case as well.

Now we give an algorithm for the application of the tupling strategy, as a method of transforming the schema S_2 to derive a more efficient program schema.

1. Derive the minimal compressed descent dag (called m-dag) of the function defined by S_2, via symbolic evaluation.

2. Let a "cut" in an m-dag be a set of nodes s.t. if we remove them, together with the incoming and outgoing edges, the m-dag is divided into two disconnected graphs.
   In the m-dag derived in Step 1, we determine a totally ordered

sequence of cuts (possibly not disjoint) which is consistent with the function calls relationships, i.e. the cut which includes a "son node" does not precede the cut which includes the "father node" (unless both nodes are in both cuts). Those cuts are such that the function calls occurring in a cut can be evaluated using only the function calls of the cut which immediately follows it in the total order (Cut Isolation Property).

3. Tuple together the function calls in a cut and find their recursive definition in terms of the function calls of the cut which follows (in the total order of the cuts).

4. Check the base cases definitions so that the recursive definition found in the previous step is well-founded.

5. Express the evaluation of the given function in terms of the tupled functions.

The following examples will clarify the ideas.

Example 7.

Consider the familiar problem of the towers of Hanoi, with 3 pegs A, B and C. It is required to move a pile of n disks with $n \geq 0$ from peg A to peg B using peg C as an auxiliary peg, under the constraint that no disk can be placed on top of a smaller one.

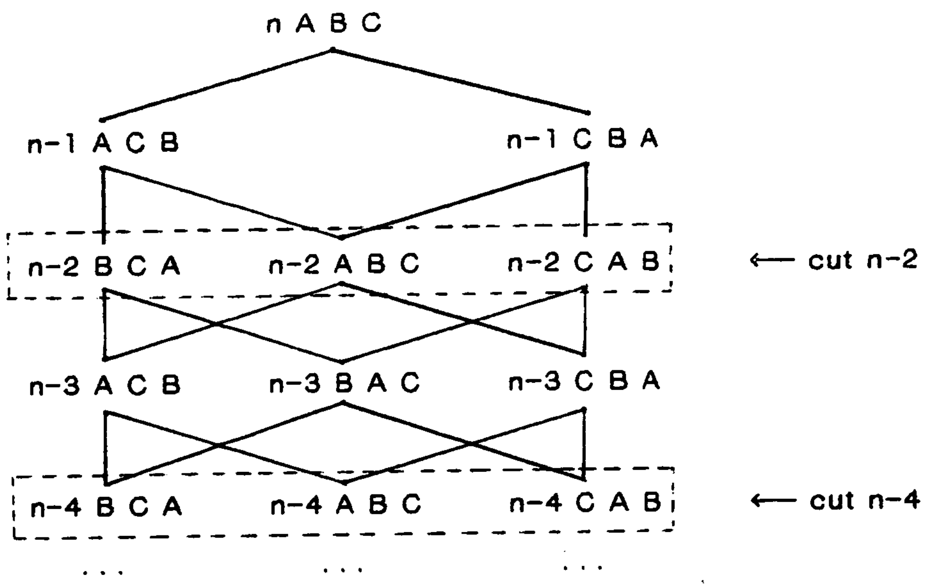
The solution may be expressed by the following recursive program:

```
f(n,A,B,C) = skip                              if n = 0
f(n,A,B,C) = f(n-1,A,C,B)::AB::f(n-1,C,B,A)    if n ≥ 1
```

where f(n,A,B,C) moves n disks from peg A to peg B using C as auxiliary peg, XY denotes the move of the top disk from peg X to peg Y, :: denotes the concatenation of moves and "skip" is the "empty move".

Notice that the above definition of the function f is an instance of the schema S_2.

Step 1. m-dag for f(n,A,B,C):



Step 2. A set of function calls which produces cuts in the m-dag is:
{ f(k,B,C,A), f(k,A,B,C), f(k,C,A,B) } (see cuts (n-2) and (n-4)).

Step 3. Tupling the above functions together we have:
r(k) = <f(k,A,B,C), f(k,B,C,A), f(k,C,A,B)>

(The order of the functions is not significant.)

```
r(0) = <skip, skip, skip>
r(k) = <(u::AC::v) :: AB :: (w::CB::u),
        (v::BA::w) :: BC :: (u::AC::v),
        (w::CB::u) :: CA :: (v::BA::w)>      by unfolding (assuming k≥2)
           where <u,v,w> = r(k-2)
     = <x::AB::z, y::BC::x, z::CA::y>
           where x = u::AC::v, y = v::BA::w, z = w::CB::u
           where <u,v,w> = r(k-2)
```

Step 4. In order to achieve a well-founded definition of the function r(k), since r(k) calls r(k−2) and we have already the definition of r(0), we need to give the definition of r(1). By unfolding we get:

r(1) = <AB, BC, CA>

Step 5.

```
f(n,A,B,C) = skip                                   if n = 0
f(n,A,B,C) = AB                                     if n = 1
f(n,A,B,C) = f(n-1,A,C,B)::AB::f(n-1,C,B,A)
           = (u::AC::v)::AB::(w::CB::u)
                where <u,v,w> = r(n-2)              if n ≥ 2
```

Putting together the above equations for f (see Step 5.) and the ones for r we have a linear recursive program for solving the Hanoi towers problem. (Notice that the original straightforward solution did not exhibit a linear recursion.)

The fact that we obtained a linear recursion is a general property of the application of the tupling strategy because cuts are totally ordered and the Cut Isolation property holds.

When a linear recursion is obtained, it can always be removed without use of a stack. Here we will not go into the details of this subsidiary problem. The interested reader may refer to [Paterson and Hewitt 70, Walker and Strong 72, Chandra 73, Swamy and Savage 79]. By applying the recursion removal techniques described in those papers, one may obtain various iterative algorithms, for solving the Hanoi towers problem. (Therefore those algorithms do not need any proof of correctness, unlike the ones in [Hayes 77, Er 82]). We only recall that the recursive schema RH (fig. 1-12) can be translated into the flowchart FH of fig. 1-13 (see [Walker and Strong 72]).

$$\text{Schema RH:} \qquad f(n) = \begin{cases} e0 & \text{if } n = 0 \\ e1 & \text{if } n = 1 \\ a(f(n-2)) & \text{if } n \geq 2 \end{cases}$$

Figure 1-12: Recursive Schema for the Towers of Hanoi problem

We get the following iterative program for moving a tower of n disks from peg A to peg B:

```
if n = 0 then res := skip
elseif n = 1 then res := AB
else begin if even(n) then r := <skip, skip, skip>
           else r := <AB, BC, CA>;
           while n > 3 do r := <r1::AC::r2::AB::r3::CB::r1,
                               r2::BA::r3::BC::r1::AC::r2,
                               r3::CB::r1::CA::r2::BA::r3>;
                          n := n-2
                      od
           res := r1::AC::r2::AB::r3::CB::r1
     end
```

where ri denotes the i-th component of r, for i=1,2,3.

The assignment for obtaining the "new" value of r from the "old" one is a "parallel assignment", i.e. the components on the r.h.s. are all referring to the old values.

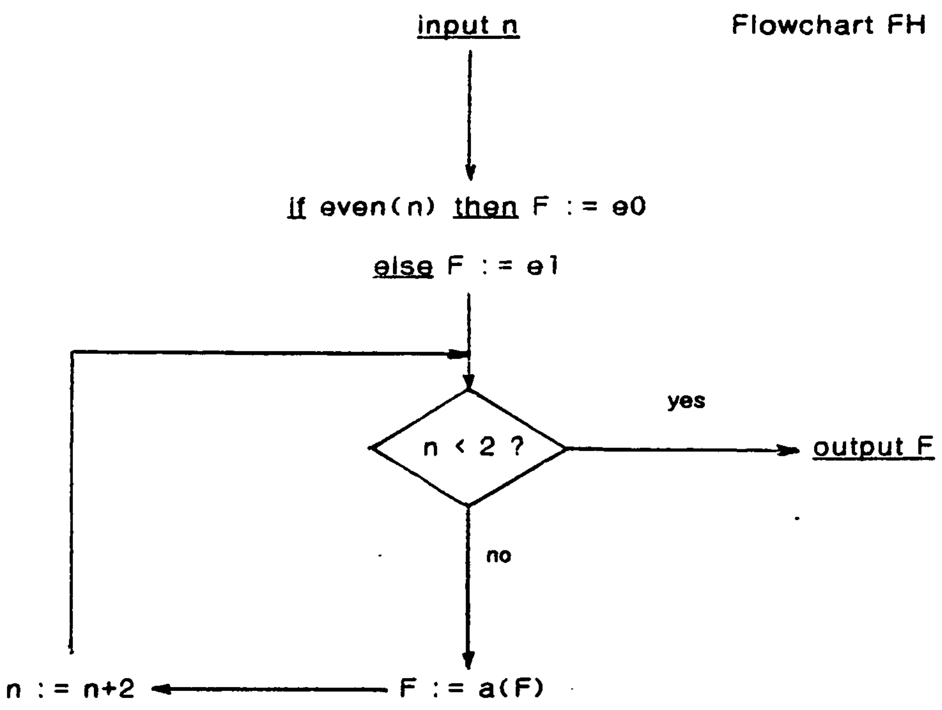


Figure 1-13: Flowchart Schema for the Towers of Hanoi problem

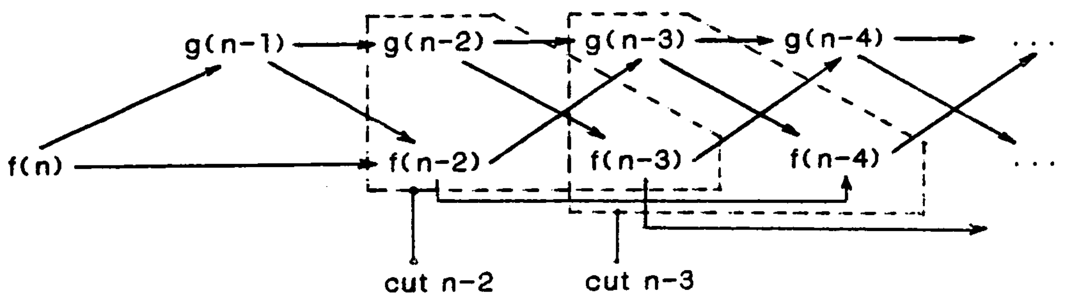
Example 8.

It is taken from [Bird 80] pp. 409-410.

Consider the following mutually recursive definitions of f and g:

$$\left[\begin{array}{ll} f(n) = 1 & \underline{\text{if}}\ n \leq 1 \\ f(n) = 2 \cdot f(n-2) - 3 \cdot g(n-1) & \underline{\text{otherwise}} \\ g(n) = 1 & \underline{\text{if}}\ n \leq 1 \\ g(n) = f(n-1) + g(n-1) & \underline{\text{otherwise}} \end{array}\right.$$

Step 1. m-dag for f and g:



Steps 2, 3 and 4.

p(n) = <g(n), f(n), f(n-1)>

We get:

p(n) = <1, 1, 1> _if_ n ⩽ 1

p(n) = <f(n-1) + g(n-1), 2·f(n-2) - 3·g(n-1), f(n-1)>

= <v + u, 2w - 3u, v> _where_ <u, v, w> = p(n-1) _otherwise_

Step 5.

f(n) = 1 _if_ n ⩽ 1

f(n) = 2w - 3u _where_ <u, v, w> = p(n-1) _otherwise_

Notice that the tupling strategy may determine the evaluation of unnecessary expressions (see for instance the v component in the second equation of Step 5).

In order to avoid that difficulty we may use a "call by need" approach to the computation of the various components of the tupled functions occurring in _where_ clauses, or we may introduce suitable conditional expressions (see the following example).

<u>Example 9.</u>

This example is taken from [Atkinson 81]. It is the "cyclic Towers of Hanoi" problem. This problem is a simple modification of the standard one: the 3 pegs are supposed to be cyclically arranged and the moves are allowed in the <u>clockwise</u> direction only (i.e. AB, BC, and CA are the only moves allowed).

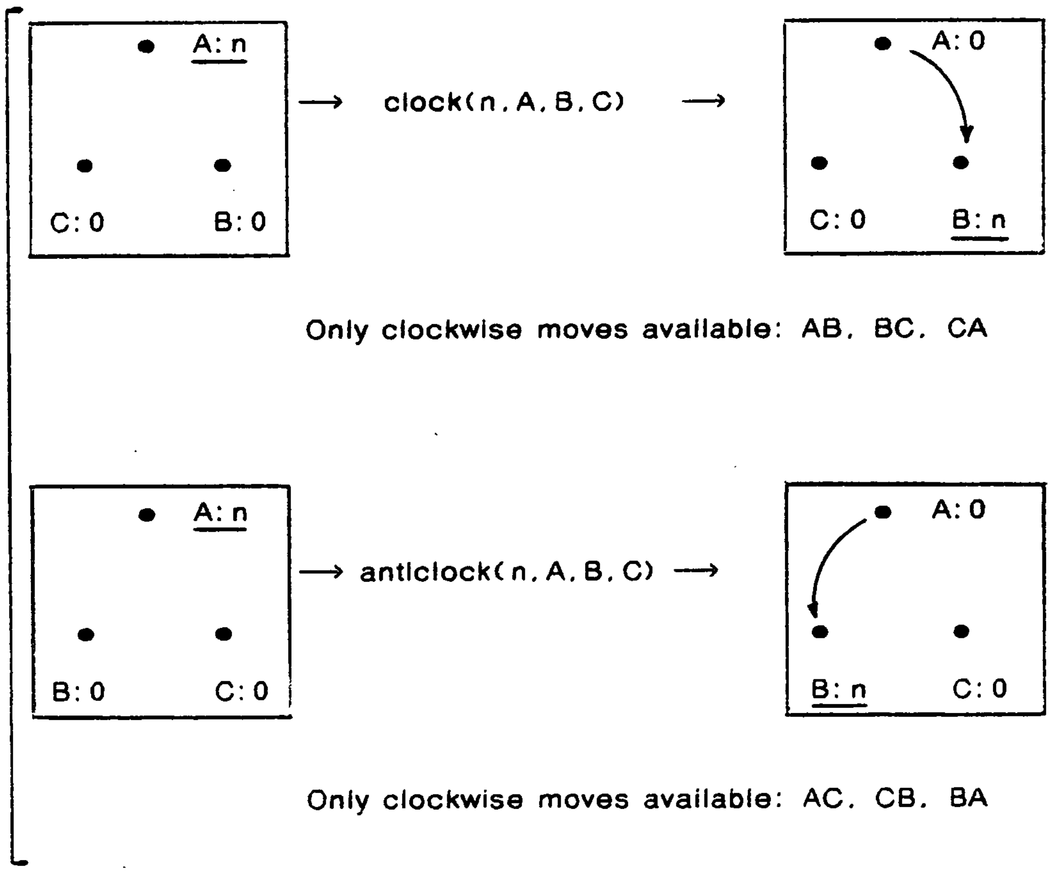


Figure 1-14: The Cyclic Towers of Hanoi problem. The "clock" and "anticlock" functions. "X: n" means "n disks are in the peg X".

The solution may be expressed as follows (see also fig. 1-14):

RECURSIVE CYCLIC TOWERS OF HANOI

Moving a tower of n disks from peg A to peg B. Allowed moves: AB, BC, CA.

clock(n, A, B, C) = skip *if* $n = 0$

clock(n, A, B, C) = anticlock(n−1, A, C, B) : : AB: : anticlock(n−1, C, B, A)

if $n \geqslant 1$

anticlock(n, A, B, C) = skip *if* $n = 0$

anticlock(n, A, B, C) = anticlock(n−1, A, B, C) : : AC: : clock(n−1, B, A, C)

: : CB: : anticlock(n−1, A, B, C) *if* $\geqslant 1$

Step 1. m−dag as in fig. 1−15.

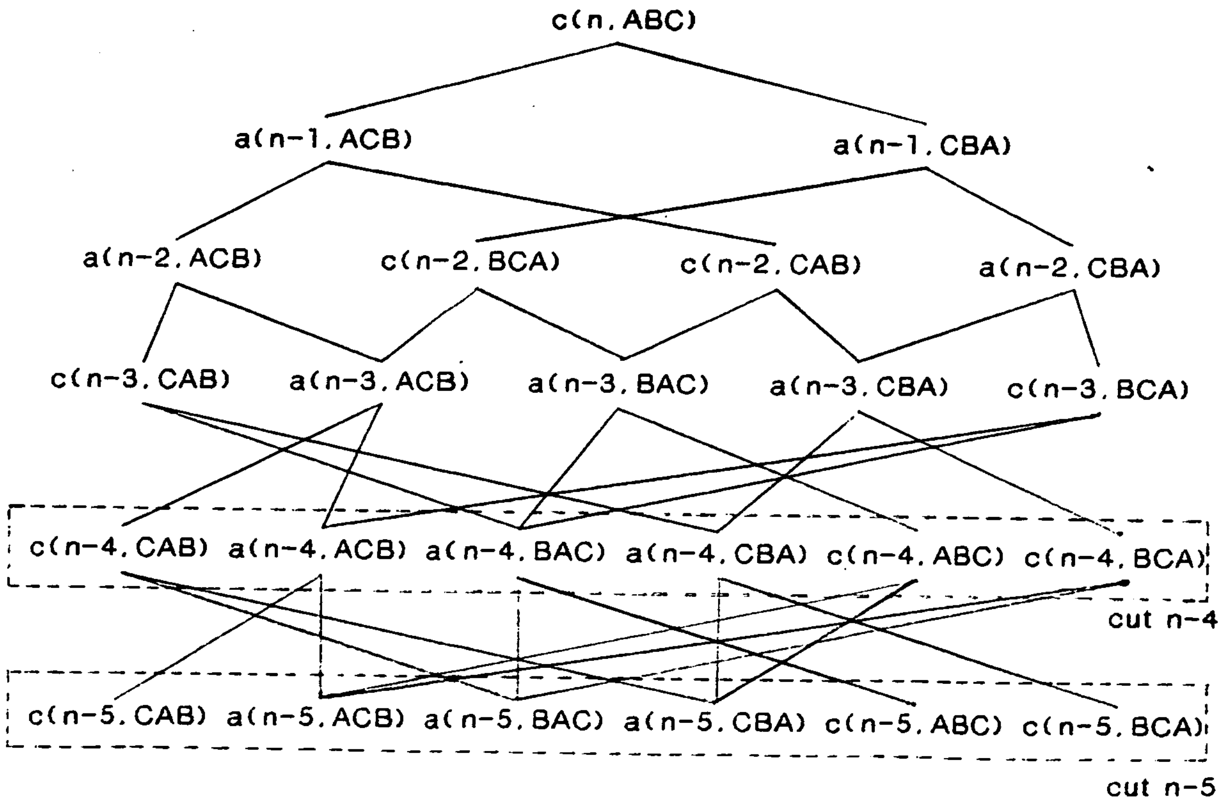


Figure 1−15: The m−dag for the Cyclic Towers of Hanoi problem.
c(n, xyz) stands for clock(n, x, y, z).
a(n, xyz) stands for anticlock(n, x, y, z).

<u>Steps 2, 3 and 4</u>.

We write c(n,x,y,z) for clock(n,x,y,z) and a(n,x,y,z) for anticlock(n,x,y,z). We define:

```
t(n,A,B,C) = < c(n,C,A,B), a(n,A,C,B), a(n,B,A,C),
               a(n,C,B,A), c(n,A,B,C), c(n,B,C,A) >.
```

We have:

```
t(n,A,B,C) = <skip, skip, skip, skip, skip, skip>              if n = 0
t(n,A,B,C) = <t4::CA::t3, t2::AB::t1::BC::t2, t3::BC::t5::CA::t3,
              t4::CA::t6::AB::t4, t2::AB::t4, t3::BC::t2>
                 where <t1,t2,t3,t4,t5,t6> = t(n-1,A,B,C)      if n ≥ 1
```

<u>Step 5</u>.

```
clock(n,A,B,C) = skip                                          if n = 0
clock(n,A,B,C) = t2::AB::t4
                 where <t1,t2,t3,t4,t5,t6> = t(n-1,A,B,C)      if n ≥ 1
```

We can easily obtain an iterative solution using the same technique we applied in the standard Towers of Hanoi problem.

ITERATIVE CYCLIC TOWERS OF HANOI

Moving a tower of n disks from peg A to peg B. Allowed moves: AB, BC, CA.

```
if n = 0 then res := skip
else begin T := <skip, skip, skip, skip, skip, skip>;
        while n > 1 do T := <T4::CA::T3,
                             T2::AB::T1::BC::T2,
                             T3::BC::T5::CA::T3,
                             T4::CA::T6::AB::T4,
                             T2::AB::T4,
                             T3::BC::T2>;
                        n := n-1
                     od
        res := T2::AB::T4
   end
```

As usual Ti denotes the i-th component of T for i = 1,...,6, and the assignment from the "old" value of T to the "new" one is a parallel assignment.

Notice that an iterative solution for the Cyclic Towers of Hanoi problem was claimed to be not easy to find ([Atkinson 81] page 118).

The iterative algorithm we have presented produces exactly the same sequence of moves which is produced by the recursive algorithm. As we already remarked in the previous example, we could improve the iterative algorithm taking into consideration the fact that in the tuple T only the components T2 and T4 are needed for computing the final sequence of moves. The introduction of suitable conditional expressions may realize that improvement.

In particular, by unfolding we can get:

```
if n = 0 then res := skip
if n = 1 then res := AB
if n = 2 then res := AB::BC::AB::CA::AB
else begin
        ...(as above)...
    end
```

Example 9.1

In this example we derive an iterative algorithm for computing the "fusc" function defined as follows:

```
fusc(0)    = 0
fusc(1)    = 1
fusc(2n)   = fusc(n)
fusc(2n+1) = fusc(n) + fusc(n+1)
```

Our derivation is an answer to Prof. Dijkstra's challenge ([Dijkstra 82] pages 215, 230), and it can be contrasted with the one given in [Bauer and Wössner 81] pages 288-290. There the iterative algorithm is obtained at the expense of a "linear combination eureka step" which is not straightforward.

The m-dag for fusc(n) is given in fig. 1-16.

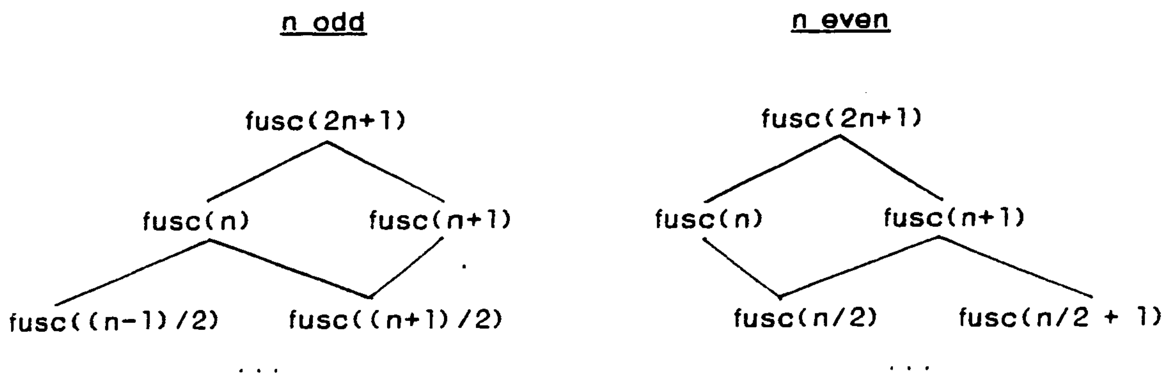


Figure 1-16: The m-dag for the fusc function.

At any given level the m-dag can be cut by the pair ⟨fusc(n), fusc(n+1)⟩. Therefore we can apply the tupling strategy and we get:

```
fusc(n)   = $\pi 1(g(n))$
g(0)      = ⟨0, 1⟩
g(2n)     = ⟨fusc(2n), fusc(2n+1)⟩
          = ⟨fusc(n), fusc(n)+fusc(n+1)⟩
          = ⟨u, u+v⟩      where ⟨u,v⟩=g(n)
g(2n+1)   = ⟨u+v, v⟩      where ⟨u,v⟩=g(n)
```

The above linear recursion can be transformed into an iterative program using the same schema-flowchart equivalence used in Section 1.5 for recurrence relations. We get:

```
                                   { $n \geq 0$ }
if n=0 then B(0), ℓ:=0,0
else begin m, i, ℓ:=n, 0, ⌊log n⌋;
      while i<ℓ do B(i), m, i:=rem(m//2), m//2, i+1 od end;
   {B[ℓ..0] stores the binary digits of n. The most significant one is B(ℓ)}
⟨u,v⟩, p := ⟨0,1⟩, ℓ;
{ ⟨u,v⟩ = ⟨fusc(m), fusc(m+1)⟩  and
      m = if p≥ℓ then 0 else $\sum_{i=p+1}^{\ell} B(i)\cdot 2^{i-p-1}$ }

while p≥0 do if B(p)=0 then v := u+v else u := u+v;
             p := p-1 od
                    { ⟨u,v⟩ = ⟨fusc(n), fusc(n+1)⟩ = g(n) }
```

Note. // denotes the integer division.

The above program has a logarithmic running time as Prof. Dijkstra's one, (given below). Unfortunately it uses logarithmic space requiring the array B (although that is not a great disadvantage, because B stores the binary digits of n and we can assume that they are given us for free).

In [Pettorossi 84] it is shown that, by applying the generalization strategy w.r.t. the initial values of u and v, we can obtain, after suitable transformations and instantiations, the following program:

```
                              { n ≥ 0 }
<u,v>,q := <0,1>,n;
while q>0 do if even(q) then v := u+v else u := u+v;
             q := q//2 od
                           { u = fusc(n) }
```

which is exactly the one suggested by Prof. Dijkstra.

1.8.4 Limitations of the tupling strategy

In this subsection we will show a "limitation result". It holds for the tupling strategy as well as for any other method which avoids redundancy in recursive calls, under the hypothesis that only a bounded amount of memory cells can be used.

That hypothesis is not very restrictive because, if we allow for unbounded number of memory cells, we can avoid redundant computations in recursive calls by implementing the recursion using stacks and tables of already computed values, as done in the memo-function technique [Michie 68]. The description of more advanced methods (such as the "overtabulation" and "exact tabulation") which make use of an unbounded number of memory cells can be found in [Bird 80].

Let us consider the following recursive program schema:

S_2: $f(x) = a(x)$ $\qquad$ *if* $p(x)$

$\quad f(x) = b(x, f(c(x)), f(d(x)))$ $\qquad$ *otherwise*

Theorem 2: For any integer n there exists an interpretation for the function symbols and predicate symbols of S_2 such that f(x) cannot be computed using less than n memory cells.

Proof: (see [Paterson and Hewitt 70]).

The above theorem implies that for any schema S_n for $n \geq 2$ (see Sect. 1.8.3) the tupling strategy cannot avoid redundancy in recursive calls, because with the tupling strategy, we are able to memorize only a fixed number of intermediate results (corresponding to the values of the functions calls in each cut).

Even if we assume commutative redundancy (i.e. $c(d(x)) = d(c(x))$ whenever $p(x)$ is false), the tupling strategy cannot completely avoid redundancy. The statement of this fact is in [Cohen 83], but his inductive proof (see pages 289-290) is not satisfactory. As it has been presented, his proof would be valid for asserting the need of n memory cells for evaluating the function $f(x)$ defined by the schema S_2 under the assumption (which we will call "A1") that "$c(c(x)) = d(c(x))$ whenever $p(x)$ is false".

This is not the case because any minimal compressed descent dag for S_2 under the assumption A1 is of the form $t(k)$ for some k (see fig. 1-17).

$t(1) = o$ and $t(n) =$

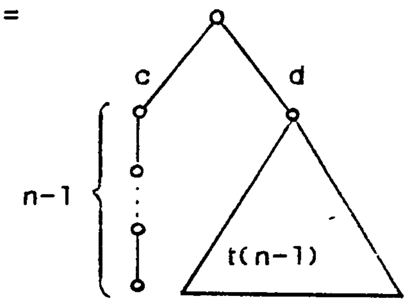


Figure 1-17: Dags $t(n)$ for $n \geq 1$.
n is the number of nodes from the root to the leftmost leaf.

and any $t(n)$ can be evaluated using at most two memory cells.

In the remaining part of this subsection we give a non-inductive proof of the above theorem under the commutative assumption that "$c(d(x)) = d(c(x))$ whenever $p(x)$ is false".

As it has been pointed out by [Paterson and Hewitt 70] the proof can be described using a <u>pebble game</u> over the minimal compressed descent dag of $f(x)$.

Let us consider the indexed set of grids as defined in fig. 1-18.

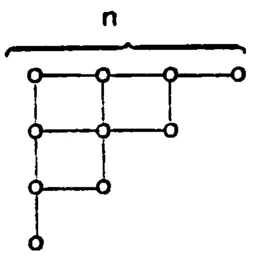
grid(1) = o and grid(n) = 

Figure 1-18: Definition of grid(n) for $n \geq 1$.
n is the number of the nodes in the first row of grid(n).

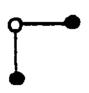
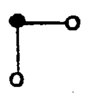
If the left-upper corner (called root) corresponds to the $f(x_0)$ call and a vertical [or horizontal] arc connects the node for f(y) with the node for f(d(y)) [or f(c(y))] then grid(n) is the minimal compressed descent dag for the <u>free interpretation</u> of S_2, called I_n, s.t. $p(c^i d^j x_0)$ is true iff $i+j \geq n$ for the given x_0. In that case, in fact, the nodes in the diagonal side of grid(n) correspond to base cases.

(For the notion of "free interpretation" see [Paterson and Hewitt 70]).

Notice that the m-dags are in the form of grids because of the commutativity hypothesis.

The pebble game over a grid(n) is described by the following rewrite rules:

R1. For any leaf: o $\rightarrow$ ●

R2. For any subconfiguration:  $\rightarrow$ 

The minimum number of pebbles necessary for placing one pebble in the root is equal to the minimum number of memory cells necessary to compute $f(x_0)$ (see [Paterson and Hewitt 70]).

An <u>arc</u> is a vertical or horizontal line connecting two adjacent nodes of a grid. Arcs are oriented "left-to-right" and "top-down".

A <u>path</u> is a sequence of adjacent arcs, which agrees with their orientation.

A pebbled grid is <u>closed</u> if for every path from the root to a leaf there exists one pebble in one of its nodes.

Notice that during the game a closed grid may become open. During the game a node may be pebbled more than once.

Lemma 3: The only move which makes an open grid to become closed is R1.

Proof: Immediate.

Lemma 4: During the game, whenever an open grid(n) becomes closed we need at least n pebbles in that closed grid.

Proof: Let us denote the nodes in the grid(n) by the <row, column> indexes $\{\langle i,j\rangle \mid 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant n,\ i+j \leqslant n+1\}$. Suppose that the move which closes an open grid(n), puts a pebble in the leaf <i, j>. Let us consider a path from the root, which does not have any pebble prior to that move. That path crosses i−1 columns and j−1 rows (see fig. 1−19 where the crossings have been denoted by arrows).

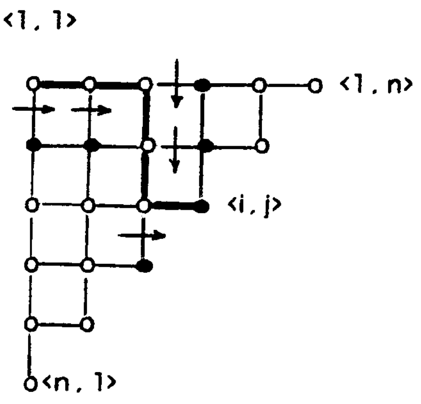


Figure 1−19: Closing grid(n) by a pebble in <i, j>.

Since all paths, branching off that path should be closed, we must intersect with a pebble all columns below that path and all rows to the right of that path and we need the pebble in <i, j>. Thus we need at least $(i-1)+(j-1)+1 = i+j-1$ pebbles to close an open grid(n).

Since $n = i+j-1$ for any leaf position <i, j>, the proof of Lemma 4 is completed.

A method (let us call it M) of using exactly $i-1$ pebbles to close the $i-1$ columns below an open path which intersects them is the following:

- Place a pebble anywhere in the leftmost column;
- Let $\langle r,c\rangle$ be the $\langle$row, column$\rangle$ position of the pebble placed in the c column. Place the pebble in column $c+1$ in the row position r' s.t. $r' \geq r-1$.

Analogous method can be defined for using exactly $j-1$ pebbles to close the $j-1$ rows on the right of an open path.

Two ways of applying the above mentioned method are the following:

1. place the pebbles on the leaves. (In this case the method M is applied with $r' = r-1$ for any column position c.)

2. place the pebbles on the "highest" row position below the open path and on the "leftmost" column position on the right of the open path (In this case the method M is applied with $r' \geq r$). We depicted this second way in the figure 1-19.

Lemma 5: If an open grid(n) is closed and it has $m \geq n$ pebbles, we can conclude the game of placing a pebble on the root of grid(n) using no more than m pebbles.

Proof: By induction on n. For $n=0$ it is obvious. Suppose the lemma true for $n-1$, we show it for n. Suppose the open grid(n) has been closed using $m \geq n$ pebbles.

Let us consider the following sequence of grids, all subgrids of grid(n): grid(n-1) with root $\langle 1,2\rangle$, grid(n-2) with root $\langle 2,2\rangle$, ..., grid(1) with root $\langle n-1,2\rangle$.

When the grid(n) becomes closed, say after the move r, grid(n-1) with root $\langle 1,2\rangle$ either has been closed by that same move r or it was already closed. By the lemma 4 and the induction hypothesis we have at least $n-1$ pebbles for closing the grid(n-1) with root $\langle 1,2\rangle$. We can then conclude the game for closing such grid(n-1) placing 1 pebble in position <1,2>. We have then at least n-2 pebbles at our disposal for concluding the game for the grid(n-2), by applying the induction hypothesis again. Eventually we get to the situation where grid(n-1),....,grid(1) are all closed with pebbles in their roots (see the figure 1-20). We used n-1 pebbles.

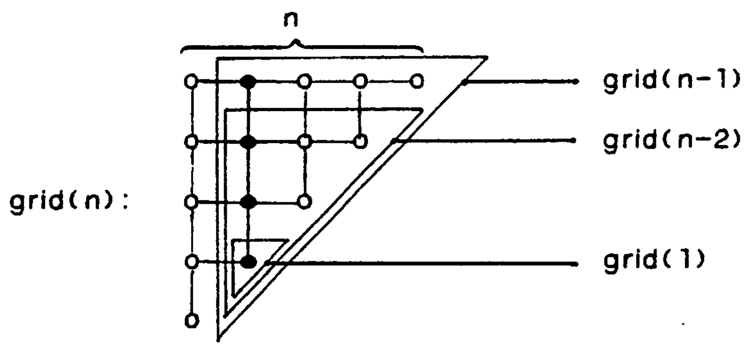


Figure 1-20: Closing n-1 grids in grid(n).

Since grid(n) is closed and we have at our disposal m≥n pebbles, there must be at least one pebble in the first column. We will then be able to conclude the game for grid(n) applying rule R2 (one or more times) and therefore using no more than m pebbles.

Lemma 6: The number of memory cells necessary to compute f(x) of S_2 under the free interpretation I_n and the commutative hypothesis, is not less than the minimum number of pebbles placed in a closed grid(n) which was open in the previous move.

Proof: The number of memory cells necessary to compute f(x) of S_2 under I_n is at least equal to the minimum number of pebbles necessary i) to close an open grid(n) and ii) to finish the pebble game for grid(n), keeping it always closed. By Lemma 5 condition ii) can be dropped.

From the previous lemmas 4 and 6 we conclude that for any n there exists a commutative interpretation for S_2 such that n memory cells are needed for evaluating f(x) and avoiding redundancy. This concludes the proof of Cohen's statement ([Cohen 83] page 288) we promised at the beginning of this subsection.

As a final remark, we will show the following upper bound result.

Theorem 7: n pebbles suffice to pebble the root of grid(n) for any $n \geq 1$.

Proof: By complete induction on n. For n = 1 it is obvious. Assume that the theorem holds for any $k \leq n-1$ then we can shows that it holds for k = n as follows. Let us refer to the figure 1-20. Starting from an empty grid(n) we can place a pebble in <1,2> using n-1 pebbles. With the remaining n-2 pebbles we place a pebble in <2,2> and so on (by complete induction). Therefore we can place n-1 pebbles as shown in the figure 1-20 using n-1 pebbles only. One extra pebble place in <1,1> allows us to conclude the game having eventually only 1 pebble in <1,1>.

1.9 The tupling strategy compared with other methods of eliminating redundancy in recursive calls

In the literature we can find various methods for eliminating redundant computations in recursive programs. We can divide them into two groups: the first one for which the number of extra memory cells for storing already computed values is input-independent (Group G1), and the second one for which the number of extra memory cells is input-dependent (Group G2).

Group G1 includes:

1.1 Variable small-table heuristic methods [Hilden 76]

1.2 Descent-conditions-based strategies for:

- 1.2.1 explicit redundancy
- 1.2.2 common generator redundancy
- 1.2.3 commutative periodic redundancy [Cohen 83]

1.3 Tupling Strategy [Pettorossi 77]

Group G2 includes:

2.1 Memo-functions [Michie 68]

2.2 Overtabulation techniques [Bird 80]

2.3 Exact tabulation techniques [Bird 80]

2.4 Descent-conditions-based strategies for commutative redundancy [Cohen 83].

We would like to show that the tupling strategy is indeed a generalization of the methods listed in Group G1. The methods in Group G2 allow for an unbounded number of memory cells and therefore comparing the tupling strategy with them is not particularly significant.

As far as Hilden's methods are concerned, we will only say that they are of a heuristic nature and we can never be sure of avoiding redundant computations when applying them.

The comparison in which we are most interested is the one between the tupling strategy and the descent-condition-based strategies of group G1 [Cohen 83]. In what follows we will show that the tupling strategy is indeed a <u>proper extension of Cohen's methods</u>.

Our work was done independently from Cohen's and this section is devoted to clarify the relationship between his results and ours.

Let us consider the following schema:

S_2: f(x) = a(x) <u>if</u> p(x)

f(x) = b(x, f(c(x)), f(d(x)) <u>otherwise</u>

<u>Tupling vs. explicit redundancy</u>.

There is explicit redundancy if c(x) = d(x). In that case the m-dag for f(x) is:

$f(x) \rightarrow f(c(x)) \rightarrow f(c^2(x)) \rightarrow \ldots$

A cut in m-dag is any function call of the form $f(c^i(x))$ for $i \geq 0$. Therefore, by a trivial application of the tupling strategy, we tuple one function only, say $f(c^i(x))$, and we express it in terms of the subsequent cut, $f(c^{i+1}(x))$, as follows:

$f(c^i(x)) = b(x, z, z)$ <u>where</u> $z = f(c^{i+1}(x))$ for $i \geq 0$

Step 5 of the application of the tupling strategy gives us:

f(x) = a(x) <u>if</u> p(x)

f(x) = b(x, z, z) <u>where</u> z = f(c(x)) <u>otherwise</u>

which is exactly the same program schema we get by applying Cohen's explicit redundancy strategy.

In order to compare the tupling strategy with the common generation redundancy and the commutative periodic redundancy strategies given in [Cohen 83] we need only to deal with the latter one because the former is a special case of it. However, in order to make the presentation as clear as possible, we will consider the two cases separately. We will also show some improvements to Cohen's methods.

Tupling vs. common-generator redundancy.

In the common-generator-redundancy we have that:

1. $c(x) = g^m(x)$ and $d(x) = g^n(x)$

   for some function $g(x)$, for some non-negative integers m and n

   and for all $x \in R$ (i.e. such that $p(x)$ is false).

2. the frontier condition holds for $\{g^i(x) \mid 0 < i \leq \max(m,n)\}$.

Without loss of generality we may assume that m and n are relatively prime. The descent tree generated by $f(x)$ is represented in figure 1-21.

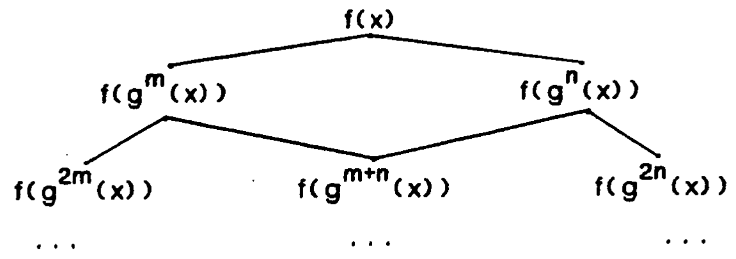


Figure 1-21: Descent tree for common generator redundancy.

In the expressions of the form $f(g^p(x))$ occurring in it, we have $p = im+jn$ for some integers i,j. Since every natural number greater than $(m-1)(n-1)-1$ can be expressed as $im+jn$, for integers i,j and m and n relatively prime [Cohen 80], and m and n are typically small, at the nodes of the descent tree we have almost all $f(g^p(x))$ for $p \geq 0$.

Therefore the computation of $f(x)$ can be performed according to the following m-dag:

$f(x) \longrightarrow f(g(x)) \longrightarrow f(g^2(x)) \longrightarrow f(g^3(x)) \longrightarrow \ldots \longrightarrow f(g^t(x))$,

where we know that a recurrence relation of order k, with $k = \max(m,n)$, holds among the nodes of the m-dag, because $f(x) = b(x, f(g^m(x)), f(g^n(x)))$. In the above m-dag t is such that $p(g^t(x))$ = true and

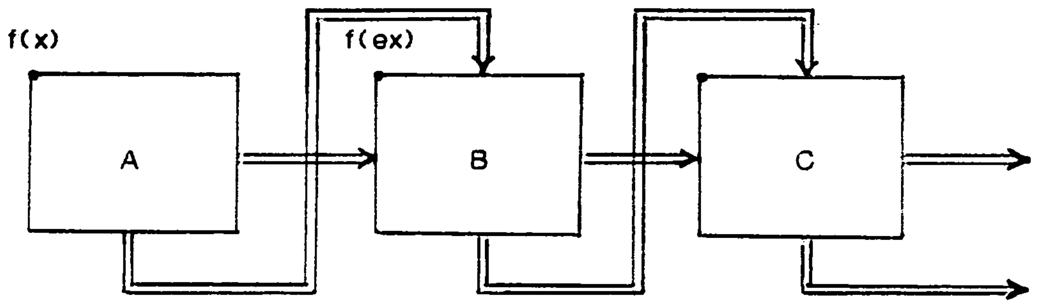
$\forall\, q < t \quad p(g^q(x)) = \text{false}$. Thus $f(g^t(x)) = a(g^t(x))$. A cut for the above m-dag can be chosen to be a sequence of k adjacent function calls. We tuple them together and we can express $\langle f(g^q(x)), \ldots, f(g^{q-k+1}(x))\rangle$ in terms of $\langle f(g^{q-1}(x)), \ldots, f(g^{q-k}(x))\rangle$ using the second equation of S_2.
Notice that the frontier condition ensures that the recursion of the tupled functions is well-founded. In fact the frontier condition tells us that if $p(g^t(x))$ is true, then also $p(g^{t+i}(x))$ is true for $0 < i \leq \max(m,n)$. Therefore we can start the recursion from the k-tuple: $\langle f(g^{t-1}(x)), \ldots, f(g^{t-k}(x))\rangle$. In fact, since $k=\max(m,n)$ we are sure that the frontier condition holds for all functions in that k-tuple, if it holds for $g^t(x)$.

The above k-tuple of values corresponds exactly to the values BACKg[i] for $i=1,\ldots,\max(m,n)$ of Cohen's solution (see [Cohen 83] page 277). In our solution we avoid the need for the extra location BACKg[0], which is used by Cohen.

Let us call "permanent" memory cells the ones needed when entering and exiting a recursive call of the tupled function, and "temporary" memory cells the ones needed during a recursive call.
The interesting point of the application of the tupling strategy is not the use of one permanent memory cell less (i.e. BACKg[0]), but the fact that we are able to distinguish between the need for permanent and the need for temporary memory cells (or, as in Cohen's terminology, non-local and local memory cells).

Tupling vs. commutative periodic redundancy.
In the commutative periodic redundancy we have that:
1. $c^i(x) = d^j(x)$ for some i and j for all $x \in R$
2. $c(d(x)) = d(c(x))$ for all $x \in R$
3. the frontier condition holds for $\{ c^m d^n(x) \mid 0 \leq m < i,\ 0 \leq n < j,\ m+n > 0 \}$.
Let $e(x)$ be $c^i(x)$ (which is also equal to $d^j(x)$).

The m-dag for the computation of f(x) looks like this [Cohen 83]:



where each rectangle is of the form shown in fig. 1-22.

$$\begin{tikzcd}
f(y) \arrow[r] \arrow[d] & f(c^0d^1y) \arrow[r] \arrow[d] & \cdots \arrow[r] & f(c^0d^{j-1}y) \arrow[r] \arrow[d] & {} \\
f(c^1d^0y) \arrow[r] \arrow[d] & f(c^1d^1y) \arrow[r] \arrow[d] & \cdots \arrow[r] & f(c^1d^{j-1}y) \arrow[r] \arrow[d] & {} \\
\cdots & \cdots & \cdots & \cdots & \\
f(c^{i-1}d^0y) \arrow[r] \arrow[d] & f(c^{i-1}d^1y) \arrow[r] \arrow[d] & \cdots \arrow[r] & f(c^{i-1}d^{j-1}y) \arrow[r] \arrow[d] & {} \\
{} & {} & & {} &
\end{tikzcd}$$

Figure 1-22: A "rectangle" for the commutative periodic redundancy. y ranges over e^kx for $k \geq 0$.

Therefore a cut of the m-dag can be obtained by tupling together the following function calls, i.e. the ones which occur in a rectangle:

$$\begin{array}{llll}
C(x) = \langle f(x), & f(c^0d^1x), & \ldots & f(c^0d^{j-1}x), \\
f(c^1d^0x), & f(c^1d^1x), & \ldots & f(c^1d^{j-1}x), \\
\ldots & \ldots & & \ldots \\
f(c^{i-1}d^0x), & f(c^{i-1}d^1x), & \ldots & f(c^{i-1}d^{j-1}x)\rangle.
\end{array}$$

We can express the function calls occurring in a cut in terms of the function calls occurring in a subsequent cut, as was shown by [Cohen 83] page 283.

We will not go into the details here. The interested reader may refer to Cohen's paper. In that paper the redundancy of recursive calls is avoided by keeping in permanent memory cells the values of the function calls which realize the cut C(x). Some extra $i \times j$ temporary memory cells are needed: they are the array ARGUMENT[0: (i−1), 0: (j−1)].

Notice that the frontier condition, as in the case of the common-generator-redundancy, ensures that the recursive definition of C(x) is well-founded. In fact, if p(x) = true, i.e. x is a base case, then all arguments of the function calls in C(x), i.e. $c^m d^n(x)$ for $0 \leq m < i$, $0 \leq n < j$, $m+n > 0$, are also base cases. In other words, for any m and n satisfying the above constraints, if $f(c^m d^n(x))$ is defined then it is equal to $a(c^m d^n(x))$, by the frontier condition.

In the case of commutative periodic redundancy we can apply the tupling strategy in a different way, by using cuts corresponding to the first row and the first column of the above mentioned rectangles. This is done by defining the following function:

$$\begin{aligned} D(x) = \langle f(x), \quad & f(c^0 d^1 x), \ \dots \ f(c^0 d^{j-1} x), \\ & f(c^1 d^0 x), \\ & \dots \\ & f(c^{i-1} d^0 x) \rangle. \end{aligned}$$

In order to elucidate the correspondence with Cohen's solution, let us store the values $f(x), f(c^0 d^1 x), \dots, f(c^0 d^{j-1} x)$ in the array BACKd[0: (j−1)] and the values $f(c^1 d^0 x), \dots, f(c^{i-1} d^0 x)$ in the array BACKc[1: (i−1)] (Cohen uses instead the array BACKc[0: (i−1)]). We use $i \times j$ temporary memory cells of the array ARGUMENT[0: (i−1), 0: (j−1)] and one extra temporary cell Temp, which initially gets the value BACKd[0] (which is also equal to BACKc[0]).

We can express D(x) in terms of D(e(x)), using the following program written in an Algol-like notation.

```
Temp:=BACKd[0];
for M from (i-1) step -1 until 0 do
   begin
    BACKd[(j-1)]:=
      if p(ARGUMENT[M,(j-1)])
        then a(ARGUMENT[M,(j-1)])
        else b(ARGUMENT[M,(j-1)], BACKd[(j-1)],
                                          if M=0 then Temp else BACKc[M]);
    for N from (j-2) step -1 until 0 do
      BACKd[N]:=
        if p(ARGUMENT[M,N])
          then a(ARGUMENT[M,N])
          else b(ARGUMENT[M,N],BACKd[N],BACKd[N+1]);
    if M≠0 then BACKc[M]:=BACKd[0]
   end
```

The reader may refer to Cohen's paper for an explanation of the above program and why it indeed computes D(x) from D(e(x)). That program is a variant of the one used in Cohen's "improved solution" (see page 284 [Cohen 83]). We only added the pieces of code which are underlined. Those additions are needed because we are using the temporary cell Temp instead of the permanent cell BACKc[0].

As we already mentioned, the important issue here is not the saving of one permanent memory cell, but the fact that, through the application of the tupling strategy, we are able to distinguish between the need of permanent cells and the need of temporary cells.

So far we have shown that Cohen's methods for avoiding redundancy in recursive calls can be considered as particular cases of application of the tupling strategy. We will now show that the tupling strategy is a <u>proper extension</u> of those methods. In order to do so we have only to refer to the Towers of Hanoi example for which the tupling strategy can be applied as shown in subsection 1.8.3. but none of Cohen's method can (as already stated in [Cohen 83] page 295).

1.10 The tupling strategy and the use of data structures

We would like to present another important use of the tupling strategy for improving programs via transformations.
The basic idea consists in tupling together those functions whose evaluation requires the same data structure. It turns out that if during the computation we reduce the number of "passes" over a given data structure, we improve the time $\times$ space performances, because we can release storage sooner. Some preliminary results in that direction were published in [Pettorossi 77], and they were applied in [Feather 79]. Related work can be found in [Wadler 81, Scherlis 80]: their methods also allow to improve the time$\times$space performances.

Instead of defining a formal framework and formally proving the properties of that use of the tupling strategy, we prefer to give two examples from which the reader may gain deeper insights, together with concrete programming suggestions. The definition of a formal framework is left for future investigation.

The first introductory example is about the computation of the difference between the smallest and the biggest leaf of a binary tree.
The following HOPE-like program [Burstall, MacQueen and Sannella 80] solves the problem.

```
data btree(num) == niltree ++ tip(num) ++ btree(num)Δbtree(num)
dec minℓ: btree(num) → num
--- minℓ(niltree)    = +∞
--- minℓ(tip(n))     = n
--- minℓ(t1Δt2)      = min(minℓ(t1), minℓ(t2))
dec maxℓ: btree(num) → num
--- maxℓ(niltree)    = -∞
--- maxℓ(tip(n))     = n
--- maxℓ(t1Δt2)      = max(maxℓ(t1), maxℓ(t2))
dec task: btree(num) → num
--- task(t) = maxℓ(t) - minℓ(t)
```

Since minℓ(t) and maxℓ(t) both have the argument t, even though it is not a common subexpression, just a simple variable, we tuple them together and we define the function p(t) = <minℓ(t), maxℓ(t)>.
By instantiation we get:

```
dec p: btree(num) → num×num
--- p(niltree)    = <+∞, -∞>
--- p(tip(n))     = <n, n>
--- p(t1Δt2)      = <min(a1,a2), max(b1,b2)> where <a1,b1>=p(t1)
                                                  <a2,b2>=p(t2)
--- task(t) = π2(p(t)) - π1(p(t))
```

One can easily see that the derived program is more time×space efficient than the given one, because after the evaluation of p(t1) the storage used by t1 can be released, while after the evaluation of maxℓ(t1), we cannot, because t1 is necessary for computing minℓ(t1).

Let us now give a more interesting (though more complex) example concerning the destructive evaluation problem [Pettorossi 78].

Given a term to be evaluated, we want to mark the occurrences of its function symbols so that, during evaluation, the cells storing the arguments of the marked functions can be released without effecting the correctness of the result.

Let $A = \{f, g, h, \dots\}$ be a (finite or denumerably infinite) ranked alphabet and $V = \{x, y, z, \dots\}$ a (finite or denumerably infinite) set of variables. We define the set $T_{A,V}$ of <u>A-terms</u> over V as follows:

- for any $x \in V$ $\quad <x> \in T_{A,V}$ (variable terms)
- for any $f \in A$ with rank (or arity) n, for any $t1, \dots, tn \in T_{A,V}$
  $<f\ t1 \dots tn> \in T_{A,V}$ (application terms)

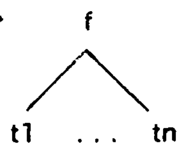
For simplicity we also write x instead of $<x>$ for any $x \in V$.

Let us consider an <u>evaluation function</u> "eval" from $T_{A,V}$ to a given domain D. We assume that "eval" uses a variable-environment $\rho : V \longrightarrow D$ for assigning values in D to variables and a function-environment for assigning functions to ranked symbols in A. We also assume that "eval" corresponds to the <u>call-by-value</u> evaluation rule and it evaluates its arguments in the <u>left-to-right</u> order, with the exception that <u>variable terms are evaluated after application terms</u>.

For instance, for the term $< f\ x < g\ x\ z > < h\ y > >$ the order of evaluation of the subterms is denoted by the integers associated with their leftmost symbols as follows: $< f7\ x6 < g3\ x1\ z2 > < h5\ y4 > >$, where subterms with smaller numbers are evaluated first.

In intuitive terms, the evaluation proceeds as a "left-to-right" and "bottom-up" visit of the tree one can associate with a given term by the following correspondence:

$< f\ t1 \dots tn > \longleftrightarrow$



Let us now extend our alphabet A so that for each symbol $a \in A$ we have a "marked symbol" $\bar{a}$ (with the same arity) also belonging to A.

We extend the "eval" function as follows:

for any f (with arity n) $\in$ A, for any t1, ..., tn $\in T_{A,V}$ after the evaluation of $<\bar{f}$ t1...tn$>$ we have that $\forall$ x $\in$ V s.t. $\exists$ ti $1 \leq i \leq n$ and ti = x, $\rho(x)$ is undefined. In other words, $<\bar{f}$ t1...tn$>$ is the "destructive version" of $<$ f t1...tn $>$ [Schwarz 78, Pettorossi 78], i.e. it destroys the values of all variables occurring as immediate arguments of $\bar{f}$.

Example 10.

$< \bar{f}$ x $<$ g x z $>$ $<$ $\bar{h}$ y $>$ $>$, $<$ f $<$ $\bar{g}$ x z $>$ y $<$ h y $>$ $>$ are terms correctly marked. The first one is maximally marked, while the second is not (because the leftmost f also could have been marked).

$<$ f x $<$ $\bar{g}$ x z $>$ y $>$ is not correctly marked because $\bar{g}$ destroys the value of x which will be used by the function f.

The following program Prog1 written in a HOPE-like style optimally marks a given $T_{A,V}$-term.

Program Prog1

```
data term == vn(name) ++ an(name, list term)


dec mark: term → term
--- mark(n) ⇐ mark2(n, ∅)


dec mark2: term × set name → term
--- mark2(vn(name1), v)       ⇐ vn(name1)
--- mark2(an(name1, ℓn), v)   ⇐ an(name1m, mark2list(ℓn, v ∪ w))
                                 where name1m = if v ∩ w = ∅ then $\overline{name1}$
                                                  else name1
                                 where w = singlevar(ℓn)
```

```
dec singlevar: list term ⟶ set name
--- singlevar(nil)                    ⟸ ϕ
--- singlevar(vn(name1)::tℓ)          ⟸ {name1} ∪ singlevar(tℓ)
--- singlevar(an(name1,tℓ1)::tℓ)      ⟸ singlevar(tℓ)

dec mark2list: list term × set name ⟶ list term
--- mark2list(nil,v)                  ⟸ nil
--- mark2list(vn(name1)::tℓ,v)        ⟸ vn(name1)::mark2list(tℓ,v)
--- mark2list(an(name1,tℓ1)::tℓ,v)    ⟸ mark2(an(name1,tℓ1),
                                                  v ∪ varinapp(tℓ))
                                         ::mark2list(tℓ,v)

dec varinapp: list term ⟶ set name
--- varinapp(nil)                     ⟸ ϕ
--- varinapp(vn(name1)::tℓ)           ⟸ varinapp(tℓ)
--- varinapp(an(name1,tℓ1)::tℓ)       ⟸ allvar(tℓ1) ∪ varinapp(tℓ)

dec allvar: list term ⟶ set name
--- allvar(nil)                       ⟸ ϕ
--- allvar(vn(name1)::tℓ)             ⟸ {name1} ∪ allvar(tℓ)
--- allvar(an(name1,tℓ1)::tℓ)         ⟸ allvar(tℓ1) ∪ allvar(tℓ)
```

Let us make a few comments to help the readability of our program.

1. vn and an are constructors for variable-terms and application-terms respectively. name is supposed to be a primitive data type.

2. The second argument of mark2 is the set of variables which are required in future computations.

3. singlevar, given a list of terms, builds the set of variable terms occurring at the top level of that list.

4. mark2list behaves as mark2 for a list of terms.

5. varinapp, given a list of terms, builds the set of all variables occurring in application-terms of that list.

6. allvar, given a list of terms, builds the set of all variables occurring in those terms.

For instance, if t = [x, < f y < h z > >] then singlevar(t) = { x }, varinapp(t) = { y, z }, allvar(t) = { x, y, z }.

Looking at the definition of mark2 we see that both functions singlevar and mark2list use the list ℓn. Therefore we tuple them together and we get:

H(ℓn, v) = <singlevar(ℓn), mark2list(ℓn, v $\cup$ singlevar(ℓn))>

We obtain:

```
H(nil, ∅)                 ⟸ <∅, nil>
H(vn(name1)::tℓ, v)       ⟸ <{name1} ∪ singlevar(tℓ), vn(name1) ::
                                 mark2list(tℓ , v ∪ {name1} ∪ singlevar(tℓ))>
                          ⟸ < {name1} ∪ a, vn(name1)::b >
                                  where <a, b> = H(tℓ, v ∪ {name1})
H(an(name1, tℓ1)::tℓ, v)   ⟸ <singlevar(tℓ),
                                 mark2(an(name1, tℓ1), v ∪ singlevar(tℓ)
                                   ∪ varinapp(tℓ))::mark2list(tℓ, v∪singlevar(tℓ))>
                          ⟸ <a,  mark2(an(name1, tℓ1), v∪a∪varinapp(tℓ))::b>
                                  where <a, b> = H(tℓ, v)
```

In the definition of H, both functions varinapp and H use the list tℓ. We tuple those functions together and we get (after "flattening" the pair of functions in H):

K(tℓ, v) = <singlevar(tℓ), mark2list(tℓ, v $\cup$ singlevar(tℓ)), varinapp(tℓ)>

We obtain:

```
K(nil, ∅)                 ⟸ <∅, nil, ∅>
K(vn(name1)::tℓ, v)       ⟸ < {name1} ∪ a, vn(name1)::b, c>
                                  where <a, b, c> = K(tℓ, v ∪ {name1})
```

K(an(name1, tℓ1) : : tℓ, v) $\Leftarrow$ <a, mark2(an(name1, tℓ1), v $\cup$ a $\cup$ c) : : b,
allvar(tℓ1) $\cup$ c> <u>where</u> <a, b, c> = K(tℓ, v)

By unfolding in the equation for K(an(name1, tℓ1) : : tℓ, v) the definition of mark2 we get the expression:

<a, an(name1m, mark2list(tℓ1, v $\cup$ a $\cup$ c $\cup$ w)) : : b, allvar(tℓ1) $\cup$ c>

<u>where</u> name1m = <u>if</u> (v $\cup$ a $\cup$ c) $\cap$ w = $\emptyset$ <u>then</u> $\overline{\text{name1}}$ <u>else</u> name1

<u>where</u> w = singlevar(tℓ1)

<u>where</u> <a, b, c> = K(tℓ, v)

We see that both mark2list and allvar visit the same data structure tℓ1. Therefore we are led to the following definition:

L(ℓn, v) =

<singlevar(ℓn), mark2list(ℓn, v $\cup$ singlevar(ℓn)), varinapp(ℓn), allvar(ℓn)>

and we obtain:

L(nil, v) $\Leftarrow$ <$\emptyset$, nil, $\emptyset$, $\emptyset$>

L(vn(name1) : : tℓ, v) $\Leftarrow$ <{name1} $\cup$ a, vn(name1) : : b, c, {name1} $\cup$ d>
<u>where</u> <a, b, c, d> = L(tℓ, v $\cup$ {name1})

L(an(name1, tℓ1) : : tℓ, v) $\Leftarrow$ <a, an(name1m, b1) : : b, d1 $\cup$ c, d1 $\cup$ d>
<u>where</u> name1m = <u>if</u> (v $\cup$ a $\cup$ c) $\cap$ a1 = $\emptyset$
<u>then</u> $\overline{\text{name1}}$ <u>else</u> name1
<u>where</u> <a1, b1, c1, d1> = L(tℓ1, v $\cup$ a $\cup$ c)
<u>where</u> <a, b, c, d> = L(tℓ, v)

The resulting final program Prog2 is as follows:

Program Prog2

```
data term == vn(name) ++ an(name,list term)


dec mark: term → term
        ...(defined as above)...
dec mark2: term × set name → term
--- mark2(vn(name1),v)          ⇐ vn(name1)
--- mark2(an(name1,ℓn),v)       ⇐ an(name1m,π2(L(ℓn,v)))
                                   where name1m = if v ∩ π1(L(ℓn,v))=∅
                                                  then ‾name‾ else name1
dec L: list term × set name → (set name × list term × set name × set name)
       ...(defined as above)...
where πi denotes the i-th projection function, for i=1,...,4.
```

The Hope implementation [Burstall, MacQueen and Sannella 80] of Prog1 and Prog2 on the DEC-10 at Edinburgh University confirms the expected improvements of the program performances.

Now we will make a few remarks about the derivation from Prog1 to Prog2. We leave to the reader the task of abstracting from those remarks the related general properties of the tupling strategy.

Remark 1. The recursive structure of the functions in the derived program Prog2 is similar to the one of the given Prog1. For example in Prog2 the definition of L(an(name1,tℓ1)::tℓ,...) is in terms of L(tℓ1,...) and L(tℓ,...), and in Prog1 the definition of allvar(an(name1,tℓ1)::tℓ) is also in terms of allvar(tℓ1) and allvar(tℓ).

Remark 2. As we already noticed, the use of the tupling strategy may force us to compute values not strictly necessary.

For instance in the definition of L one sees that it is not necessary to compute the value of c1. As usual this inconvenience may be avoided in two ways:

- by applying a call-by-need (meta) rule of evaluation when evaluating tuples, or
- by explicitly defining some suitable auxiliary functions. This second way can be implemented in our case as follows.

The computation of the c1 component in the definition of L is avoided by writing:

"where <a1, b1, d1> = M(tℓ1, v ∪ a∪ c)"

instead of: "where <a1, b1, c1, d1> = L(tℓ1, v ∪ a ∪ c)"

and defining the function M as follows:

M(tℓ1, v) =
<singlevar(tℓ1), mark2list(tℓ1, v ∪ singlevar(tℓ1)), allvar(tℓ1)>.

We have:

```
M(nil, v)                  ⇐ <φ, nil, φ>
M(vn(name1)::tℓ, v)        ⇐ <{name1} ∪ a, vn(name1)::b, {name1} ∪ d>
                              where <a, b, d> = M(tℓ, v ∪ {name1})
M(an(name1, tℓ1)::tℓ, v)   ⇐ <a, an(name1m, b1)::b, d1 ∪ d>
                              where name1m = if (v ∪ a ∪ c) ∩ a1 = φ
                                     then $\overline{\text{name1}}$ else name1
                              where <a1, b1, d1> = M(tℓ1, v ∪ a ∪ c)
                              where <a, b, c, d> = L(tℓ, v)
```

Making an extensive use of this second way of avoiding the above mentioned inconvenience, we may obtain a final program where in the mark2 definition in Prog2, instead of $\pi i(L(\ell n, v))$, we have $\pi i(H(\ell n, v))$ for i=1, 2.

The reader may easily verify that the final program includes, together with the definitions of the functions L and M, also the definition of H and K suitably "linked", so that for instance, in the definition of H we have the following equation:

H(an(name1,tℓ1)::tℓ,v) $\Leftarrow$ <a, mark2(an(name1,tℓ1), v $\cup$ a $\cup$ c)::b>
  where <a,b,c> = K(tℓ,v).

Analogously, in the definition of K we have the equation:

K(an(name1,tℓ1)::tℓ,v) $\Leftarrow$ <a, an(name1m,b1)::b,d1 $\cup$ d>
  where name1m = if (v $\cup$ a $\cup$ c) $\cap$ a1 = $\emptyset$
    then $\overline{\text{name1}}$ else name1
  where <a1,b1,d1> = M(tℓ1, v $\cup$ a $\cup$ c)
  where <a,b,c,d> = L(tℓ, v).

Remark 3. This remark concerns a property which is analogous to the "safety property" described in section 1.8.
The need to tuple together various functions can be discovered "incrementally" (by using the unfolding rule or symbolically evaluating programs), while new tupled definitions are synthesized from old ones. This is elucidated by the process we followed in the above example, starting from the function H and deriving the need of defining the function K and then the function L.
The process of tupling functions together is bound to terminate because the initial program has a finite number of defined functions, and new tupled functions are, so to speak, members of the powerset of the set of the given initial functions.

Remark 4. When applying the tupling strategy one should comply with the following "independence requirement": the functions which are tupled together should have arguments which (once instantiated) do not depend on other functions of the same tuple.
We may see the reason for that requirement considering program Prog1.
After noticing that both singlevar and mark2list visit the list ℓn we could have defined the function H as follows:
H(ℓn,z) = <singlevar(ℓn), mark2list(ℓn,z)>.
Unfortunately, this pairing does not satisfy our independence requirement.

because in Prog1 z is instatiated as "v $\cup$ singlevar(ℓn)" and singlevar(ℓn) is a component of the $\underline{H}$(ℓn, z).

Indeed it is impossible to write the third equation for $\underline{H}$. In fact:

$\underline{H}$(an(name1, tℓ1) : : tℓ, z) $\Leftarrow$ <a, mark2 (an(name1, tℓ1), z$\cup$varinapp(tℓ)) : : b>

$\underline{\text{where}}$ <a, b> = $\underline{H}$(tℓ, z)

and by unfolding the mark2 definition, we get:

$\Leftarrow$ <a, an (name1m, mark2list(tℓ1, z$\cup$varinapp(tℓ)

$\cup$singlevar(tℓ1))) : : b>

$\underline{\text{where}}$ <a, b> = $\underline{H}$(tℓ, z)

Now singlevar(tℓ1) is an argument of mark2list(tℓ1, . . .) and we cannot express them as components of the function $\underline{H}$(tℓ1, . . .) despite the fact that they both visit the list tℓ1. The composition strategy solves that problem as our previous definition of H shows.

1.11 Conclusions and some motivations for communications in applicative languages

In this first part of the thesis we considered a strategy called "tupling strategy" for transforming applicative programs, and making them more efficient. That strategy avoids redundant computations by the "synchronized evaluation" of a predefined set of functions. In that way intermediate computed values may be used in more than one evaluation, and data structures may be kept in memory for the minimum amount of time.

We showed that the use of the tupling strategy allows us to achieve very efficient programs at least for certain classes of problems (as for instance for the evaluation of linear recurrence relations).

We also showed that there are classes of functions for which the complete avoidance of repeated computations requires an unbounded number of function evaluations to be synchronized. Therefore, for those classes, the tupling strategy is not powerful enough. That fact provides the motivation of the second part of the thesis, which is concerned with various approaches for allowing "run-time communications" among function evaluations. (Applying the tupling strategy can be viewed as establishing "compile-time communications".)

In particular we envisage a computing environment where tasks are performed by a network of computing agents performing subtasks in a concurrent way. Those agents may communicate with each other and help each other towards the achievement of a common goal.

In the second part of the thesis we will consider a very simple way of establishing run-time communications. It consists in remembering already computed values and allowing computing agents to access them. This idea is taken from [Michie 68] and we will apply it also to the case where agents (or subtasks) are evoked by recursive functions calls.

Chapter 2

OPERATIONAL SEMANTICS OF MEMOFUNCTIONS

In this chapter of the thesis no new programming ideas are introduced, except for some suggestions concerning the program annotation methodology [Schwarz 82] (see Section 2.4) and communications among processes in applicative languages (see Section 2.7).
We give the definition of the operational semantics of a simple applicative language in which repeated evaluations of recursive calls are avoided using memofunctions [Michie 68].

We store the "memo information" in environments which during the computation are discarded as soon as possible. We destroy all argument-value pairs of the memo component of the environments when exiting the scope of the memofunction definitions.

We start off by presenting in Section 2.1 and 2.2 the method of "generalized inductive definitions" as used in [Plotkin 81], and we first apply it for describing the operational semantics of a simple applicative language without memofunctions. (Section 2.1 is an improved version of what can be found in [Plotkin 81] Chapter 3 and 4).
That semantics definition is later used for proving the correctness of the operational semantics of memofunctions given in Section 2.3.
In Section 2.4 we present that correctness proof by showing that the transformation of functions to memofunctions does not effect the result of the computation.

In Section 2.5 and in Appendix C we present some techniques for embedding generalized inductive definitions, also called "structural definitions", into formal theories.
Section 2.6 is devoted to the description of Prolog implementations of the operational semantics rules.

Finally, in Section 2.7 we introduce some work we have done on "annotations denoting communications" and "computing agents" in applicative languages.

2.1 Structural Operational Semantic Definition of the Language L

We will consider a simple recursive language L, whose abstract syntax is defined by the following basic and derived sets.

Basic Sets. Numbers	$m \in N = \{0, 1, \ldots\}$		(1)
Truthvalues	$t \in T = \{\text{true}, \text{false}\}$		(2)
Variables	$x, y, z, \ldots, f, g, \ldots \in Var$		(3)
Basic Operators	$bop_i \in Bop = \bigcup_{i=0}^{n} Bop_i$	(i is the arity)	(4)
Types	$\tau \in Typ = \{\text{int}, \text{bool}\}$		

Notes.

(1) For each basic set we give its name (e.g. Numbers), the metavariable ranging over it (e.g. m) and its symbol (e.g. N).

(2) Instead of "true" and "false" we may use "tt" and "ff".

(3) Variables have arities. We assume that individual variables (e.g. x, y, z, ...) have arity 0, and functional variables (e.g. f, g, ...) have arity 1. This last assumption is made only to simplify the notation. In Appendix B we will give the rules for extending the operational semantics to functions with arity $n \geq 0$.

(4) Basic operators have arities and we assume that bop_i has arity i. Bop_i is the set of all basic operators with arity i. The if-then-else construct is an element of Bop_3.

Derived Sets. Expressions $e \in Exp$

$$e ::= m \mid t \mid x \mid bop(e_0, \ldots) \mid \underline{if}\ e_0\ \underline{then}\ e_1\ \underline{else}\ e_2$$
$$\mid \underline{let}\ d\ \underline{in}\ e \mid f(e)$$

Definitions $d \in D$

$$d ::= x{:}\ \tau = e \mid f(x{:}\ \tau_0) : \tau_1 = e \mid \underline{rec}\ d \mid d_0\ \underline{and}\ d_1$$

We used the usual BNF notation. Thus, for instance, e ::= ... | x | ... means that any (individual) variable x ∈ Var is an expression. The same meaning can be expressed as follows: Exp = ... + Var + ... where + denotes the disjoint union of sets.

Example 11.

5+3 and if eq(2,y) then x else 3 are expressions in L.

eq denotes the equality predicate and we can write it as the infix =.

f(x: int) : int = plus(x,2) is a definition in L. □

In the language L we can write expressions which denote "programs" for computing the value of (possibly) recursively defined functions.

For instance, we can write:

let rec fact(x: int) : int = if x=0 then 1 else x · fact(x-1)

in fact(5)

which will be evaluated to 5!.

Note.

In order to define the operational semantics of the language L we will need to extend the set "Definitions" by the set "Env" of environments, ranged over by ρ. Environments occur only during evaluation.

2.1.1 Preliminary remarks and a simple example

Before presenting the operational semantics of L, using generalized inductive definitions [Shoenfield 67, Plotkin 81], let us make the following remarks.

(1) We will introduce as few auxiliary notions as possible. This should be contrasted, for instance, with the SECD machine approach [Landin 66] in which the operational semantics of an applicative language is given through the use of auxiliary concepts such as the stack S, the environment E, the

control C and the dump D. Following [Plotkin 81], we avoid the use of auxiliary concepts and we consider as primitive concepts only the expressions and the definitions to be evaluated. The operational steps for evaluating a given expression (or definition) will be represented by a syntactic modification of the expression (or definition) itself.

(2) The problems related to the parsing of expressions and definitions are avoided by considering only their abstract syntax.

(3) We will specify the transformations of the expressions (or definitions) in a structural way, using formal theories. Those theories have axioms which specify how transformations of "basic expressions" are to be realized, and rules of inference which define the transformations of "compound expressions" in terms of the transformations of the "component expressions" (Analogously for definitions).

(4) An important advantage of a structural approach to the operational semantics is that via generalized inductive definitions, it is possible to prove properties of programs using the structural induction method [Burstall 69]. Some examples are given in [Plotkin 81].

In what follows we will write $e_i \longrightarrow e_{i+1}$ to denote that there is a transition from the expression e_i to the expression e_{i+1}. We will also say that e_i rewrites into e_{i+1}, or e_i is transformed into e_{i+1}.

Often we will also use the λ-calculus terminology [Barendregt 81], which could be applied to any rewriting system. For instance we will use the concepts of "contraction", "redex", "reduction", etc.

Let us look now at an example of a very simple theory for specifying the operational semantics of arithmetic expressions, with + only.

We have:

i) the axiom schema:

$$+(m,n) \longrightarrow s \qquad \text{where } s=n+m.$$

which stands for all axioms which are its instances for any n, m $\in$ Numbers and

ii) the rule schema r:

$$\frac{e \longrightarrow e'}{+(\ldots, e, \ldots) \longrightarrow +(\ldots, e', \ldots)}$$

which stands for all its instances obtained by instantiating the expressions e and e' in any argument position.

Using those axioms and rules we may reduce the expression 3+(7+2) according to the diagram of fig. 2-1, where by the subdiagram of fig. 2-2 we mean that the transition e $\longrightarrow$ e' holds because sube $\longrightarrow$ sube' holds and we can apply rule r. A pictorial view of the transformations evoked by our semantic definitions can be given as in fig. 2-3.

$$\begin{tikzcd}
3 + (7+2) \arrow[r] & 3 + 9 \arrow[r] & 12 \\
7 + 2 \arrow[r] \arrow[u, "r"'] & 9 &
\end{tikzcd}$$

Figure 2-1: The evaluation of a simple expression.

$$\begin{tikzcd}
e \arrow[r] & e' \\
\text{sube} \arrow[r] \arrow[u, "r"'] & \text{sube}'
\end{tikzcd}$$

Figure 2-2: Deriving a transition from a sub-transition. sube [or sube'] is a subexpression of e [or e'].

We could express the meaning of that figure by saying that in order to justify a "surface transition" (e.g. $e_0 \longrightarrow e_1$) we need a (possibly empty) tree of subtransitions (e.g. $e_{01} \longrightarrow e_{11}, \ldots, e_{0n} \longrightarrow e_{1n}$) connected by applications of rules (e.g. $r_{01}, r_{02}, \ldots$) ending with "axiom transitions" (e.g. $e_{0n} \longrightarrow e_{1n}$).

$$\left[\begin{array}{ccccccccc}
e_0 & \longrightarrow & e_1 & \longrightarrow & e_2 & \longrightarrow & \ldots\ e_m & \longrightarrow & \ldots \\
 & \uparrow r_{01} & & \uparrow & & \uparrow & & \uparrow & \\
e_{01} & \longrightarrow & e_{11} & \ldots & & \ldots & & \ldots & \\
 & \uparrow r_{02} & & & & & & & \\
 & \ldots & & & & & & & \\
e_{0n} & \longrightarrow & e_{1n} & \ldots & & \ldots & & \ldots &
\end{array}\right.$$

Figure 2-3: A sequence of transitions.

In the figure 2-3 the tree of subtransitions is very "thin", because we considered rules with one premise only. More generally, a "surface transition" e $\longrightarrow$ e' is justified by rule r and k subtransitions, as the diagram of fig. 2-4 shows.

$$\left[\begin{array}{ccc}
 & e \longrightarrow e' & \\
 & \uparrow \text{ by } r & \\
e1 \longrightarrow e1' & \ldots & ek \longrightarrow ek'
\end{array}\right.$$

Figure 2-4: An elementary transition with more than one premise.

Notice that a transition e $\longrightarrow$ e' can be viewed as a theorem which is proved by applying rule r, once we proved the transitions e1 $\longrightarrow$ e1', ..., ek $\longrightarrow$ ek'. This proof-theoretical view of transitions is quite interesting and it will be adopted later when considering transitions (and their derivation from other transitions) as theorems in (possibly different) formal theories.

2.1.2 Static Semantics of L

In order to provide the definition of the operational semantics of our language L, we need to give a set of axioms and a set of rules for transforming expressions and definitions. Since not all expressions (and definitions) in the language L are well-formed, we first select a subset of them to which axioms and transformation rules may be applied, by using a method called static semantics [Plotkin 81]. For instance, the expression "(1+3)+false" will not be considered to be well-formed, and although it might be transformed into "4+false", we do not do so.
The static semantics is a deductive system for performing the typechecking and deducing well-formed compound expressions from well-formed component subexpressions (and analogously for definitions). For an introductory example of use of the static semantics see Appendix A.

Let us now introduce the static semantics for L.
We first consider the following auxiliary sets:
1. a set V of variables which is a finite subset of Var, i.e. $V \subseteq_{fin} Var$.
2. a set of Expressible Types, called ETypes, ranged over by et. The elements of ETypes are the types of the expressions. We choose for our language L:

$$et ::= \tau \quad \text{where } \tau \in Typ$$

Therefore expressions may be evaluated to either an integer or a boolean value only, because we have Typ = {bool, int}.
3. a set of Denotable Types, called DTypes, ranged over by dt. The elements of DTypes are the types of the values which are bound to variables (defined in the definitions).
We choose for our language:

$$dt ::= \tau \mid \tau_0 \rightarrow \tau_1 \qquad \text{where } \tau, \tau_0, \tau_1 \in Typ.$$

Therefore variables may have one of the following types:
int, bool, int $\rightarrow$ int, int $\rightarrow$ bool, bool $\rightarrow$ int, bool $\rightarrow$ bool.

4. a set of Type-Environments, called TEnv, ranged over by $\alpha, \beta, \ldots$

$$\mathrm{TEnv} = \sum_{V \subseteq_{fin} Var} \mathrm{TEnv}_V \quad \text{where } \mathrm{TEnv}_V = V \rightarrow \mathrm{Dtypes}.$$

Therefore a type-environment $\alpha \in \mathrm{TEnv}_V$ binds variables in V to denotable types in DTypes. (We used the symbol $\sum$ to denote the disjoint sum operation.) We will write $\alpha: V$ meaning that $\alpha \in V \rightarrow \mathrm{DTypes}$ where the intended set DTypes should be understood from the context.

A particular pair $\langle x, \tau \rangle$ in α will also be written as: $x=\tau$.

5. a set of Denotable Values, called DVal, ranged over by dv.

(This set and the set of Environments (see 6.) are needed only for the dynamic semantics definition, but we introduce them here for convenience reasons.) The elements of DVal are the values to which variables are bound in the environments. We choose:

dv $\in$ DVal = N+T+Abstracts, where + denotes the disjoint sum, N is the set of natural numbers, T is {tt, ff}, and Abstracts is the following set:

$\{\lambda x{:}\, et0.\, e{:}\, et1 \mid \alpha[\{x=et0\}] \vdash_{V \cup \{x\}} e{:}\, et1$

for some $\alpha{:}\, V$ and $FV(e) \subseteq V\}$.

Abstracts is a set of constructs of the form $\lambda x{:}\, et0.\, e{:}\, et1$ where x is a variable of type et0 and e is an expression of type et1, satisfying the property written to the right of the vertical bar |. Informally, that property says that from the type information given by the type environment α and the fact that x has type et0, we can derive that e has type et1. (The formal definition of $\vdash$ will be given later on.) In the definition of Abstracts the set V is constrained by the fact that the set of "free variables" of e (denoted by FV(e) and later defined) is a subset of V.

6. a set of Environments, called Env, ranged over by ρ, $\rho1$, ...

$$\mathrm{Env} = \sum_{V \subseteq_{fin} Var} \mathrm{Env}_V \quad \text{where } \mathrm{Env}_V = V \rightarrow \mathrm{DVal}.$$

Therefore an environment $\rho \in \mathrm{Env}_V$ binds variables in V to denotable values in

DVal. We will write ρ: V meaning that $\rho \in V \longrightarrow$ DVal where the intended set DVal should be understood from the context.

A particular pair <x, c> in ρ will be also be written as: x=c.

7. the set of the free variables FV(e) for any expression e, and the set of the free variables FV(d) and that of the defined variables DV(d) for any definition d. They are defined as the table 2-1 shows.

We will say that an expression (or a definition) is closed if the set of its free variables is empty.

Example 12.

α = {x=int, y=bool, f=int$\rightarrow$int} $\in$ TEnv$_V$, where V={x, y, f}.

λz: int. z+x: int $\in$ Abstracts. Indeed α[{z=int}] $\vdash_{V \cup \{z\}}$ z+x: int (see also Example 13 below).

ρ = {x=5, y=true, f=λz: int. z+x: int} $\in$ Env$_V$.

FV(z+x) ={z, x} and FV(λz: int. z+x: int) ={x}. □

For defining the static semantics we also need some formulas which allow us to express the well-formedness property of the expressions and the definitions of our language L. Now we introduce those formulas and we give their intuitive meaning.

For expressions we have:

i) $\alpha \vdash_V e : \tau$

which means that, given the type environment α: V such that FV(e) $\subseteq$ V, e is well-defined (or well-typed) and e has type τ. (We need α for knowing the types of the free variables in e.)

For definitions we have:

ii) $\alpha \vdash_V d$

which means that, given the type environment α: V such that FV(d) $\subseteq$ V, d is well-defined (or well-typed). (We need α for knowing the types of the free variables in d.)

For expressions:

	m	t	x	bop(. . . . , e_i, . . .)	if e_0 then e_1 else e_2
FV	$\emptyset$	$\emptyset$	{x}	$\cup_i$ FV(e_i)	FV(e_0) $\cup$ FV(e_1) $\cup$ FV(e_2)

	let d in e	f(e)
FV	FV(d) $\cup$ (FV(e)\DV(d))	{f} $\cup$ FV(e)

For Abstracts: FV(λx: τ_0. e: τ_1) = FV(e)\{x}

For definitions:

	x: τ = e	f(x: τ_0) : τ_1 = e	rec d
FV	FV(e)	FV(e)\{x}	FV(d)\DV(d)
DV	{x}	{f}	DV(d)

	d0 and d1	ρ (with ρ: V)
FV	FV(d0) $\cup$ FV(d1)	$\bigcup_{x \in V}$ FV(ρ(x))
DV	DV(d0) $\cup$ DV(d1)	V

Notes.

1. τ, τ_0, τ_1 $\in$ ETypes.

2. If f(x: τ_0) : τ_1 = e and f $\in$ FV(e) then f $\in$ FV[f(x: τ_0) : τ_1 = e].

3. If rec f(x: τ_0) : τ_1 = e and f $\in$ FV(e) then f $\notin$ FV[rec f(x: τ_0) : τ_1 = e].

Table 2-1: Free and Defined Variables for Expressions and Definitions.

iii) $\vdash_V d:\beta$ (where $\beta \in$ TEnv)

which means that: $FV(d) \subseteq V$ and <u>for all</u> type-environments $\alpha: V \longrightarrow$ DTypes if d is well-defined then "d agrees with β", i.e. if in the definition d a (functional or individual) variable x is bound to a value v then $\beta(x)$ is the type of v.

iv) $\alpha \vdash_V d:\beta$

which is an abbreviation for the formula "$\alpha \vdash_V d,\ \vdash_V d:\beta$", where "," denotes the logical "and" of the two subformulas. (Often the comma is not written.)

Now we give the axioms and rules of inference for the formal definition of the formulas introduced above. These axioms and rules define the <u>static semantics</u>. Unless otherwise specified, they hold for any $V \subseteq$ Var and any $\alpha \in \text{TEnv}_V = V \longrightarrow$ DTypes.

In what follows $\alpha\vdash\ldots$ stands for $\alpha\vdash_V\ldots$ where $V=\text{dom}(\alpha)$, and $\vdash d:\beta$ stands for $\vdash_V d:\beta$ where $FV(d)\subseteq V$.

<u>For expressions</u>:

Numbers

N. $\alpha \vdash m:\text{int}$ for any $m \in N$.

Truthvalues

T. $\alpha \vdash t:\text{bool}$ for any $t \in T$.

(Individual) Variables

V. $$\frac{\alpha(x) = \tau}{\alpha \vdash x:\tau}$$ for any (individual) variable x.

Basic Operators

B. $$\frac{\alpha \vdash e_0 : \tau_0, \ \ldots, \ \alpha \vdash e_{n-1} : \tau_{n-1}}{\alpha \vdash \mathrm{bop}_n(e_0, \ldots, e_{n-1}) : \tau}$$

if $\tau_{\mathrm{bop}_n} : \tau_0 \times \ldots \times \tau_{n-1} \longrightarrow \tau$.

Conditional

C. $$\frac{\alpha \vdash e_0 : \mathrm{bool}, \quad \alpha \vdash e_1 : \tau, \quad \alpha \vdash e_2 : \tau}{\alpha \vdash \underline{\mathrm{if}}\ e_0\ \underline{\mathrm{then}}\ e_1\ \underline{\mathrm{else}}\ e_2 : \tau}$$

Let-clause

L. $$\frac{\alpha \vdash d : \beta, \qquad \alpha[\beta] \vdash e : \tau}{\alpha \vdash_V \underline{\mathrm{let}}\ d\ \underline{\mathrm{in}}\ e : \tau}$$ where β: DV(d).

Note.

As usual, if α: V and β: V_0 then $\alpha[\beta] \in V \cup V_0 \longrightarrow$ DTypes and it is defined as follows:

$$\alpha[\beta](x) = \begin{cases} \beta(x) & \text{if } x \in V_0 \\ \alpha(x) & \text{if } x \in V - V_0. \end{cases}$$

Application

A. $$\frac{\alpha \vdash e : \tau_0}{\alpha \vdash f(e) : \tau_1}$$ if $\alpha(f) = \tau_0 \rightarrow \tau_1$.

Abstracts

Ab. $$\frac{\alpha[\{x = \tau_0\}] \vdash e : \tau_1}{\alpha \vdash (\lambda x : \tau_0 . e : \tau_1) : (\tau_0 \longrightarrow \tau_1)}$$

Note.

This rule says that, if we know that e has type τ_1 knowing that x has type τ_0, then we conclude that $\lambda x{:}\,\tau_0.\, e{:}\,\tau_1$ has type $\tau_0 \rightarrow \tau_1$.

Since Abstracts only occur during execution, static semantic rules for them would not be necessary. We give them for technical reasons. The same holds for Environments.

<u>For definitions</u>:

Simple definitions

S1. $$\frac{\alpha \vdash e{:}\,\tau}{\alpha \vdash (x{:}\,\tau = e)}$$

S2. $\vdash (x{:}\,\tau = e)\ :\ \{x = \tau\}$

Function definitions

F1. $$\frac{\alpha[\{x = \tau_0\}] \vdash e{:}\,\tau_1}{\alpha \vdash f(x{:}\,\tau_0) : \tau_1 = e}$$

F2. $\vdash (f(x{:}\,\tau_0) : \tau_1 = e)\ :\ \{f = \tau_0 \rightarrow \tau_1\}$

Recursive definitions

R1. $$\frac{\vdash d{:}\,\beta \qquad \alpha[\beta \restriction V_0] \vdash d}{\alpha \vdash \underline{\text{rec}}\ d}$$ where $V_0 = FV(d) \cap DV(d)$.

R2. $$\frac{\vdash d{:}\,\beta}{\vdash (\underline{\text{rec}}\ d) : \beta}$$

Here is an example of application of rule R1:

from

$\phi[\{f : int \rightarrow int\}] \vdash f(x: int) : int =$ <u>if</u> $x=0$ <u>then</u> 1 <u>else</u> $x \cdot f(x-1)$

and

$\vdash f(x: int) : int =$ <u>if</u> $x=0$ <u>then</u> 1 <u>else</u> $x \cdot f(x-1) : \{f = int \rightarrow int\}$

we can derive:

$\phi \vdash$ <u>rec</u> $f(x: int) : int =$ <u>if</u> $x=0$ <u>then</u> 1 <u>else</u> $x \cdot f(x-1)$.

Notes.

1) $\beta\restriction V_0$ is the type-environment β restricted to V_0.

2) V_0 is the set of recursively defined (functional) variables in <u>rec</u> d, which also occur as free variables in d.

3) If $DV(d) \subseteq FV(d)$ then $V_0=DV(d)$. In that case by the Strong Agreement Theorem, given at the end of this subsection, which asserts that domain(β) = DV(d), we could replace the premises of rule R1 by the following ones: $\vdash d : \beta$ and $\alpha[\beta] \vdash d$.

Simultaneous definitions

AND1. $$\frac{\alpha \vdash d0 \qquad \alpha \vdash d1}{\alpha \vdash d0 \ \underline{\text{and}}\ d1} \quad \text{if } DV(d0) \cap DV(d1) = \phi.$$

AND2. $$\frac{\vdash d0 : \beta 0 \qquad \vdash d1 : \beta 1}{\vdash d0 \ \underline{\text{and}}\ d1 : \beta 0 \cup \beta 1} \quad \text{if } DV(d0) \cap DV(d1) = \phi.$$

Environments.

For some type-environment β: dom(ρ)

E1. $$\frac{\forall x \in dom(\rho) \qquad \alpha \vdash \rho(x) : \beta(x)}{\alpha \vdash \rho} \quad \text{if } FV(\rho) \subseteq dom(\alpha).$$

$$\text{E2.}\quad \frac{\forall\alpha \text{ s.t. } \alpha \vdash \rho \qquad \forall x \in \mathrm{dom}(\rho). \qquad \alpha \vdash \rho(x) : \beta(x)}{\vdash \rho : \beta}$$

if $FV(\rho) \subseteq \mathrm{dom}(\alpha)$.

The above rules establish the validity of the formulas $\alpha \vdash \rho$ and $\vdash \rho : \beta$ by considering one element of ρ at a time.

<u>Example 13.</u>

Let α be $\{y=\mathrm{bool},\ z=\mathrm{int}\}$.

1. $\alpha[\{x=\mathrm{int}\}] \vdash z: \mathrm{int}$	by rule V.
2. $\alpha[\{x=\mathrm{int}\}] \vdash x: \mathrm{int}$	by rule V.
3. $\alpha[\{x=\mathrm{int}\}] \vdash z+x: \mathrm{int}$	by 1, 2 and rule B.
4. $\alpha \vdash f(x: \mathrm{int}) : \mathrm{int} = z+x$	by 3. and F1. ☐

Now we give the Strong Agreement Theorem we have anticipated in this section.

Theorem 1: <u>Strong Agreement Theorem</u>. In the language L <u>if</u> $\vdash_V d : \beta$ for some $V \subseteq \mathrm{Var}$ <u>then</u> $\mathrm{dom}(\beta) = DV(d)$.

Proof: By structural induction.

For simple definitions and function definitions it is obvious, because $DV(x: \tau = e) = \{x\}$ and $DV(f(x: \tau_0) : \tau_1 = e) = \{f\}$.

For recursive definitions the thesis derives from DV(<u>rec</u> d) = DV(d) and the fact that the same β occurs in the premise and the conclusion of rule R2.

The simultaneous definitions case is analogous to the recursive definition case. ☐

2. 1. 3 Dynamic Semantics for L

Now we will define the Dynamic Semantics for our language L, by giving axioms and rules of inference in much the same way as we defined the static semantics.

In order to do so we will first introduce some definitions and useful notions.

1. Agreement of an environment ρ with a type-environment α.

Given $\rho \in Env_V$ and $\alpha \in TEnv_W$ such that $V \subseteq W$, the agreement of an environment ρ with a type-environment α, denoted by $\rho:\alpha$, is defined as follows:

$\rho:\alpha$ iff

$$\begin{aligned} \forall x \in V.\ \{ & (\rho(x) \in N \supset \alpha(x) = \text{int}) \\ \wedge\ & (\rho(x) \in T \supset \alpha(x) = \text{bool}) \\ \wedge\ & (\forall y \in V.\ \rho(x) = \lambda y{:}\tau_0.\ e{:}\tau_1 \supset \alpha(x) = \tau_0 \rightarrow \tau_1 \\ & \qquad \text{where } \tau_0,\ \tau_1 \in \text{ETypes})\ \}. \end{aligned} \qquad (1)$$

Therefore, $\rho:\alpha$ means that α is a type-environment "corresponding" to the environment ρ in the sense specified by (1) above. Notice that dom(ρ) can be a subset of dom(α).

(This notion of agreement between an environment ρ and a type-environment α is slightly different from the one given in [Plotkin 81] page 84, where ρ and α are assumed to have the same domain.)

Example 14.

Given $\rho = \{x=5,\ y=6\}$, $\alpha1 = \{x=\text{int}\}$, $\alpha2 = \{x=\text{int},\ y=\text{int}\}$, $\alpha3 = \{x=\text{int},\ y=\text{int},\ z=\text{int}\}$, and $\alpha4 = \{x=\text{int},\ y=\text{bool}\}$, we have: $\rho:\alpha2$, $\rho:\alpha3$. However, $\rho:\alpha1$ or $\rho:\alpha4$ does not hold.

2. <u>Well-formedness of an expression and a definition w.r.t. a type-environment</u> α.

(I) Given $e \in$ Exp and $\alpha \in \mathrm{TEnv}_V$, s.t. $FV(e) \subseteq V$, the well-formedness of the expression e w.r.t. the type-environment α, denoted by $W_V(e,\alpha)$, is defined as follows:

$W_V(e,\alpha)$ iff $\exists\tau \in$ ETypes. $\alpha \vdash e:\tau$

(II) Given $d \in$ Def and $\alpha \in \mathrm{TEnv}_V$, s.t. $FV(d) \subseteq V$, the well-formedness of the definition d w.r.t. the type-environment α, denoted by $W_V(d,\alpha)$, is defined as follows:

$W_V(d,\alpha)$ iff $\exists\beta \in \mathrm{TEnv}_{DV(d)}$. $\alpha \vdash d$ <u>and</u> $\vdash d:\beta$.

<u>Example 15.</u>

Let α be $\{x=\mathrm{int}, y=\mathrm{int}\}$, V be (x,y), e be $x+5$ and d be <u>rec</u> f(x: int) : int = <u>if</u> x=0 <u>then</u> 1 <u>else</u> x+f(x−1). We have that $W_V(e,\alpha)$ holds with τ=int, and $W_V(d,\alpha)$ holds with $\beta=\{f: \mathrm{int}\rightarrow\mathrm{int}\}$.

3. <u>Expression Configurations and Definition Configurations w.r.t. a type-enviroment</u> α.

Given $\alpha \in \mathrm{TEnv}_V$, the set $E\Gamma_\alpha$ of <u>Expression Configurations</u> is defined as follows:

$$E\Gamma_\alpha = \{ \langle\rho, e\rangle \mid e \in \mathrm{Exp}\ \underline{\text{and}}\ W_V(e,\alpha)\ \underline{\text{and}}\ \rho:\alpha\ \underline{\text{and}}\ FV(e) \subseteq V \}.$$

Analogously we define the <u>Definition Configurations</u> as follows:

$$D\Gamma_\alpha = \{ \langle\rho, d\rangle \mid d \in \mathrm{Def}\ \underline{\text{and}}\ W_V(d,\alpha)\ \underline{\text{and}}\ \rho:\alpha\ \underline{\text{and}}\ FV(d) \in V \}.$$

The set of <u>Configurations</u> Γ_α is $E\Gamma_\alpha \cup D\Gamma_\alpha$.

These notions are introduced because we are interested in giving the dynamic semantics of expressions (and definitions) which are not closed, i.e. whose set of free variables is not empty. We need, in fact, to express the dynamic semantics of let-expressions by giving the semantics of its subexpressions, which do not need to be closed.

Notice also that the set of configurations is parameterized by type-environments and therefore, the transitions from configurations to configurations (later defined) belong to different formal theories, each of which is indexed by a different type-environment. (This point, however, has only a theoretical relevance.)

Example 16.

Given $\alpha1$ = {x=int}, $\alpha2$ = {x=bool}, $\alpha3$ = {y=int}, and γ = <{x=5}, x+3>, we have: $\gamma \in E\Gamma_{\alpha1}$, $\gamma \notin E\Gamma_{\alpha2}$ because $\alpha2$(x)=bool and $\gamma \notin E\Gamma_{\alpha3}$ because x$\in$FV(x+3) but x$\notin$dom($\alpha3$).

4. Terminal Configurations.

Among the configurations we distinguish the following:

i) Terminal Expression Configurations:

TEΓ is the largest subset of $\bigcup_{\alpha \in TEnv} E\Gamma_{\alpha}$, such that $\forall\gamma \in TE\Gamma$ the second component (i.e. the expression) of γ is an element of DVal = N + T + Abstracts.

ii) Terminal Definition Configurations:

TDΓ is the largest subset of $\bigcup_{\alpha \in TEnv} D\Gamma_{\alpha}$, such that $\forall\gamma \in TD\Gamma$ the second component (i.e. the definition) of γ is an element of Env.

The set of the terminal configurations TΓ is TEΓ $\cup$ TDΓ.

Example 17.

<{x=5}, 8> $\in$ TEΓ, <{x=5}, x+3> $\notin$ TEΓ, <{x=5}, {f=λz: int. x+z: int}> $\in$ TDΓ.

5. Transition Systems w. r. t. a type-environment α.

A transition system w. r. t. a type-environment α is a triple $\langle \Gamma_\alpha, T_\alpha, \rightarrow_\alpha \rangle$ where Γ_α is the set of Configurations, $T_\alpha = T\Gamma \cap \Gamma_\alpha$ is the set of Terminal Configurations, and $\rightarrow_\alpha$ is a subset of $\Gamma_\alpha \times \Gamma_\alpha$ such that $\forall \gamma \in T_\alpha.$ $\forall \gamma' \in \Gamma_\alpha.$ $not(\gamma \rightarrow_\alpha \gamma')$.

Since by the definition we will give, $\rightarrow_\alpha$ may only relate pairs of expression configurations or definition configurations, the binary relation $\rightarrow_\alpha$ can be partitioned into two binary relations $\rightarrow_\alpha = \rightarrow_{\alpha,e} \cup \rightarrow_{\alpha,d}$ such that $\rightarrow_{\alpha,e} \subseteq E\Gamma_\alpha \times E\Gamma_\alpha$ and $\rightarrow_{\alpha,d} \subseteq D\Gamma_\alpha \times D\Gamma_\alpha$.

An element of $\rightarrow_{\alpha,e}$ will be written as $\vdash_\alpha \langle \rho 0, e0 \rangle \rightarrow \langle \rho 1, e1 \rangle$ and an element of $\rightarrow_{\alpha,d}$ as $\vdash_\alpha \langle \rho 0, d0 \rangle \rightarrow \langle \rho 1, d1 \rangle$.

If the environment component is unchanged (i. e. $\rho 0 = \rho 1$) we write also $\rho \vdash_\alpha e0 \rightarrow e1$ (or $\rho \vdash_\alpha d0 \rightarrow d1$).

In what follows we will write $\rho \vdash \ldots$ instead of $\rho \vdash_\alpha \ldots$

Now we give the rules and the axioms for the Dynamic Semantics of L.
Let us assume that $\rho \in Env_V$ and $\alpha \in TEnv_W$ s. t. $\rho : \alpha$ (and therefore $V \subseteq W$).

For expressions:

Variables

V. $\rho \vdash x \rightarrow \rho(x)$ if $x \in V$

Basic Operators

B1. $$\frac{\rho \vdash e_i \rightarrow e_i'}{\rho \vdash bop(\ldots, e_i, \ldots) \rightarrow bop(\ldots, e_i', \ldots)} \quad \text{for } i = 0, 1, \ldots$$

B2. $\rho \vdash bop(c_0, c_1, \ldots, c_{n-1}) \rightarrow c_n$
if $c_0, c_1, \ldots, c_{n-1} \in N + T$ and $bop(c_0, c_1, \ldots, c_{n-1}) = c_n$.

Notes.

1) Rule B1 allows the arguments to be evaluated in a non-deterministic way. For instance +(+(1,2),+(3,4)) can be rewritten into +(3,+(3,4)) or +(+(1,2),7). They are both eventually rewritten into 10.

2) In rule B2 bop stands for the mathematical function corresponding to bop.

Conditional

C1. $$\frac{\rho \vdash e \longrightarrow e'}{\rho \vdash \underline{\text{if}}\ e\ \underline{\text{then}}\ e_0\ \underline{\text{else}}\ e_1 \longrightarrow \underline{\text{if}}\ e'\ \underline{\text{then}}\ e_0\ \underline{\text{else}}\ e_1}$$

C2. $\rho \vdash \underline{\text{if}}\ tt\ \underline{\text{then}}\ e_0\ \underline{\text{else}}\ e_1 \longrightarrow e_0$

C3. $\rho \vdash \underline{\text{if}}\ ff\ \underline{\text{then}}\ e_0\ \underline{\text{else}}\ e_1 \longrightarrow e_1$

Note. The above rules force the evaluation of the predicate of a conditional before evaluating any of its arms.

Let-clause

L1. $$\frac{\rho \vdash d \longrightarrow d'}{\rho \vdash \underline{\text{let}}\ d\ \underline{\text{in}}\ e \longrightarrow \underline{\text{let}}\ d'\ \underline{\text{in}}\ e}$$

L2. $$\frac{\rho[\rho_0] \vdash e \longrightarrow e'}{\rho \vdash \underline{\text{let}}\ \rho_0\ \underline{\text{in}}\ e \longrightarrow \underline{\text{let}}\ \rho_0\ \underline{\text{in}}\ e'}$$

L3. $\rho \vdash \underline{\text{let}}\ \rho_0\ \underline{\text{in}}\ c \longrightarrow c$ where $c \in N + T$.

Note. The above rules force a sequential evaluation of the expression $\underline{\text{let}}\ d\ \underline{\text{in}}\ e$. We first evaluate the definition d and then we evaluate the subexpression e in a suitable new environment.

For example, we have:

$\phi \vdash \underline{\text{let}}\ \{x=2\}\ \underline{\text{in}}\ x+3 \longrightarrow \underline{\text{let}}\ \{x=2\}\ \underline{\text{in}}\ 2+3$ because $\{x=2\} \vdash x \longrightarrow 2$.

Application

A1. $$\frac{\rho \vdash e \longrightarrow e'}{\rho \vdash f(e) \longrightarrow f(e')}$$

A2. $\rho \vdash f(c) \longrightarrow \underline{\text{let}}\ x{:}\,\tau_0 = c\ \underline{\text{in}}\ e$

where $c \in N + T$ and $\rho(f) = \lambda x{:}\,\tau_0.\, e{:}\,\tau_1$.

Notes.

1) These rules correspond to a call-by-value mode of evaluation.

2) In order to avoid an extra rewriting we could have rule A2 as follows:
$\rho \vdash f(c) \longrightarrow \underline{\text{let}}\ \{x=c\}\ \underline{\text{in}}\ e$ (see below the dynamic semantics for Simple Definitions).

Notice that we have not given rewriting rules for elements in N+T. They are, in fact, second components of Terminal Expression Configurations.

For definitions:

Simple Definitions

S1. $$\frac{\rho \vdash e \longrightarrow e'}{\rho \vdash x{:}\,\tau = e \longrightarrow x{:}\,\tau = e'}$$

S2. $\rho \vdash x{:}\,\tau = con \longrightarrow \{x = con\}$ if $con \in N + T$.

Note. We could keep in the environments the type information corresponding to individual variables. We could have $\{x = \langle con{:}\,\tau\rangle\}$, instead of having $\{x = con\}$ [Burstall and Lampson 84]. For simplicity reasons we do not do so.

Function Definitions

F1. $\rho \vdash f(x{:}\,\tau_0) : \tau_1 = e \longrightarrow \{f = \lambda x{:}\,\tau_0.\ (\underline{\text{let}}\ \rho \restriction V_0\ \underline{\text{in}}\ e) : \tau_1\}$

where $V_0 = FV(e) \setminus \{x\}$

Recursive Definitions

R1. $$\frac{\rho \setminus V_0 \vdash d \longrightarrow d'}{\rho \vdash \underline{\text{rec}}\ d \longrightarrow \underline{\text{rec}}\ d'} \qquad \text{where } V_0 = DV(d) \cap FV(d).$$

R2. $$\rho \vdash \underline{\text{rec}}\ \rho_0 \longrightarrow \{\ x = con \mid (x = con) \in \rho_0\ \underline{\text{and}}\ con \in N + T\ \}$$
$$\cup$$
$$\{\ f = \lambda x{:}\,\tau_0.\ (\underline{\text{let}}\ [(\underline{\text{rec}}\ \rho_0) \setminus \{x\}]\ \underline{\text{in}}\ e) : \tau_1 \mid (f = \lambda x{:}\,\tau_0.\ e{:}\,\tau_1) \in \rho_0\ \} \qquad (1)$$

Notes.

(1) $\rho \setminus V_0$ is the function defined as follows:

$(\rho \setminus V_0)(x) = \rho(x)$ if $x \notin V_0$. otherwise undefined.

(2) Let dom(ρ) be V. We have: $FV(d) \subseteq V$ because $\langle\rho, d\rangle \in D\Gamma_\alpha$. Since $V_0 \subseteq FV(d)$ we get: $V_0 \subseteq V$.

(3) The effect of the rules R1 and R2 is to first evaluate a recursive definition with no knowledge about the recursively defined variables ($\rho \setminus V_0$ is used) and, when no further evaluation is possible (i.e. a definition of the form "$\underline{\text{rec}}\ \rho_0$" is obtained), a new environment $\overline{\rho}_0$ is created. $\overline{\rho}_0$ is like ρ_0 except that, for any $x \in \text{domain}(\rho_0)$ the free variables occurring in $\rho_0(x)$ (and they can only occur in Abstracts) are bound as in $\underline{\text{rec}}\ \rho_0$.

We will give some examples of this rule in what follows.

Simultaneous Definitions

AND1. $$\frac{\rho \vdash d0 \longrightarrow d0'}{\rho \vdash d0 \ \underline{\text{and}}\ d1 \longrightarrow d0' \ \underline{\text{and}}\ d1}$$

AND2. $$\frac{\rho \vdash d1 \longrightarrow d1'}{\rho \vdash d0 \ \underline{\text{and}}\ d1 \longrightarrow d0 \ \underline{\text{and}}\ d1'}$$

AND3. $\rho \vdash \rho 0 \ \underline{\text{and}}\ \rho 1 \longrightarrow \rho 0[\rho 1]$

Note. The static semantics tells us that domain($\rho 0$) $\cap$ domain($\rho 1$) = ϕ. Therefore: $\rho 0[\rho 1] = \rho 1[\rho 0] = \rho 0 \cup \rho 1$.

Notice that no dynamic semantics rules are given for definitions which are environments. They are, in fact, the second components of <u>Terminal</u> Definition Configurations, and they cannot be rewritten.

<u>2.1.4 Some Examples of the Operational Semantics of the Language L</u>

Now we would like to show some examples of use of the operational semantics definitions we have given.

<u>Example 18.</u>

Suppose we are given the expression (4+(2+1))+(2+1) and we would like to evaluate it in the environment $\rho = \phi$. Since the environment does not play any interesting role, we will write configurations as expressions only. We leave to the reader the obvious static semantics analysis. The dynamic semantics gives the diagram of rewritings of figure 2-5. Therefore the application of the

$$\begin{tikzcd}
(4+(2+1))+(2+1) \arrow[r] \arrow[d] & (4+3)+(2+1) \arrow[r] \arrow[d] & 7+(2+1) \arrow[r] & 7+3 \arrow[r] & 10 \\
(4+(2+1))+3 \arrow[r] & (4+3)+3 \arrow[urr] & & &
\end{tikzcd}$$

Figure 2-5: The possible rewritings of a simple expression.

dynamic semantics rules is non-deterministic and one may be interested in looking for strategies for choosing the redexes to be reduced in the "best possible way" (e.g. for obtaining the shortest sequence of rewritings). Obviously this problem is, in general, undecidable, and therefore one can only hope to give "restricted" solutions.
The problem is somehow more complicated if one would like to avoid repeated evaluations of common subexpressions. We will deal with that issue later on by providing a structural operational semantics definition, for which redundant evaluations of recursive functions applications will not occur.

In our structural style of giving the operational semantics definitions we keep separated the two concerns of correctness and efficiency of reductions (this motivation was already in the Burstall-Darlington transformation approach for recursive equation programming). That separation of concerns may be viewed in logical terms as the separation between Model Theory (for dealing with correctness) and Proof Theory (for dealing with efficiency). One can therefore apply what is already known in those theories and apply the relevant techniques when studying correctness and efficiency of operational semantics.

Let us now look in detail at a particular sequence of rewritings for the given expression (4+(2+1))+(2+1). If we write the "full proof" of such a sequence we have the diagram of figure 2-6. That sequence requires 8 applications of the rules B1 or B2, and we consider the rewritings from

$$\begin{array}{ccccc}
(4+(2+1))+(2+1) \longrightarrow & (4+3)+(2+1) \longrightarrow & 7+(2+1) \longrightarrow & 7+3 \longrightarrow & 10 \\
\uparrow \underline{B1} & \uparrow \underline{B1} & \uparrow B1 & \uparrow B2 & \\
4+(2+1) \rightarrow 4+3 & 4+3 \rightarrow 7 & 2+1 \rightarrow 3 & & \\
\uparrow B1 & \uparrow B2 & \uparrow B2 & & \\
2+1 \rightarrow 3 & & & & \\
\uparrow B2 & & & &
\end{array}$$

Figure 2-6: Proof of the rewritings of a simple expression (using rule B1).

"4+(2+1)+(2+1)" to "10" as done in 8 steps, although the "surface transitions" are only 4.

Is there any way of reducing the number of steps for making the rewriting process "faster"?

The two underlined uses of rule B1 (see figure 2-6) could have been reduced to one only, by using the following rule:

$$B1^{+}. \quad \frac{\rho \vdash e_i \longrightarrow^{+} con}{\rho \vdash bop(\ldots, e_i, \ldots) \longrightarrow^{+} bop(\ldots, con, \ldots)}$$

for $i=0, 1, \ldots$ and $con \in N + T$.

where $\longrightarrow^{+}$ denotes the transitive closure of $\longrightarrow$.

The resulting sequence of reductions is given in figure 2-7. This example illustrates that the <u>structural approach</u> we presented (when we used rule B1 instead of $B1^{+}$) is, in a sense, <u>too structural</u> because any transformation of a subexpression has to be represented as a transformation of the given expression (thereby using extra deduction steps).

We will examine again the issue of using $\longrightarrow^{+}$ (or $\longrightarrow^{*}$) instead of $\longrightarrow$ in Appendix C.

$$\begin{tikzcd}[column sep=small]
(4+(2+1))+(2+1) \arrow[rrrr, "+"] & & & & 7+(2+1) \arrow[r, "+"] & 7+3 \arrow[r] & 10 \\
& & \uparrow B1^{+} & & \uparrow B1^{+} & \uparrow B2 & \\
4+(2+1) \arrow[r, "+"] & 4+3 \arrow[r] & 7 & & 2+1 \arrow[r] & 3 & \\
\uparrow B1^{+} & \uparrow B2 & & & \uparrow B2 & & \\
2+1 \arrow[r] & 3 & & & & & \\
\uparrow B2 & & & & & &
\end{tikzcd}$$

Figure 2-7: Proof of the rewritings of a simple expression using rule $B1^{+}$ (instead of rule B1).

The following example will illustrate the "theorem proving" view of the operational semantics definitions.

Example 19.

Suppose we want to evaluate the expression "let y: int=7 in x+y" in the environment ρ = {x=5, z=3}. We have: ρ: V where V={x, z}. Let the type environment α be {x=int, z=int}.

Static Semantics.

Suppose β : {y} $\longrightarrow$ DTypes s.t. β(y)=int. We use a linear Hilbert-like presentation of the proof, as follows:

1.	$\alpha[\beta] \vdash$ x: int	by V
2.	$\alpha[\beta] \vdash$ y: int	by V
3.	$\alpha[\beta] \vdash$ x + y: int	by 1, 2, B.
4.	$\alpha \vdash$ 7: int	by N
5.	$\alpha \vdash$ (y: int = 7)	by S1, 4.
6.	$\vdash$ (y: int = 7) : {y = int}	by S2
7.	$\alpha \vdash$ let y: int = 7 in x + y: int	by 3, 5, 6, L.

Dynamic Semantics

1.	$\rho \vdash$ y: int = 7 $\longrightarrow$ {y = 7}	by S2

2. $\rho \vdash \underline{\text{let}}\ y{:}\ \text{int} = 7\ \underline{\text{in}}\ x + y \longrightarrow \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ x + y$ by L1, 1
3. $\rho[\{y = 7\}] \vdash y \longrightarrow 7$ by V
4. $\rho[\{y = 7\}] \vdash x + y \longrightarrow x + 7$ by 3, B1
5. $\rho \vdash \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ x + y \longrightarrow \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ x + 7$ by 4, L2
6. $\rho[\{y = 7\}] \vdash x \longrightarrow 5$ by V
7. $\rho[\{y = 7\}] \vdash x + 7 \longrightarrow 5 + 7$ by 6, B1
8. $\rho \vdash \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ x + 7 \longrightarrow \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ 5 + 7$ by 7, L2
9. $\rho[\{y = 7\}] \vdash 5 + 7 \longrightarrow 12$ by B2
10. $\rho \vdash \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ 5 + 7 \longrightarrow \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ 12$ by 9, L2
11. $\rho \vdash \underline{\text{let}}\ \{y = 7\}\ \underline{\text{in}}\ 12 \longrightarrow 12$ by L3.

$\square$

Figure 2-8 presents the above proof in a tree-like fashion. In that figure the numbers stand for the corresponding transitions of the proof and $i \Rightarrow j$ denotes that the final configuration of the transition i is the same as the initial configuration of the transition j.

$$\begin{tikzcd}
2 \arrow[r, Rightarrow] & 5 \arrow[r, Rightarrow] & 8 \arrow[r, Rightarrow] & 10 \arrow[r, Rightarrow] & 11 \\
1 \arrow[u, "L1"'] & 4 \arrow[u, "L2"'] & 7 \arrow[u, "L2"'] & 9 \arrow[u, "L2"'] & {} \arrow[u, "L3"'] \\
{} \arrow[u, "S2"'] & 3 \arrow[u, "B1"'] & 6 \arrow[u, "B1"'] & {} \arrow[u, "B2"'] & \\
 & {} \arrow[u, "V"'] & {} \arrow[u, "V"'] & &
\end{tikzcd}$$

Figure 2-8: Schema of transitions for $\underline{\text{let}}\ y{:}\ \text{int} = 7\ \underline{\text{in}}\ x + y$, where x is bound to 5. The numbers denote the transitions given in the text.

The process of deriving the transitions justifying a "surface transition", is a theorem proving process.

In order to illustrate this point, let us consider, for instance, the initial configuration of the transition 8:

$< \{x = 5,\ z = 3\},$ <u>let</u> $(y = 7)$ <u>in</u> $x + 7 >$

It matches the conclusion of rule L2 for $\rho = \{x=5, z=3\}$ and $\rho_0 = \{y=7\}$. We may try to show the premise of that rule (which becomes a subgoal to be shown) by finding the expression e1 such that:

$\rho[\rho_0] \vdash x + 7 \longrightarrow e1$ (2)

In order the find e1, since (2) matches the conclusion of the rule B1, we have to find an expression e2 such that:

$\rho[\rho_0] \vdash x \longrightarrow e2.$

However, we could have also looked for an expression e3 such that: $\rho[\rho_0] \vdash 7 \longrightarrow e3$. Unfortunately e3 does not exist and a backtracking step would have been necessary.

This process of replacing goals by subgoals and remembering the rules we have to apply for obtaining the proof of a goal from the proofs of the subgoals, is exactly what is done in theorem proving.
Therefore, what is known in the theory of the automated deduction can be used in the context of our operational semantics definitions. Later on (in Section 2.5 and Appendix C) we will show how to define theories for formalizing the deduction process inherent in those definitions.

The operational semantics given in this section has been implemented on the DEC-10 Prolog system [Byrd et al. 80] at the Department of Computer Science in Edinburgh. Details on the implementation are found in Section 2.6.

2.2 Some Remarks on the Structural Operational Semantics of the Language L

We would like to study some problems concerning the semantics definition of the language L we have given in the previous section and we will propose some possible solutions.
This study will also provide the motivations for the definition (given in the following section) of the memofunction language, called memoL, in which redundant evaluations of recursive calls are avoided.

In particular, we will define the syntax and the semantics of memoL so that only one binding for each recursively defined function occurs during execution. The uniqueness of those bindings allows us to identify the "place" where we will store the memoinformation.
Unfortunately for the language L, according to our dynamic semantic rules, the uniqueness of the bindings does not hold (see below). In order to regain the validity of that property we will restrict in a suitable way the set of expressions of memoL w.r.t. those of L.

2.2.1 Multiple occurrences of bindings for the language L

Let us first analyse the mechanism of variable binding when evaluating expressions with recursive definitions. We will see that the binding for recursive defined functions is, in most cases, recorded "too often".
Suppose we want to evaluate in the empty environment the following expression (where we do not write the type information for sake of simplicity):

$\underline{\text{let}}\ \underline{\text{rec}}\ f(x) = \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x \cdot f(x-1)\ \underline{\text{in}}\ f(3).$ (exp1)

By rule F1 we have: for any $\rho \in Env$

$\rho \vdash f(x) = \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x \cdot f(x-1) \longrightarrow \{f=\lambda x.(\underline{\text{let}}\ \rho\lceil\{f\}\ \underline{\text{in}}\ IF)\}$

where IF stands for $\underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x \cdot f(x-1)$.

By rule R1 we get:

$\phi \vdash \underline{\text{rec}}\ (f(x)=IF) \longrightarrow \underline{\text{rec}}\ \{f=\lambda x.(\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\},$

because $(\rho\backslash\{f\})\lceil\{f\}=\phi$, and from exp1 we obtain:

$\underline{\text{let}}$ $\underline{\text{rec}}$ $\{f=\lambda x.\,(\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\}$ $\underline{\text{in}}$ $f(3)$. (exp2)

By R2 we have:

$\phi \vdash \underline{\text{rec}}\ \{f=\lambda x.\,(\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\} \longrightarrow \{f=\lambda x.\,\underline{\text{let}}\ \underline{\text{rec}}\ (f=\lambda x.\,\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\ \underline{\text{in}}\ \underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF\}$.

Thus from exp2 we obtain:

$\underline{\text{let}}\ \{f=\lambda x.\,\underline{\text{let}}\ \underline{\text{rec}}\ (f=\lambda x.\,\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\ \underline{\text{in}}\ \underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF\}\ \underline{\text{in}}\ f(3)$ (exp3)

We will write (F-REC) instead of $\{f=\lambda x.\,\underline{\text{let}}\ \underline{\text{rec}}\ (f=\lambda x.\,\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\ \underline{\text{in}}\ \underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF\}$.

The next deduction step produces from exp3 the expression:

$\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ (x=3)\ \underline{\text{in}}\ \underline{\text{let}}\ \underline{\text{rec}}\ (f=\lambda x.\,\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF)\ \underline{\text{in}}\ \underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF$ (exp4)

Then we get:

$\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ (x=3)\ \underline{\text{in}}$ $\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ IF$ (exp5)

After a few steps we derive:

$\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ (x=3)\ \underline{\text{in}}$ $\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ 3\cdot$
[$\underline{\text{let}}\ (x=2)\ \underline{\text{in}}$ $\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ 2\cdot$
[$\underline{\text{let}}\ (x=1)\ \underline{\text{in}}$ $\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ 1\cdot$
[$\underline{\text{let}}\ (x=0)\ \underline{\text{in}}$ $\underline{\text{let}}$ (F-REC) $\underline{\text{in}}$ $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x\cdot f(x-1)$]]].

There are two causes for inefficiency here.

One is due to the presence of expressions like $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ e$ which can be easily avoided by adding the rule:

L4. $\rho \vdash \underline{\text{let}}\ \phi\ \underline{\text{in}}\ e \longrightarrow e$

We can avoid the generation of $\underline{\text{let}}\ \phi\ \underline{\text{in}}\ e$ expressions by adding to rule F1 the condition "if $\rho\restriction V_0 \neq \phi$" and introducing the following:

F1'. $\rho \vdash f(x{:}\ \tau_0) : \tau_1 = e \longrightarrow (f=\lambda x{:}\ \tau_0.\, e{:}\ \tau_1)$ $\underline{\text{if}}$ $\rho\restriction(FV(e)\setminus\{x\}) = \phi$.

The second cause of inefficiency is due to the <u>repeated occurrences of the bindings for f</u>. It seems that there is not a simple remedy for that problem. However, assuming that $FV(e) \subseteq \{x, f\}$, one can see that axiom R2 is equivalent to the following:

R2'. $\rho \vdash$ rec $\rho_0 \longrightarrow \rho_0$.

provided that only one binding occurs for any functional variable in the expression of the "surface transition". (This condition will be somehow relaxed in Section 2.4).

Let us now give an example of application of the rules F1' and R2'. Suppose we want to evaluate in the empty environment the following expression:

let rec f(x) = if x=0 then 1 else x·f(x−1) in f(3).

We have:

$\phi \vdash$ let rec f(x) = if x=0 then 1 else x·f(x−1) in f(3) $\longrightarrow$

$\longrightarrow$ let rec {f=λx. IF} in f(3) by F1'

(where "IF" stands for "if x=0 then 1 else x·f(x−1)")

$\longrightarrow$ let {f=λx. IF} in f(3) by R2'.

We then get: let {f=λx. IF} in let {x=3} in if x=0 then 1 else x·f(x−1)

$\longrightarrow^*$ let {f=λx. IF} in let {x=3} in 3·f(2)

$\longrightarrow^*$ let {f=λx. IF} in let {x=3} in 3·let {x=2} in 2·let {x=1} in 1·
let {x=0} in if x=0 then 1 else x·f(x−1)

$\longrightarrow^*$ 6

Now the binding for f occurs only once.
Therefore rule R2' is the one we would like to have for our memofunction language memoL, because using that rule only one environment is created during the evaluation of recursive functions. As we already mentioned, that uniqueness of environment will allow a simple semantics definition of memoL.

Unfortunately rule R2' makes the presence of the keyword rec immaterial, while in the language L the presence of rec is, in general, relevant.
For instance, in L the evaluation of let f(x)=0 in let rec f(x)=f(x) in f(2) does not terminate, while the evaluation of let f(x)=0 in let f(x)=f(x) in f(2) yields 0.

As shown in Section 2.4, it turns out that if we would like to keep the "agreement" of the semantics of the language memoL, which uses rule R2', with the one of the language L, which uses rule R2, we must assume that

$f(x)=e$ is a recursive definition of f, if f occurs free in e.

The hypothesis that $FV(e) \subseteq \{x, f\}$ for R2' may seem quite restrictive, but we can satisfy it by applying the following program transformations: [1], [2] and [3].

[1] If $f(x) = e$ and the variable y of "ground type" (i.e. bool or int) belongs to FV(e), then we can transform $f(x) = e$ into $\bar{f}(x, y) = e$ and we can replace the calls of f in the given program by suitable calls of $\bar{f}$.

This transformation technique is quite common in applicative programming, and it is used for passing global variables among function calls.

(In Appendix B we give the extension of the structural operational semantics rules for allowing functions with more than one argument.)

Example 20.

We can transform:

```
let rec g(y) = let rec(f(x) = if x=0 then 0 else y+f(x-1))
                    in if y=0 then 0 else f(y)+g(y-1)
in g(m).
```

into:

```
let rec g(y) = let rec (f(x,y) = if x=0 then 0 else y+f(x-1,y))
                    in if y=0 then 0 else f(y,y)+g(y-1)
in g(m).
```

Both expressions evaluate to $\sum_{i=0}^{m} i^2$. □

[2] If we have an expression like the following one:
<u>let</u> y=e1 <u>in</u> <u>let</u> f(x) = ...x....f...y...<u>in</u> e where
FV(e1)=$\emptyset$, and FV(...x...f...y...) = {x,y,f}, we can transform it into:
<u>let</u> f(x)=...x...f...e1...<u>in</u> e. □

[3] If we have a recursive definition of the form:
<u>rec</u> [f(x) = ...f...x...g... <u>and</u> g(x) = ...g...x...f...], we can construct a function which is the pair <f,g> as follows:
<u>rec</u> <f,g>(x) = <...π1<f,g>...x...π2<f,g>....
...π2<f,g>...x...π1<f,g>...>.
This transformation has been derived from [Landin 64]. □

Notice that the transformations we presented above can be easily generalized, but we will not do so here.

<u>2.2.2 Further Discussions</u>

Let us now suggest another possible variant of the dynamic semantics rules for recursive definitions with the aim of reducing their complexity. The rule under examination is again rule R2.
Let us first consider the following example.
Let ρ_0 be {f=λx:int.(<u>if</u> x=0 <u>then</u> 1 <u>else</u> x·g(x−1)):int;
g=λy:int.(<u>if</u> y=0 <u>then</u> 1 <u>else</u> y·f(y−1)):int;
h=λz:int.(<u>if</u> z=0 <u>then</u> 1 <u>else</u> 2·h(z−1)):int}.
We have: f(x)=g(x)=x! and h(z)=2^z.
When rewriting <u>rec</u> ρ_0 using R2, we need to write a definition of the form (<u>rec</u> ρ_0)\{x} for the binding of f. But a simple syntactic analysis of the environment ρ_0 shows that the use of (<u>rec</u> ρ_0)\{x,h}, instead of (<u>rec</u> ρ_0)\{x}, in the binding for f is also correct, because never f (or the functions invoked by f) need to refer to the binding for h.
In general we can replace rule R2, by the following R2":

R2". $\rho \vdash \underline{\text{rec}}\ \rho_0 \longrightarrow \{x{=}\text{con} \mid (x{=}\text{con}) \in \rho_0\ \underline{\text{and}}\ \text{con} \in N{+}T\} \quad \cup$

$\{f{=}\lambda x{:}\ \tau_0.\ (\underline{\text{let}}[(\underline{\text{rec}}\ \rho_0)\backslash\{x\}]\restriction V_0\ \underline{\text{in}}\ e) : \tau_1$

$\mid (f{=}\lambda x{:}\ \tau_0.\ e{:}\ \tau_1) \in \rho_0\}$

where V_0 is the smallest set of variables s.t.

1. $(FV(e)\backslash\{x\}) \subseteq V_0$

2. $\underline{\text{if}}\ g \in V_0\ \underline{\text{and}}\ (g{=}\lambda y{:}\ \tau_0.\ \text{body}{:}\ \tau_1) \in \rho_0\ \underline{\text{then}}\ (FV(\text{body})\backslash\{y\}) \subseteq V_0$.

In intuitive terms, V_0 is the set of all variables whose value is necessary when evaluating f.

We leave to the reader the formal proof of the equivalence between R2 and R2". □

The operational semantics we have presented, is according to the static binding mechanism and it works correctly only for closed expressions. (See also Section 2.4. for a deeper discussion on these issues.)

For example, let y=3 in let f(x)=x+y in let y=2 in f(2) evaluates to 5 (not to 4), because the static binding for y in x+y is 3.

For open expressions the rules we have given reduce, for instance, let f(x)=x+y in let y=2 in f(2) to 4, while the correct evaluation (using the static binding) gives "2+y" (or "error", as one might desire, for denoting that the variable y in x+y is not bound).

A solution to those problems can be achieved via closures or via "stratification" of the environments: they have to be structured as "stacks" of sets of <variable,value> pairs, not as "sets" of pairs as in the approach we presented. That change of the environment structure is necessary for representing the nesting of the recursive calls and keeping track of the links between binding occurrences of the variables and the corresponding bound occurrences.

2.3 Use of memofunctions for avoiding repeated computations in recursive programs

In this section we will present the operational semantics of an applicative language where memofunctions are used in order to avoid repeated evaluations of recursive calls. The use of memofunctions is the first step towards methods of allowing "communications among function calls" for a fast evaluation of applicative programs. We will say something about those methods at the end of the thesis.

The memo information for a given function is stored in a 'place' (called 'rote' in [Michie 68]) associated with the function definition, and it consists of the argument-value pairs already computed. The memo information can be looked up and updated by the various function calls, so that they may know at run time the results of previous computations and increase their efficiency. The memo information is available within the scope of the function definition only. It is automatically discarded when the evaluation is outside the relevant scope.
The reader may contrast this approach with the one implemented in [Burstall, Collins and Popplestone 71] pages 209–214, where memo information is not dynamically updated during recursive function evaluations.

We give the operational semantics for memofunctions using the structural method à la [Plotkin 81].

We stick to the idea of introducing as <u>fewer notions as possible</u>, which is one of the basic principles of Plotkin's method. In particular we want only to modify the notion of Environments by binding the function symbols to the corresponding lambda expressions and the sets of the argument-values pairs already computed.

Our semantics definitions for memofunctions can also be viewed as an experiment in testing how far one can stick to the principle of not introducing extra notions, without making the semantics rules too restrictive or too hard to understand.
The result of that experiment is quite satisfactory: indeed the semantic rules for our language with memofunctions turn out to be not very complicated, at the expense of a slight restriction on the set of Expressions (see below).

We can also give the operational semantics of memofunctions using Landin's style and introducing the transition rules for an SECDM machine (where M stands for the Memo component), but we will not do so here.

Let us give first a simple example of use of memofunctions. Suppose we want to evaluate the following expression:

```
let mfib(n) = if n≤1 then 1 else   mfib(n-1) valueof mfib at n-1
                                 + mfib(n-2) valueof mfib at n-2
in  mfib(4).
```

We use the expression "mfib(n) valueof mfib at n" to denote that the value of mfib(n) has to be stored in the memo of the function mfib.
The argument-value pairs already computed can then be looked up in order to avoid redundant computations.
The following fig. 2-9 gives a pictorial view of a possible computation sequence. Using the memo information we saved many computation steps.

As an introductory example of the syntactic notations we will define and use in this section, let us present in detail the steps from expression (1) to expression (2) of fig. 2-9.
Let us focus our attention to the first occurrence of mfib(1) in (1). Using the "syntactic sugar" of our language memoL, it will be written as mfib(1) valueof mfib at 1.
We also explicit-ly write for mfib: (i) its binding:

```
      mfib(4)                                  memo: {}
→*    mfib(3) + mfib(2)                              {}
→*    mfib(2)+mfib(1) + mfib(2)                      {}
→*    (mfib(1)+mfib(0))+mfib(1) + mfib(2)            {}                          (1)
→*    (1+mfib(0))+mfib(1) + mfib(2)                  {<1,1>}                     (2)
→*    (1+1)+mfib(1) + mfib(2)                        {<1,1>,<0,1>}
→     2+mfib(1) + mfib(2)                            {<1,1>,<0,1>,<2,2>}
→     2+1 + mfib(2)  (by memo lookup)                {<1,1>,<0,1>,<2,2>}
→     3 + mfib(2)                                    {<1,1>,<0,1>,<2,2>,<3,3>}
→     3 + 2  (by memo lookup)                        {<1,1>,<0,1>,<2,2>,<3,3>}
→     5                                              {<1,1>,<0,1>,<2,2>,<3,3>,
                                                                          <4,5>}
```

Figure 2-9: Computing the Fibonacci Numbers using Memofunctions.

mfib=λn. <u>if</u> n≤1 <u>then</u> 1 <u>else</u> mfib(n−1) <u>valueof</u> mfib <u>at</u> n−1
+ mfib(n−2) <u>valueof</u> mfib <u>at</u> n−2,

denoted by MFIB, and (ii) its memo, initially empty, denoted by {}. Therefore we have:

<u>let</u> {MFIB, {}} <u>in</u> mfib(1) <u>valueof</u> mfib <u>at</u> 1 . . . (1)

We then get:

<u>let</u> {MFIB, {}} <u>in</u> <u>letmemo</u> {mfib(1)=1} <u>in</u> 1 . . .

where a so-called <u>memoenvironment</u> {mfib(1)=1} is created. It is then "propagated to the left" towards its corresponding <u>environment</u> {MFIB, {}}. After a few steps we get the expression (2):

→* <u>let</u> {MFIB, {<1,1>}} <u>in</u> 1 . . .

where the pair <1,1> has been included in the memo for mfib.

As we will see in the Dynamic Semantics rules for the language memoL, the creation of the <u>letmemo</u> expression is done by the rule VF3 and the "propagation to the left" is done by the rule MPr.

The memoenvironment {mfib(1)=1}, which temporarily stores the information that mfib has value 1 at 1, has been propagated to the left until an enclosing environment with a binding for mfib has been encountered. Indeed by rule MPr we will force "the propagation to the left" in all cases, except when we meet an expression of the form:

<u>let</u> ρ <u>in</u> <u>letmemo</u> {f(a)=b} <u>in</u> e   with $f \in dom(\rho)$,

in which case we update the memo for f in ρ, thereby deriving $\rho 1$, and we get: <u>let</u> $\rho 1$ <u>in</u> e.

The problem of identifying the occurrence of the environment where the computed argument-value pairs have to inserted, will be addressed in the following section. It turns out, that by restricting the occurrences of the free variables in the definitions we can easily solve that problem. As we will see later on, the relevant environment is the "innermost" one where there exists a binding for the same function symbol occurring in the <u>letmemo</u> expression.

We could have followed <u>other approaches</u> for defining the operational semantics of memofunctions.
As we already remarked, we could have introduced extra notions in the semantics definitions choosing, for instance, an SECD style à la Landin, or we could have used more elaborate structures for defining the environments with memo information.

An alternative approach could have been the use of a recursive language different from the language L, as a basis for the definition of its variant with memofunctions.
An interesting choice could have been a language whose definitions are of the form:

(1) $d ::= x{:}\,\tau = e \mid f(x{:}\,\tau_0){:}\,\tau_1 = e \mid f(x{:}\,\tau_0){:}\,\tau_1$ <u>rec=</u> $e \mid d0$ <u>and</u> $d1$

where by <u>rec=</u> we mean that the free occurrences of f in e are recursively defined [Burstall and Lampson 84].

On the contrary the language we considered, following [Plotkin 81], has definitions of the form:

(2) d : : = x: τ = e | f(x: τ_0) : τ_1 = e | <u>rec</u> d | d0 <u>and</u> d1,

where the construction of recursive definitions is more liberal.

For definitions of the form (2), using the structural semantics approach, it seems difficult to give the semantics of "<u>rec</u> d" in terms of the environment to which "d" evaluates, without (i) either introducing extra notions or (ii) using extra primitives (like the substitution operator or the "fix" operator [Burstall and Lampson 84]) or (iii) forcing "at run time" the repetition of the bindings for the recursive functions (and in this case there is not a unique environment with which the memo must be associated).

We think that the approach we will present here has its virtues and it shows also an interesting application of the structural semantics definitions in a case where functional features are mixed with imperative ones (functional, in fact, is the underlying language, and imperative is the updating of memos).

Our language for memofunctions, called <u>memoL</u>, is formally defined by the following sets. It will be defined with reference to the language L (see Sect. 2.1).

<u>Basic Sets</u> . . . (as for the language L)

<u>Derived Sets</u>

Constants con $\in$ Con = N+T

con : : = m | t

Expressions e $\in$ Exp

e : : = . . . (as for the language L) | e_1 <u>valueof</u> f <u>at</u> e_0

Definitions d $\in$ Def

$d ::= x{:}\,\tau{=}e \mid f(x{:}\,\tau_0):\tau_1{=}e \mid d_0$ <u>and</u> d_1

<u>Notes</u>.

(1) The intuitive semantics of "e_1 <u>valueof</u> f <u>at</u> e_0" is that "the value of e_1 is the result of evaluating f at e_0".

A simpler syntax would have been: f\$(e0) where \$(...) denotes the function application with use of the memo information. We do not choose that syntax because, as we will see later, the <u>valueof at</u> construct gives us the advantage of storing for each function evaluation more than one entry in the memo environment.

(2) During evaluation, definitions become environments, as for the language L, and expressions may become of the form: <u>letmemo</u> μ <u>in</u> e, where μ is a MemoEnvironment (later defined).

<u>letmemo</u> expressions are a way of "temporarily" storing argument-value pairs. For example:

<u>let</u> $\{f=\lambda x.\, x^2, \{\langle 2,4\rangle\}\}$ <u>in</u> ... <u>letmemo</u> $\{f(3)=9\}$ <u>in</u> e

stores the information that $3^2=9$. That expression, via the memo propagation rule MPr, will eventually be transformed into:

<u>let</u> $\{f=\lambda x.\, x^2, \{\langle 2,4\rangle, \langle 3,9\rangle\}\}$ <u>in</u> ... e

where the information $3^2=9$ has been "permanently" stored in the environment for f.

(3) We assume that: $FV(e)=\emptyset$ for $x{:}\,\tau{=}e$, and $FV(e)\subseteq\{x,f\}$ for $f(x{:}\,\tau_0):\tau_1{=}e$. (FV(e) denotes the set of free variables of the expression e of memoL, which will be formally defined later.) Notice that, in order to satisfy the above hypotheses for the free variables, we may transform our programs in the way we suggested in the previous section 2.2.1. □

Under the hypotheses of point (3) above, it turns out that we need not distinguish between recursive and non-recursive definitions and we may consider <u>all functions to be recursively defined</u>. Indeed we assume that if an occurrence of a free variable f exists on the right hand side of the definition $f(x:\tau_0):\tau_1 = e$, it has to be considered as recursively defined (see Section 2.4). This is why there is not a <u>rec</u> d clause for Definitions in memoL. Since all function definitions are recursive, a better syntax would have been $f(x:\tau_0):\tau_1$ <u>rec=</u> e, but for simplicity we adopted the shorter form given above.

The computations evoked by the programs written in the language memoL, are quite efficient and via the use of memofunctions the repeated evaluation of recursive function calls is avoided.

2.3.1 Static Semantics for memoL

Through the static semantics analysis we perform the type-checking and we will discard expressions and definitions which are not well-formed.

We need to define some auxiliary sets. The sets of 1) Variables: V, 2) Expressible Types: ETypes, 3) Denotable Types: DTypes and 4) Type-Environments: TEnv are the ones introduced for the language L.

5) set of <u>Denotable Values</u>: DVal. $dv \in$ DVal.

DVal = N + T + (Abstracts$\times$Memos) where Memos = P(Con$\times$Con). mem $\in$ Memos. (As usual P(S) denotes the set of subsets of S).

6) set of <u>Environments</u>: Env. $\rho \in$ Env.

$$\mathrm{Env} = \sum_{V \subseteq \mathrm{Var}} \mathrm{Env}_V \text{ where } \mathrm{Env}_V = V \longrightarrow \mathrm{DVal}.$$

We write $\rho{:}V$ for denoting $\rho \in \mathrm{Env}_V$.

Given a functional variable, say f, we bind it to a pair whose first component is the corresponding lambda expression, and the second component is the set of the argument-value pairs already computed for f.

If $\rho(f) = \langle abstract, \{\langle c_0, c_1\rangle, \ldots\}\rangle$ we will also write ρ as follows: $\{f{=}abstract, \{\langle c_0, c_1\rangle, \ldots\}; \ldots\}$.

7) set of <u>MemoEnvironments</u>: MemoEnv. $\mu \in$ MemoEnv.

$$\mathrm{MemoEnv} = \sum_{V \subseteq \mathrm{Var}} \mathrm{MemoEnv}_V \text{ where } \mathrm{MemoEnv}_V = V \longrightarrow P(\mathrm{Con}\times\mathrm{Con}).$$

We write μ: V for denoting $\mu \in \mathrm{MemoEnv}_V$.

If $\mu(f) = \{\langle c_0, c_1\rangle, \langle c_2, c_3\rangle, \ldots\}$ we will also write μ as: $\{f(c_0)=c_1, f(c_2)=c_3, \ldots\}$.

Given a functional variable f, $\mu(f)$ gives us the set of the corresponding argument-value pairs already computed.

(The notion of MemoEnvironments has been introduced for reasons of simplicity: see the related discussion in Section 2.3.3).

Example 21.

Given W={x, f}, $\rho=\{x=5;\ f=\lambda z\colon \mathrm{int}.\ z+2\colon \mathrm{int}, \{\langle 6, 8\rangle, \langle 13, 15\rangle\}\} \in \mathrm{Env}_W$.

Given V={f}, $\mu 1=\{f(6)=8\}$ and $\mu 2=\{f(6)=8,\ f(13)=15\}$ are both members of $\mathrm{MemoEnv}_V$.

8) set of free variables FV and defined variables DV.

Those sets are defined as for the language L. The only differences are as follows:

	e1 <u>valueof</u> f <u>at</u> e_0	<u>letmemo</u> μ <u>in</u> e
FV	FV(e1) ∪ FV(e_0)	FV(e)

	x: τ=e	f(x: τ_0) : τ_1=e	ρ (if ρ: V)
FV	FV(e) (1)	FV(e)\{x} (2)	$\bigcup_{x \in V}$ FV($\underline{\pi 1}(\rho(x))$) (3)

Notes.

(1) By hypothesis we have: FV(e) = $\emptyset$.

(2) By hypothesis we have: FV(e) $\subseteq$ {x, f}.

(3) $\underline{\pi 1}$ is the identity function if $\rho(x) \in$ N+T. If $\rho(x) \in$ Abstracts×Memos, $\underline{\pi 1}$ is the first projection function. We assume that if $\lambda x\colon \tau_0.\ e\colon \tau_1=\underline{\pi 1}(\rho(f))$ then $FV(\lambda x\colon \tau_0.\ e\colon \tau_1)$ = FV(e)\{x} $\subseteq$ {f}. $\underline{\pi 1}$ allows us to obtain the λ-expression bound to a function symbol and discard the memo.

We also extend the definition of the defined variables by assuming that for any $\mu \in \mathrm{MemoEnv}_V$, DV($\mu$) = V.

As for the expressions and definitions of the language L we introduce the following formulas:

$\alpha \vdash_V e:\tau$, $\alpha \vdash_V d$, and $\vdash_V d:\beta$, with the usual meaning.

We will feel free to write $\alpha\vdash\ldots$ instead of $\alpha\vdash_V\ldots$ where V=dom(α).

We will only give the <u>Static Semantics</u> rules which are different from (or have to be added to) the rules of the language L.

Let us assume that (i) $\alpha \in TEnv_V$, (ii) $FV(e) \subseteq V$ for any expression e s.t. $\alpha\vdash e:\tau$, and (iii) $FV(d) \subseteq V$ for any definition d s.t. $\alpha\vdash d$.

<u>Memo</u>

$$\frac{\alpha \vdash e_1:\tau_1, \quad \alpha\vdash e_0:\tau_0}{\alpha \vdash e_1 \text{ \underline{valueof} } f \text{ \underline{at} } e_0:\tau_1} \qquad \text{if } \alpha(f) = \tau_0 \rightarrow \tau_1$$

This rule says that e_1 <u>valueof</u> f <u>at</u> e_0 is well-formed if e_1 and e_0 are well-formed, and the type of f agrees with those of e_0 and e_1.

<u>Letmemo</u>

For some type-environment β: W

$$\frac{\alpha \vdash e:\tau, \qquad \mu:\beta}{\alpha \vdash (\text{\underline{letmemo} } \mu \text{ \underline{in} } e):\tau}$$

where the predicate $\mu:\beta$ for $\mu \in MemoEnv_W$ is defined as follows:

$\mu:\beta \equiv \forall f \in W.\ \forall\langle a,b\rangle \in \mu(f).\ \forall\tau_0,\tau_1 \in ETypes.$

$\beta(f)=\tau_0\rightarrow\tau_1$ <u>iff</u> ($\beta \vdash a:\tau_0$ <u>and</u> $\beta \vdash b:\tau_1$).

The above rule says that a <u>letmemo</u> μ <u>in</u> e expression is well-formed if (i) e is well-formed and (ii) for any memo information "f(a)=b" in μ we have that f has type $\tau_0\rightarrow\tau_1$ iff a has type τ_0 and b has type τ_1.

Therefore the predicate $\mu:\beta$ is true iff the memo information in μ "agrees" with the type-environment β.

For instance:

$$\frac{\{f=int\rightarrow int\}\vdash f(2)+5: int, \quad \{f(3)=8\}: \{f=int\rightarrow int\}}{\{f=int\rightarrow int\}\vdash (\underline{letmemo}\ \{f(3)=8\}\ \underline{in}\ f(2)+5): int}.$$

(We have given the static semantic rules for letmemo expressions, although they arise during execution only, for technical reasons. The same applies for environments.)

Function definitions

1. $$\frac{\alpha[\{x=\tau_0,\ f=\tau_0\rightarrow\tau_1\}]\vdash e:\tau_1}{\alpha\vdash f(x:\tau_0):\tau_1 = e}$$

2. $\vdash (f(x:\tau_0):\tau_1 = e): \{f = \tau_0\rightarrow\tau_1\}$

Note. In the premise of rule 1. we included the type information for f, because by hypothesis the definition f(x)=e in memoL is a recursive definition.

Environments.

The rules for Environments are like the ones for the language L.

We need only to define $\alpha\vdash\rho(x):\beta(x)$ in the case where $\rho(x)\in Abstracts\times Memos$.

Suppose that for $f\in V$ and $\rho(f)=\langle\lambda x:\tau_0.\ e:\tau_1,\ mem\rangle$. We have by definition: $\alpha\vdash\langle\lambda x:\tau_0.\ e:\tau_1, mem\rangle: dt$ <u>iff</u> $(\alpha[\{f=\tau_0\rightarrow\tau_1\}]\vdash(\lambda x:\tau_0.\ e:\tau_1):\tau_0\rightarrow\tau_1$ <u>and</u> $dt=\tau_0\rightarrow\tau_1$ <u>and</u> $\forall\langle a,b\rangle\in mem.\ \alpha\vdash a:\tau_0$ and $\alpha\vdash b:\tau_1.)$.

Having given the Static Semantic rules for the language memoL we have defined the expressions and definitions which we consider to be well-formed. The Dynamic Semantics rules given below, will be applied only to them.

2.3.2 Dynamic Semantics for memoL

We first introduce, as for the language L, the notions of 1) agreement of an environment ρ with a type-environment α, denoted by $\rho:\alpha$, 2) well-formedness of expressions (or definitions), 3) Expression Configurations and Definition Configurations w.r.t. a type-environment α, 4) Terminal Configurations and 5) Transition Systems w.r.t. a type-environment α. The only difference is that, instead of DVal=N+T+Abstracts, we now consider DVal=N+T+(Abstracts×Memos).

Let us define an auxiliary notion: the Atomic Expression-Contexts, called AEContexts, ranged over by C[ ].

An atomic expression-context is an expression with a "hole" instead of a "subexpression at depth 1".

For example, "x+[ ]" and "<u>if</u> x=1 <u>then</u> [ ] <u>else</u> 3" are Atomic Expression-Contexts, but "x+(y+[ ])" is not. The formal definition of AEContexts is as follows:

$$\begin{array}{lll} C[\,] ::= \mathrm{bop}(\ldots,[\,],\ldots) & \mid \underline{\mathrm{if}}\ [\,]\ \underline{\mathrm{then}}\ e_1\ \underline{\mathrm{else}}\ e_2 & \mid \underline{\mathrm{let}}\ \rho\ \underline{\mathrm{in}}\ [\,] \\ & \mid \underline{\mathrm{if}}\ e_0\ \underline{\mathrm{then}}\ [\,]\ \underline{\mathrm{else}}\ e_2 & \mid \underline{\mathrm{let}}\ x{:}\ \tau = [\,]\ \underline{\mathrm{in}}\ e \\ & \mid \underline{\mathrm{if}}\ e_0\ \underline{\mathrm{then}}\ e_1\ \underline{\mathrm{else}}\ [\,] & \mid \underline{\mathrm{let}}\ f(x{:}\ \tau_0):\tau_1 = [\,]\ \underline{\mathrm{in}}\ e \\ & \mid [\,]\ \underline{\mathrm{valueof}}\ f\ \underline{\mathrm{at}}\ e_0 & \mid f[\,] \\ & \mid e_1\ \underline{\mathrm{valueof}}\ f\ \underline{\mathrm{at}}\ [\,] & \end{array}$$

The following rules for the definition of the <u>Dynamic Semantics</u> of the language memoL have to be added to (or they replace) the rules we have already given for the language L.

As usual, we assume $c_0, c_1, \ldots \in$ Con and $e \in$ Exp.

We will write $\rho\vdash\ldots$ instead of $\rho\vdash_\alpha\ldots$

<u>Valueof</u>

VF1.
$$\frac{\rho \vdash e_0 \longrightarrow e'_0}{\rho \vdash e_1\ \underline{\mathrm{valueof}}\ f\ \underline{\mathrm{at}}\ e_0 \longrightarrow e_1\ \underline{\mathrm{valueof}}\ f\ \underline{\mathrm{at}}\ e'_0}$$

VF2. $$\frac{\rho \vdash e_1 \longrightarrow e'_1}{\rho \vdash e_1 \text{ \underline{valueof} } f \text{ \underline{at} } e_0 \longrightarrow e'_1 \text{ \underline{valueof} } f \text{ \underline{at} } e_0}$$

VF3. $\rho \vdash c_1$ <u>valueof</u> f <u>at</u> $c_0 \longrightarrow$ <u>letmemo</u> $\{f(c_0)=c_1\}$ <u>in</u> c_1

if $c_0, c_1 \in$ Con and $\langle c_0, c_1\rangle \notin \pi 2(\rho(f))$

VF4. $\rho \vdash e$ <u>valueof</u> f <u>at</u> $c_0 \longrightarrow c_1$

if $c_0, c_1 \in$ Con and $\langle c_0, c_1\rangle \in \pi 2(\rho(f))$

Rules VF1 and VF2 are obvious because we need to evaluate e_1 and e_0 when evaluating e_1 <u>valueof</u> f <u>at</u> e_0.

Rule VF3 generates a <u>letmemo</u> expression to be "propagated to the left".

Rule VF4 says that if the pair $\langle c_0, c_1\rangle$ already exists in the memo for f, we need not update that memo (therefore no letmemo expression should be generated) and we can substitute c_1 for the expression e to be evaluated.

<u>Memopropagation</u>

MPr. $\rho \vdash C[$<u>letmemo</u> μ <u>in</u> $e] \longrightarrow$ <u>letmemo</u> μ <u>in</u> $C[e]$

for any atomic expression context $C[\ ] \neq$ <u>let</u> ρ <u>in</u> $[\ ]$ s.t. $DV(\rho) \cap DV(\mu) \neq \phi$.

Note. If $\mu = \{f(c_0)=c_1\}$ then the condition $DV(\rho) \cap DV(\mu) \neq \phi$ is equivalent to $f \in DV(\rho)$.

Using the MPr rule, the memo information μ is "propagated to the left", towards the corresponding environment.

<u>Memoupdating</u>

MU. $\rho \vdash$ <u>let</u> ρ_0 <u>in</u> <u>letmemo</u> $\{f(c_0)=c_1\}$ <u>in</u> $e \longrightarrow$ <u>let</u> ρ_1 <u>in</u> e if $f \in DV(\rho_0)$

where $\rho_1 = \rho_0[f \mid \langle \pi 1(\rho_0(f)), \pi 2(\rho_0(f)) \cup \{\langle c_0, c_1\rangle\}\rangle]$

Note. The pair $\langle c_0, c_1\rangle$ is inserted in the memo component of the binding for f in ρ_0. MU is the rule which can be applied to letmemo expressions when we cannot apply MPr.

It is essential that during the evaluation of a given expression, only one environment with a binding for f occurs in it, so that the environment with which the pair $\langle c_0, c_1\rangle$ should be associated, is uniquely determined. (That fact is not true for the language L, while it is true for the language memoL).

Function and Memo application

A1. $$\frac{\rho \vdash e \longrightarrow e'}{\rho \vdash f(e) \longrightarrow f(e')}$$

A2. $\rho \vdash f(c_0) \longrightarrow c_1$ if $c_0, c_1 \in$ Con and $\langle c_0, c_1\rangle \in \pi 2(\rho(f))$

A3. $\rho \vdash f(c_0) \longrightarrow$ let $x: \tau_0 = c_0$ in e where $\pi 1(\rho(f)) = \lambda x: \tau_0 . e: \tau_1$
if $c_0 \in$ Con and $\langle c_0, c_1\rangle \notin \pi 2(\rho(f))$ for some c_1

Note. Rule A1 evaluates the argument of a function application.
Via rule A3 we evaluate the application in the usual way, if no memo for the argument c_0 occurs in the binding of f. Otherwise via rule A2, we use the memo information and we produce "in one step" the result.

Definitions. Function definitions

F1. $\rho \vdash f(x: \tau_0) : \tau_1 = e \longrightarrow \{f = \langle \lambda x: \tau_0 . e: \tau_1, \phi\rangle\}$

Note. The definition is transformed into an environment with an initial "empty memo", i.e. with no argument-value pairs in it.

2.3.3 Alternative rules for memoL

This last section is about an alternative definition of the semantics rules for memofunctions which turns out to be equivalent to the one we presented above. We describe it here because it shows that the use of the letmemo expressions and the memoenvironments does not contradict the principle of not introducing extra notions in the structural operational semantics definitions. As we will see below, that use is only a "temporary" way of denoting the changes of environments which can be expressed, maybe in a less intuitive manner, without letmemo's or memoenvironments.

Instead of using rules VF3, MPr and MU we could use the following one:

VF3'. $\vdash \langle \rho_0, c_1$ $\underline{\text{valueof}}$ f $\underline{\text{at}}$ $c_0\rangle \longrightarrow \langle \rho_1, c_1\rangle$

where (i) an "old" configuration, i.e. an "old" <environment, expression> pair, is transformed into a "new" one with a possible change of the environment, and (ii) $\rho_1 = \rho_0[f \mid \langle \pi1(\rho_0(f)), \pi2(\rho_0(f)) \cup \{\langle c_0, c_1\rangle\}\rangle]$,

i.e. ρ_1 is like ρ_0 except that the memo information for the function f has been updated by the pair $\langle c_0, c_1\rangle$.

In order to denote that updating of the environment, in what follows we will use the shorter form: $\rho_1 = \rho_0\{f(c_0)=c_1\}$.

Now, since environments may be changed, in order to use VF3', we need to modify the other rules of the language memoL. In particular, rules A1 and L2 become:

$$\text{A1'}.\quad \frac{\vdash \langle \rho, e\rangle \longrightarrow \langle \rho', e'\rangle}{\vdash \langle \rho, f(e)\rangle \longrightarrow \langle \rho', f(e')\rangle} \quad \text{and}$$

$$\text{L2'}.\quad \frac{\vdash \langle \rho[\rho_0], e\rangle \longrightarrow \langle \rho', e'\rangle}{\vdash \langle \rho, \underline{\text{let}}\ \rho_0\ \underline{\text{in}}\ e\rangle \longrightarrow \langle \sigma, \underline{\text{let}}\ \sigma_0\ \underline{\text{in}}\ e'\rangle}$$

where σ and σ_0 are defined as follows.

Let ρ: V, ρ_0: V_0, and ρ' be $(\rho[\rho_0])\{f(c_0)=c_1\}$. We have:

(i) $\underline{\text{if}}$ $f \in V_0$ $\underline{\text{then}}$ $\sigma=\rho$ and $\sigma_0=\rho_0\{f(c_0)=c_1\}$, because the relevant environment for f is ρ_0 and we must update the memo of f in ρ_0;

(ii) $\underline{\text{if}}$ $f \in V-V_0$ $\underline{\text{then}}$ $\sigma=\rho\{f(c_0)=c_1\}$ and $\sigma_0=\rho_0$, because if $f \notin dom(\rho_0)$ the updating of the memo must occur in the environment of the given configuration.

Notice that, if $\rho[\rho_0]=\rho'$, we have: $\sigma=\rho$ and $\sigma_0=\rho_0$.
As we have seen, the definition of σ and σ_0 is a bit complex. Therefore we prefer, instead of the rule L2', the simpler rules VF3, MPr and MU. Indeed they express in an algorithmic way the same changes of environments evoked by L2'. □

2.3.4 An Example and Some Remarks on the Semantics of memoL

Here is an example of application of the dynamic semantic rules for the language memoL. It is an extended analysis of the example given in fig. 2-9.

Example 22.
Let us consider the following program for computing the fibonacci function mfib(x) for x=4. (The name mfib instead of fib is because we use the language memoL.)

```
let mfib(x: int) : int = if x=0 then 1 else (if x=1 then 1 else
   (mfib(x-1) valueof mfib at x-1) + (mfib(x-2) valueof mfib at x-2))
in mfib(4).
```

For simplicity reasons from now on we will not write the type information. Let IF be the expression defining mfib(x). We have:

$\phi \vdash$ let mfib(x)=IF in mfib(4) $\longrightarrow$ let {mfib=λx. IF, ϕ} in mfib(4)

We obtain:

```
ϕ ⊢ let {mfib=λx. IF, ϕ} in let {x=4} in if x=0 then 1 else
    (if x=1 then 1 else (mfib(x-1) valueof mfib at x-1) +
    (mfib(x-2) valueof mfib at x-2))
    →⁺ let {mfib=λx. IF, ϕ} in let {x=4} in
         (mfib(3) valueof mfib at 3) +
         (mfib(x-2) valueof mfib at x-2)
```

Here $\longrightarrow^+$ denotes the transitive closure of $\longrightarrow$.
(The above derivation is one among many that are possible.)
By applying the rules for let-expressions, valueof-expressions and basic operators, we obtain:

$\rightarrow^{+}$ let {mfib=λx. IF, ϕ} in let {x=4} in
(let {x=3} in (mfib(2) valueof mfib at 2) + (mfib(x−2) valueof mfib at x−2)
valueof mfib at 3) + (mfib(x−2) valueof mfib at x−2)

$\rightarrow^{+}$ let {mfib=λx. IF, ϕ} in let {x=4} in
(let {x=3} in (let {x=2} in (mfib(1) valueof mfib at 1) +
(mfib(x−2) valueof mfib at x−2) valueof mfib at 2)
+ (mfib(x−2) valueof mfib at x−2) valueof mfib at 3)
+ (mfib(x−2) valueof mfib at x−2)

$\rightarrow^{+}$ let {mfib=λx. IF, ϕ} in let {x=4} in
(let {x=3} in (let {x=2} in (let {x=1} in 1 valueof mfib at 1) +
(mfib(x−2) valueof mfib at x−2) valueof mfib at 2) +
(mfib(x−2) valueof mfib at x−2) valueof mfib at 3) +
(mfib(x−2) valueof mfib at x−2)

$\rightarrow$ as before with "1" instead of "let {x=1} in 1"

$\rightarrow$ as before with "letmemo {mfib(1)=1} in 1" instead of "1 valueof mfib at 1"

$\rightarrow^{+}$ let {mfib=λx. IF, {<1, 1>} in let {x=4} in
(let {x=3} in (let {x=2} in 1 +(mfib(x−2) valueof mfib at x−2)
valueof mfib at 2) + (mfib(x−2) valueof mfib at x−2)
valueof mfib at 3)+(mfib(x−2) valueof mfib at x−2)

Notice that we propagated the memo information mfib(1)=1, before carrying on any other transformation of subexpressions.

The evaluation may go on in the following way:

$\rightarrow^{+}$ . . . (let {x=2} in (1+(mfib(0) valueof mfib at 0))) valueof mfib at 2 . . .

$\rightarrow^{+}$ . . . (let {x=2} in (1+((let {x=0} in 1)
valueof mfib at 0))) valueof mfib at 2 . . .

$\rightarrow$ . . . (let {x=2} in (1+(1 valueof mfib at 0))) valueof mfib at 2 . . .

$\rightarrow$ . . . (let {x=2} in (1+(letmemo {mfib(0)=1} in 1))) valueof mfib at 2 . . .

$\rightarrow^{+}$ let {mfib=λx. IF, {<0, 1>, <1, 1>}} in let {x=4} in
(let {x=3} in ((let {x=2} in (1+1)) valueof mfib at 2) + (mfib(x−2) valueof

mfib at x-2) valueof mfib at 3) + (mfib(x-2) valueof mfib at x-2)

$\rightarrow$ as before with "2" instead of "1+1"

$\rightarrow$ as before with "2" instead of "let {x=2} in 2"

$\rightarrow$ as before with "letmemo {mfib(2)=2} in 2" instead of "2 valueof mfib at 2"

$\rightarrow^+$ let {mfib=λx. IF, {<2, 2>, <0, 1>, <1, 1>}} in let {x=4} in

(let {x=3} in 2+(mfib(x-2) valueof mfib at x-2) valueof mfib at 3) +

(mfib(x-2) valueof mfib at x-2)

by following the same sequence of reductions as above, we obtain:

$\rightarrow^+$ let {mfib=λx. IF, {<2, 2>, <0, 1>, <1, 1>}} in let {x=4} in

(let {x=3} in 2+(mfib(1) valueof mfib at 1) valueof mfib at 3) +

(mfib(x-2) valueof mfib at x-2)

and in one step of deduction, by using the memo information, we get:

$\rightarrow$ . . . ((let {x=3} in 2+(1 valueof mfib at 1)) valueof mfib at 3)+ . . .

$\rightarrow$ . . . ((let {x=3} in 2+1) valueof mfib at 3) + . . .

Notice that no memo updating is necessary (and therefore no generation of a letmemo expression occurred) because in the memo for mfib we have already the pair <1, 1>. Then we have:

$\rightarrow^+$ let {mfib=λx. IF, {<2, 2>, <0, 1>, <1, 1>} in let {x=4} in

(3 valueof mfib at 3) + (mfib(x-2) valueof mfib at x-2)

$\rightarrow$ as before with "letmemo {mfib(3)=3} in 3" instead of "3 valueof mfib at 3"

$\rightarrow^+$ let {mfib=λx. IF, {<3, 3>, <2, 2>, <0, 1>, <1, 1>} in let {x=4}

in 3+mfib(x-2) valueof mfib at x-2

$\rightarrow$ let . . . in let {x=4} in 3+(mfib(2) valueof mfib at 2)

$\rightarrow$ let . . . in let {x=4} in 3+(2 valueof mfib at 2)

In the last step we used the memo information for mfib(2). Then we get:

$\rightarrow$ let . . . in let {x=4} in 3+2 $\rightarrow^+$ 5. □

Now we would like to make a few remarks about the evaluation we have shown in the above example and about the given dynamic semantics rules.

Remark 1. The evaluation sequence for computing the value of mfib(4) is one of the many that are possible. We will see that the Church-Rosser property holds for our dynamic semantics rules.

Remark 2. In the evaluation sequence we presented, the updating of the memo component for mfib is done "as soon as possible". Indeed we gave priority to the rules VF3, MPr and MU. This decision makes it possible to avoid the recomputation of identical function applications.

These priority choices were also adopted in our Prolog implementation of the memoL semantics (see Sect. 2.6).

If the memo information is stored and used according to the priorities we have indicated, a complexity reduction from exponential time to linear time can be achieved, as the Fibonacci example shows.

(Similar work of establishing priorities among rules in order to find the optimal reduction sequences has been done for the λ-calculus by J. J. Lévy [Lévy 80] and others.)

Remark 3. When one uses the memoL language for avoiding repeated evaluations of recursive functions, the construct e_1 valueof f at e_0 is used with $e_1=f(e_0)$. Therefore the evaluation of e_1 will result in a repeated evaluation of the argument e_0.

In order to avoid such inconvenience one may extend the syntax of the memoL language by considering expressions of the form: "memo f(e)" as an abbreviation for "$f(e_0)$ valueof f at e_0". The corresponding rules are:

MA1. $$\frac{\rho \vdash e \rightarrow e'}{\rho \vdash \underline{\text{memo}}\ f(e) \rightarrow \underline{\text{memo}}\ f(e')}$$

MA2. $\rho \vdash \underline{\text{memo}}\ f(c) \rightarrow f(c)\ \underline{\text{valueof}}\ f\ \underline{\text{at}}\ c$

Another solution to that problem is to write let $z=e_0$ in $(f(z)$ valueof f at $z)$ instead of $f(e_0)$ valueof f at e_0.

It is not difficult to prove the semantic equivalence of all those alternative definitions, by using for instance, v-transition systems [Hennessy and Wei 82].

Remark 4. Constructs of the form e_1 valueof f at e_0 may have an interesting use for denoting "communications" among evaluations of expressions.

In particular we may write the following definition for computing the binomial coefficients, taking advantage of the fact that bin(n, m)=bin(n, n-m):

```
bin(n,m) = if m=0 or n=m then 1
           else(bin(n-1,m-1) valueof bin at (<n-1,m-1> and <n-1,n-m>)) +
               (bin(n-1,m) valueof bin at (<n-1,m> and <n-1,n-m-1>)),
```

where we used the notation: e valueof f at (e_0 and e_1 and ...) as an abbreviation for ((e valueof f at e_0) valueof f at e_1)...

According to the above definition, after the computation of bin(n-1, m-1) and bin(n-1, m), we also know the values of bin(n-1, n-m) and bin(n-1, n-m-1).

In fig. 2-10 we have shown a sequence of calls which may be evoked by

let bin(n, m)=...(as above)... in bin(7, 3).

Notice that an improvement of efficiency has been achieved with respect to both the usual recursive definition and the definition which avoids commutative redundancy [Cohen 83].

Let [n, m] denote the binomial coefficient bin(n, m).

With reference to fig. 2-10 the sequence of recursive calls, evaluated without use of the memo information when computing bin(7, 3), is: [7,3], [6,2], [5,1], [4,0], [4,1], [3,0], [3,1], [2,0], [2,1], [1,0], [5,2], [4,2]. They are denoted by solid arrows.

The sequence of memoupdatings is: [4,0], [4,4], [3,0], [3,3], [2,0], [2,2], [1,0], [1,1], [2,1], [3,1], [3,2], [4,1], [4,3], [5,1], [5,4], [4,2], [5,2], [5,3], [6,2], [6,3].

In fig. 2-10 dashed arrows denote the memo updatings. Underlined values are stored in the memo-environment because of the valueof at constructs.

$$\begin{tikzcd}
 & & & {[7,3]} \arrow[dl] & & & & & \\
 & & {[6,2]} \arrow[rr, dashed] \arrow[dl] \arrow[dr] & & {[\underline{6,3}]} & & & & \\
 & {[5,1]} \arrow[rrrrrr, dashed, bend left] \arrow[dl] \arrow[dr] & & {[5,2]} \arrow[rr, dashed] \arrow[dl] \arrow[dr] & & {[\underline{5,3}]} & & {[\underline{5,4}]} & \\
{[4,0]} \arrow[rrrrrrrr, dashed, bend left] & & {[4,1]} \arrow[rrrr, dashed, bend left] \arrow[dl] \arrow[dr] & & {[4,2]} \arrow[dl] & & {[\underline{4,3}]} & & {[\underline{4,4}]} \\
 & {[3,0]} \arrow[rrrrrr, dashed, bend left] & & {[3,1]} \arrow[rr, dashed] \arrow[dl] \arrow[dr] & & {[\underline{3,2}]} & & {[\underline{3,3}]} & \\
 & & {[2,0]} \arrow[rrrr, dashed, bend left] & & {[2,1]} \arrow[dl] \arrow[dr] & & {[\underline{2,2}]} & & \\
 & & & {[1,0]} \arrow[rr, dashed] & & {[\underline{1,1}]} & & &
\end{tikzcd}$$

Figure 2-10: Computing binomial coefficients. Use of memo for [n, m] = [n, n-m]

2.4 Some properties of the operational semantics definitions and program annotation methodology

In this section we will study some properties of the operational semantics definitions we have given in the previous sections. Some of those properties are quite important as, for instance, the fact that for a given class of expressions the operational semantics of the language memoL with memo information "agrees" (in a sense later specified) with the operational semantics of the language L, while avoids redundant evaluations of recursive calls.
We also suggest a programming methodology based on annotations as follows (see also fig. 2-11).

$$\begin{tikzcd}
P \in C(L) \arrow[rr, "M"] \arrow[dr, "S_L"'] & & M(P) \in C(memoL) \arrow[dl, "S_{memoL}"] \\
& D &
\end{tikzcd}$$

Figure 2-11: The Program Annotation Methodology for avoiding repeated functions calls.

We will define a class C(L) of programs written in the language L and a method M for annotating them, thereby deriving the class C(memoL) of programs in the language memoL, so that the evaluation of any program P ∈ C(L) can be more efficiently performed. Correctness is preserved by requiring the commutativity of the diagram in fig. 2-11.
The class C(L) of programs is defined by imposing some restrictions on the occurrences of the free variables in the definitions. The translation M consists in substituting some occurrences of the function applications "f(e)" by "f(e) <u>valueof</u> f <u>at</u> e".

In what follows we will give more details about the definitions of C(L) and M which make the diagram of fig. 2-11 commute, and we will propose two such pairs <C(L), M>.

As we already mentioned, more general results could be obtained by defining an operational semantics for a language with memofunctions if we did not stick to the principle of not introducing extra notions in the structural operational semantics definitions.

2.4.1 Relationship between the language L and memoL. Proposal of a program transformation methodology.

The relationship between the language L and the language memoL can be best understood in the frameworks of the program transformation and the program annotation methodologies.

Given a program P in a class C(L) of programs, written in the language L, in order to improve its efficiency, we may derive the program T(P), called the transformed program, so that efficiency of execution is improved, while correctness is preserved.

$$\begin{tikzcd}
 & T(P) \in C(L) \arrow[dr, "sem"] & \\
P \in C(L) \arrow[ur, "T"] \arrow[rr, "sem"] \arrow[dr, "M"'] & & D \\
 & M(P) \in C(memoL) \arrow[ur, "memosem"'] &
\end{tikzcd}$$

Figure 2-12: Relating two operational semantics: sem for the language L and memosem for the language memoL

We need the upper triangle of the diagram of fig. 2-12 to commute, and we need that the evaluation of T(P) is more efficient than the one of P.

The efficiency in evaluating C(L) programs can be inferred from the definition of the semantics function "sem", if we assume that the interpreter for the language L is directly derived from that definition [Cardelli 83].

Instead of transforming programs we may annotate them [Schwarz 82], i.e. translate them into programs in an extended language, for which we provide, once and for all, an efficient evaluator. Therefore given a program $P \in C(L)$, we can translate it into the program M(P) in the language memoL, and if the lower triangle of fig. 2-12 commutes, by evaluating M(P) using "memosem" we get the desired result in an efficient way. (In our case redundant evaluations of recursive calls are avoided.)
The translation M from the language L to the language memoL we will propose is quite straightforward and it does not require any difficult step, unlike the "eureka steps" of the program transformation technique.

The "transformation technique" and the "annotation technique" can also be compared in the following way. Let us assume that, given a semantics function s, we have for each program p an associated complexity measure $c_s(p)$, which in intuitive terms, gives us the cost of evaluating p using s.
In the transformation technique we have a fixed semantics function, say sem, and we look for a transformed program T(p) such that sem(T(p)) = sem(p) and $c_{sem}(T(p)) < c_{sem}(p)$.
In the annotation technique we factorize, so to speak, the "transformation" T into two steps: an annotation "M" and an improved semantics "impsem" (for an extended language) with the aim of getting: $c_{impsem}(M(p)) < c_{sem}(p)$, and impsem(M(p)) = sem(p). (see fig. 2-13).
That factorization gives us:
i) more flexibility in making program improvements, in that we are not forced

to code the improvements as changes in the program text, but we can get them through the semantics of the extended language, and
ii) it allows us to structure and classify program improvements as deriving from classes of annotations.

$$\begin{tikzcd}
& T(p) \arrow[dr, "sem"] & \\
p \arrow[ur, "T"] \arrow[rr, "sem"] \arrow[dr, "M"'] & & d \\
& M(p) \arrow[ur, "impsem"'] &
\end{tikzcd}$$

Program Transformation:

$c_{sem}(T(p)) < c_{sem}(p)$

Program Annotation:

$c_{impsem}(M(p)) < c_{sem}(p)$

Figure 2-13: Program Transformation and Program Annotation.

With reference to the fig. 2-12 we need to show that: (1) the given operational semantics definitions for our languages L and memoL make the diagram commute, and (2) the memoL operational semantics avoid redundant evaluation of recursive functions calls.

Point (1) holds for two classes of programs as introduced by the Definitions 2 and 9 (see Theorems 7 and 11). Notice also that most programs which do not belong to those classes can easily be transformed into equivalent programs belonging to them, by using the techniques mentioned in Section 2.2.

Point (2) is immediate, because the VF4 rule (see Section. 2.3) avoids repeated evaluations of recursive function calls.

In order to prove the main theorems, i.e the agreement of the semantics definitions of the languages L and memoL for the two classes of programs we consider, let us first show that the Church-Rosser property holds for those definitions. By that property we can pay no attention to the nondeterminism of the semantics rules.

2.4.2 The Church-Rosser property for the operational semantics of the language L

Let us consider the case in which functional variables have arity n=1. (The general case for arity $n \geq 0$ can be shown with no extra difficulties).

The rules of the dynamic semantics definition of L are deterministic in the sense that:

$\forall \rho \in Env_V$. $\forall e \in Exp$. $\exists$ a unique e' s.t. $\rho \vdash e \rightarrow e'$, if $\rho \vdash e \rightarrow e'$ holds for some e',

with the only exception of the rule B1 for the basic operators and the rules AND1 and AND2 for and-definitions.

Given a set S with a binary relation $\rightarrow \subseteq S \times S$ we say that (S, $\rightarrow$) is Church-Rosser iff $\forall$ s,u,v $\in$ S if s $\rightarrow^*$ u and s $\rightarrow^*$ v then $\exists$t $\in$ S s.t. u $\rightarrow^*$ t and v $\rightarrow^*$ t. $\rightarrow^*$ denote the reflexive transitive closure of $\rightarrow$.

We can show that the set of configurations Γ with the relation $\rightarrow$ is Church-Rosser by applying the following "strong confluence" result.

(S, $\rightarrow$) is strongly confluent iff $\forall$s,u,v $\in$ S if s $\rightarrow$ u and s $\rightarrow$ v then for some t $\in$ S u $\rightarrow^{0,1}$ t and v $\rightarrow^{0,1}$ t, where $\rightarrow^{0,1}$ denotes the reflexive closure of $\rightarrow$.

Strong confluence implies the Church-Rosser property, [Huet 80].

By the nondeterminism we may have from both $\rho \vdash e0 \rightarrow e0'$ and $\rho \vdash e1 \rightarrow e1'$:

$\rho \vdash$ bop(...,e0,....,e1...) $\rightarrow$ bop(...,e0',....,e1,...) and

$\rho \vdash$ bop(...,e0,....,e1,...) $\rightarrow$ bop(...,e0,....,e1',...).

From both bop(...,e0',....,e1,...) and bop(...,e0,....,e1',...) we obtain bop(...,e0',....,e1',...), because the redexes e0 and e1 are disjoint. One can analogously deal with the non-determinism arising from and-definitions.

This concludes the proof of the Church-Rosser property.

Notice that the above proof does not require the rewriting system to be noëtherian [Huet and Oppen 80].

2.4.3 The Church-Rosser property for the operational semantics for the language memoL

The proof is similar to the above one for the language L.

The only nondeterministic rules, which are in memoL and not in L, are the rules for the valueof at constructs and the memo rules. There are the following cases:

(1) one may apply the rule VF1 or the rule VF2, to the expression e1 valueof f at e0. By Strong Confluency we get the Church-Rosser property.

(2) one may apply the rule VF2 or the rule VF4, to the expression e valueof f at c_0. In this case, after the application of VF2 we can still apply VF4, because the memo information in ρ is not changed by VF2. By Strong Confluency we get the Church-Rosser property.

(3) one may apply the memopropagation rule (MPr) or any other rule, say R, to C[letmemo μ in e]. In this case we notice that there are no rules for transforming expressions of the form letmemo μ in e, except those which can be applied for transforming e.

Therefore if the memopropagation rule and rule R can both be applied to the expression C[letmemo μ in e], the latter rule should be applicable also to C[e]. Thus it can be applied to letmemo μ in C[e]. We have the following commutative diagrams:

(i) if R changes C[. . .] into C'[. . .]:

$$\begin{tikzcd} \rho \vdash C[\text{letmemo } \mu \text{ in } e] \arrow[r, "\text{(by R)}"] \arrow[d, "\text{(by MPr)}"] & C'[\text{letmemo } \mu \text{ in } e] \arrow[d, "\text{(by MPr)}"] \\ \rho \vdash \text{letmemo } \mu \text{ in } C[e] \arrow[r, "\text{(by R)}"] & \text{letmemo } \mu \text{ in } C'[e] \end{tikzcd}$$

and

(ii) if R changes e into e':

$$\begin{tikzcd}
\rho\vdash C[\underline{\text{letmemo}}\ \mu\ \underline{\text{in}}\ e] \arrow[r, "(\text{by } R)"] \arrow[d, "(\text{by } MPr)"] & C[\underline{\text{letmemo}}\ \mu\ \underline{\text{in}}\ e'] \arrow[d, "(\text{by } MPr)"] \\
\rho\vdash \underline{\text{letmemo}}\ \mu\ \underline{\text{in}}\ C[e] \arrow[r, "(\text{by } R)"] & \underline{\text{letmemo}}\ \mu\ \underline{\text{in}}\ C[e']
\end{tikzcd}$$

By Strong Confluency we get the Church-Rosser property.

Analogously one deals with the case when MU and any other rule can be applied.

Having proved that the Church-Rosser property holds for the Dynamic Semantics rules for the language L and memoL we can proceed towards the proof of the commutativity of the diagram of fig. 2-11 without caring about the possible nondeterminism.

2.4.4 Consistency of the operational semantics for the languages L and memoL

The commutativity of the diagram of fig. 2-11 holds for a class of programs which we will define here. They are programs expressed by elementary expressions which are expressions with no free variables and such that the right hand sides of their definitions have no free variables (except formal parameters or recursive function symbols). The commutativity holds also for another class of programs defined in the following subsection 2.4.5.

Let us start by giving some definitions.

Definition 2: e0 is an elementary expression, and we write $e0 \in$ elExp,
iff e0 $\in$ Exp of the language L and
$FV(e)=\emptyset$ for any x: τ=e definition occurring in e0
$FV(e) \subseteq \{x\}$ for any f(x: τ_0) : τ_1=e definition occurring in e0
$FV(e) \subseteq \{x, f\}$ for any rec f(x: τ_0) : τ_1=e definition occurring in e0, and no other kinds of definitions occur in e0. □

We also say that an abstract λz: τ_0. e: τ_1 is an elementary expression if e is an elementary expression. (z should not be considered while computing the free variables of the definitions in e.)
An environment $\rho \in Env_V$ (for the language L) is called an elementary environment (and we write $\rho \in elEnv_V$) iff $\forall x \in V$. $\rho(x) \in$ elExp.

An elementary expression context D[...] $\in$ elExp-contexts is an elementary expression with a "hole" s.t. when filling that "hole" with the expression e we have: D[e] $\in$ elExp. (The easy formal definition is left to the reader.)
For simplicity we will not write types in definitions.

We define a translation relation M: elExp (in L) $\Longrightarrow$ Exp (in memoL) which derives from the following elementary translations T1 (removing rec's) and T2 (introducing valueof at):

for any elExp-context D[. . .] and any occurrence of the expression f(e1) where f is free in D[f(e1)].

(T1) let rec f(x)=e in e1 $\Longrightarrow$ let f(x)=e in e1

(T2) let f(x)=e in D[f(e1)] $\Longrightarrow$ let f(x)=e in D[f(e1) valueof f at e1]

with the condition that all occurrences of rec should disappear in the translated expression (otherwise it would not be an element of Exp in memoL). We recall that in memoL all function definitions are recursive, though rec does not occur in them.

In defining the translation M, the translation T2 need not be applied to all subexpressions to which it is applicable.

We will often write M(e) meaning any expression e1 s.t. $(e \Longrightarrow e1) \in M$.

Given an environment $\rho \in elEnv_V$ for the language L we denote by N(ρ) the corresponding environment for the language memoL, where every abstract occurring in ρ is paired with an empty memo, and all occurrences of expressions of the form "let rec ρ0 in e" are replaced by "e". (This last modification corresponds to the replacement of the rule R2 by the rule R2' as suggested in Section 2.2).

For instance,

if ρ = {x=5, f=λx: int. let rec (f=λx: int. if x=0 then 1 else x·f(x−1): int) in if x=0 then 1 else x·f(x−1): int}

then N(ρ) = {x=5, f=<λx: int. if x=0 then 1 else x·f(x−1): int, $\emptyset$>}.

Rewritings in the language L will be denoted by "e1 (L)$\longrightarrow$ e2", while the ones in the language memoL by "e1 (memoL)$\longrightarrow$ e2".

We will say that an expression e is irreducible iff $e \in N+T$.

In the language L we will write $\rho \vdash_\alpha e1 \approx e2$ to assert that:

$\vdash_\alpha \langle \rho, e1 \rangle$ (L)$\longrightarrow^*$ $\langle \rho, t \rangle$ and t is irreducible iff

$\vdash_\alpha \langle \rho, e2 \rangle$ (L)$\longrightarrow^*$ $\langle \rho, t \rangle$ and t is irreducible.

Analogously in the language memoL for the rewriting (memoL)$\longrightarrow^*$.

In order to prove the commutativity of the diagram of fig. 2-11 let us start with some preliminary lemmas.
In what follows we will write $\rho \vdash e \dots$ instead of $\rho \vdash_{\alpha} e \dots$ and we assume that $FV(e) \subseteq dom(\rho)$. When we assume that $\rho \in Env_V$, we also implicitely assume that $\alpha \in TEnv_W$ with $\rho : \alpha$ and $V \subseteq W$.
Moreover, when we write <u>let</u> f(x)=e <u>in</u> E[f(e1)] (or similarly defined <u>let-expressions</u>), the occurrence of f in f(e1) is supposed to be free in E[...], so that the relevant binding for it is f(x)=e.

Lemma 3 shows the commutativity of the diagram 2-11 for a very simple class of expressions.

Lemma 3: $\forall \rho \in elEnv_V$. $\forall$ e0 $\in$ elExp (in L) $\cap$ Exp (in memoL) (i.e. no <u>rec</u>, <u>valueof at</u>, <u>letmemo</u> occur in e0),
$\rho \vdash$ e0 $(L) \longrightarrow^*$ t and t is irreducible iff
$N(\rho) \vdash$ e0 $(memoL) \longrightarrow^*$ t and t is irreducible.

Proof: Immediate by definition of elExp. In fact, since $V_0 = \phi$, the F1 rules for L and memoL turn out to bind f to the same abstract. □

The following lemma 4 tells us that the replacement of an occurrence of "f(e1)" by "f(e1) <u>valueof</u> f <u>at</u> e1" does not introduce any extra non terminating computation.

Lemma 4: $\forall \rho \in elEnv_V$.
$\forall$ D[<u>let</u> f(x)=e <u>in</u> E[f(e1)]] $\in$ elExp $\cap$ Exp (in memoL) we have that:
$\rho \vdash$ D[<u>let</u> f(x)=e <u>in</u> E[f(e1)]] $(memoL) \longrightarrow^*$ t1 and t1 is irreducible iff
$\rho \vdash$ D[<u>let</u> f(x)=e <u>in</u> E[f(e1) <u>valueof</u> f <u>at</u> e1]] $(memoL) \longrightarrow^*$ t2 and t2 is irreducible.

Proof: It is enough to notice that the propagation of a <u>letmemo</u> construct for the function f is bound to terminate, because in the outside let-expression there is a binding for f. □

The following basic Lemma 5 says that the memo information does not affect the result of the computation.

Lemma 5: $\forall \rho \in$ elEnv$_V$.

<u>if</u> $\rho \vdash$ <u>let</u> f(x) =e <u>in</u> E[f(e0)]

$\rightarrow^+$ <u>let</u> {f=λx. e, memo U {<c0, c1>}} <u>in</u> E[f(c0)] (1)

$\rightarrow$ <u>let</u> {f=λx. e, memo U {<c0, c1>}} <u>in</u> E[c1] (2)

(by memo lookup)

<u>then</u> $\rho \vdash$ <u>let</u> {f=λx. e, memo } <u>in</u> E[f(c0)] (3)

$\rightarrow^+$ <u>let</u> {f=λx. e, memo } <u>in</u> E'[c1] (4)

(by evaluating f(c0))

where E'[. . .] possibly differs from E[. . .] for updatings of memo components.

Proof: We will give the proof under the hypothesis that <u>valueof at</u> constructs are used only in expressions of the form: f(e) <u>valueof</u> f <u>at</u> e.

If <c0, c1>∈memo the lemma is obvious. Otherwise, it is the case that for deriving (1) we have:

$\rho \vdash$ <u>let</u> {f=λx. e, memo1} <u>in</u> D[f(c0) <u>valueof</u> f <u>at</u> c0] (0. 1)

$\rightarrow^+$ <u>let</u> {f=λx. e, memo2} <u>in</u> D'[c1 <u>valueof</u> f <u>at</u> c0] (0. 2)

$\rightarrow$ <u>let</u> {f=λx. e, memo2 U {<c0, c1>}} <u>in</u> D'[c1]

$\rightarrow^*$ (1)

where <c0, c1> $\notin$ memo1 U memo2 and D'[. . .] possibly differs from D[. . .] for updatings of memo components.

In order to derive (4) from (3) we make the same steps we made for deriving (0. 2) from (0. 1). Notice also that in (3) and (4) we used the same variable "memo" (not two different ones as in (0. 1) and (0. 2)), because by hypothesis in "memo" there exist all the argument-value pairs computed while evaluating f(c0) to c1. □

We can now prove a theorem which establishes the consistency of the operational semantics of the language memoL with respect to the one of the language L. (Previously we also called that property "agreement" between the two semantics.)

Theorem 6: $\forall \rho \in \text{elEnv}_V$. $\forall$ e0 $\in$ elExp.

$\rho \vdash$ e0 $(L)\longrightarrow^*$ t and t is irreducible iff

$N(\rho) \vdash M(e0)$ $(\text{memoL})\longrightarrow^*$ t and t is irreducible.

Proof: Any expression derived by translating a given expression e0 using M can be obtained by a sequence of elementary translations T1, removing rec's, followed by a sequence of elementary translations T2, introducing valueof at's.

In writing the relation symbols $(L)\longrightarrow$ and $(\text{memoL})\longrightarrow$ (and their reflexive and transitive closures) we will omit the qualifications (L) and (memoL) when they are easily understood from the context.

Part 1 Removing rec's.

Suppose M(e0)=e0. The thesis holds by Lemma 3.

Part 1.1

Suppose that M(e0) is obtained from e0 using only one elementary translation T1.

Let e0 be D[let rec f(x)=e in e1] and M(e0) be D[let f(x)=e in e1].

In L we have:

$\rho \vdash$ D[let rec f(x)=e in e1]

$\longrightarrow$ D[let rec (f=λx. let ϕ in e) in e1]

$\approx$ D[let rec (f=λx. e) in e1]

$\longrightarrow$ D[let (f=λx. let rec (f=λx. e) in e) in e1].

In memoL we have:

$N(\rho) \vdash$ D[let f(x)=e in e1]

$\longrightarrow$ D[let (f=λx. e) in e1],

where for simplicity we have not written the empty memo. We will do the same in what follows.

Now we will reason by induction on the number of the recursive calls of f we need to evaluate e1.

<u>Base case</u>. Obvious because there are no calls of the function f.

<u>Step case</u>. Suppose e1 = E[f(c)], and suppose that the occurrence of f in f(c) is free in e1.

In L: $\rho\vdash$ D[<u>let</u> {f=λx. <u>let</u> <u>rec</u> {f=λx. e} <u>in</u> e} <u>in</u> E[f(c)]]

$\longrightarrow^*$ D[<u>let</u> {f=λx. <u>let</u> <u>rec</u> {f=λx. e} <u>in</u> e} <u>in</u>

E[<u>let</u> x=c <u>in</u> <u>let</u> <u>rec</u> {f=λx. e} <u>in</u> e]].

In memoL: N(ρ) $\vdash$ D[<u>let</u> {f=λx. e} <u>in</u> E[f(c)]]

$\longrightarrow$ D[<u>let</u> {f=λx. e} <u>in</u> E[<u>let</u> x=c <u>in</u> e]]

$\approx$ D[<u>let</u> {f=λx. e} <u>in</u> E[<u>let</u> x=c <u>in</u> <u>let</u> {f=λx. e} <u>in</u> e]]

where $\approx$ holds because the free occurrences of f in e are bound to λx. e. (Recall that FV(e) $\subseteq$ {x, f}, because e0 $\in$ elExp.)

By induction hypothesis we have that for the expressions within the context E[. . .] the following holds:

$\forall\rho$ $\in$ elEnv$_V$. $\rho\vdash$ <u>let</u> x=c <u>in</u> <u>let</u> <u>rec</u> {f=λx. e} <u>in</u> e (L)$\longrightarrow^*$ t and t is irreducible iff

N(ρ) $\vdash$ <u>let</u> x=c <u>in</u> <u>let</u> {f=λx. e} <u>in</u> e (memoL)$\longrightarrow^*$ t and t is irreducible.

Since FV(<u>let</u> x=c <u>in</u> <u>let</u> <u>rec</u> {f=λx. e} <u>in</u> e) = FV(<u>let</u> x=c <u>in</u> <u>let</u> {f=λx. e} <u>in</u> e) = $\emptyset$, the context in which E[. . .] occurs does not matter, using the above conseguence of the induction hypothesis we get that:

$\forall\rho$ $\in$ elEnv$_V$.

$\rho\vdash$ e0 (L)$\longrightarrow^*$ D[<u>let</u> {f=λx. <u>let</u> <u>rec</u> {f=λx. e} <u>in</u> e} <u>in</u> E[t]] and t is irreducible iff

N(ρ) $\vdash$ M(e0) (memoL)$\longrightarrow^*$ D[<u>let</u> {f=λx. e} <u>in</u> E[t]] and t is irreducible.

By induction hypothesis (since the number of recursive calls of f in E[t] are fewer than the ones in E[f(c)]) and by Lemma 3, taking into account that the contexts D[. . .] and E[. . .] are not changed by the translation M, we get the thesis.

Notice that in the proof we did not care about the nondeterminism in the rewritings, because the Church-Rosser property holds for the dynamic semantics rules of the languages L and memoL.

Part 1.2

Suppose M(e0) is obtained from e0 by more than one elementary translation T1.

If for each instance of an elementary translation T1 there is a subexpression e_i of e0 such that it includes the translated expression and $FV(e_i)=\phi$, and all subexpressions $\{e_i\}$ are pairwise disjoint, the thesis can be proved by structural induction using the result of Part 1.1.

If the subexpressions $\{e_i\}$ are not pairwise disjoint, let us consider, without loss of generality, the case where e0 $\equiv$ C[let rec f(x) = D[let rec f(x)=e in e1] in e2] and M(e0) $\equiv$ C[let f(x)=D[let f(x)=e in e1] in e2].

By Part 1.1 we have that: for all $\rho \in elEnv_V$.

$\rho \vdash$ D[let rec f(x)=e in e1] (L)$\longrightarrow^*$ t and t is irreducible iff $N(\rho) \vdash$ D[let f(x)=e in e1] (memoL)$\longrightarrow^*$ t and t is irreducible.

Therefore, since the free occurrences of f in e2 are bound in e0 and M(e0) to expressions which will be evaluated in any context to the same value, we get the thesis.

Part 2 Introducing valueof's.

M(e0) is obtained from e0' by zero or more elementary translations T2 (introducing valueof at's), where e0' is obtained from e0 by elementary translations T1 (removing rec's) only.

We will make the proof by induction on the number of the elementary translations T2.

The induction hypothesis is as follows:

(a0). $\rho \vdash$ C[let f(x)=e in D[f(e1)]] (L)$\longrightarrow^*$ t and t is irreducible iff

(b0). $N(\rho) \vdash$ M(C)[let f(x)=e in M(D)[f(e1)]] (memoL)$\longrightarrow^*$ t and t is irreducible.

where from C[...] to M(C)[...] we made n translations T2 and maybe some translations T1 (and analogously for D[...])
As usual, we assume that the occurrence of f in f(e1) is free in D[f(e1)].
Let us assume that the (n+1)st translation T2, which transform f(e1) into f(e1) <u>valueof</u> f <u>at</u> e1, is made outside the r.h.s. of a definition. (The other case where the (n+1)st translation T2 occurs within the r.h.s. of a definition can be proved in an analogous way). We need to show that introducing that extra <u>valueof at</u> construct does not change the result of the computation, i.e.:
(b0). $N(\rho) \vdash$ M(C)[<u>let</u> f(x)=e <u>in</u> M(D)[f(e1)]] (memoL) $\longrightarrow^*$ t and t is irreducible iff
(b). $N(\rho) \vdash$ M(C)[<u>let</u> f(x)=e <u>in</u> M(D)[f(e1) <u>valueof</u> f <u>at</u> e1]] (memoL) $\longrightarrow^*$ t and t is irreducible.
Then we will have that the inductive step is valid: i.e. (a0) iff (b), and we get the thesis.
We will not care about the nondeterminism in the rewritings because the Church-Rosser property and Lemma 4 hold.

In memoL we have for the given expression in (b0):

$N(\rho) \vdash$ M(C)[<u>let</u> f(x)=e <u>in</u> M(D)[f(e1)]] (0)

$\longrightarrow$ M(C)[<u>let</u> (f=λx.e, $\emptyset$) <u>in</u> M(D)[f(e1)]] (1)

$\longrightarrow^*$ M(C)[<u>let</u> (f=λx.e, memo1) <u>in</u> M(E)[f(c0)]] (2)

$\longrightarrow^+$ M(C)[<u>let</u> (f=λx.e, memo2) <u>in</u> M(F)[c1]] (3)

For the given expression in (b) we have:

$N(\rho) \vdash$ M(C)[<u>let</u> f(x)=e <u>in</u> M(D)[f(e1) <u>valueof</u> f <u>at</u> e1]] (4)

$\longrightarrow^+$ M(C)[<u>let</u> (f=λx.e, memo1) <u>in</u> M(E)[f(c0) <u>valueof</u> f <u>at</u> e1]] (5)

$\longrightarrow^*$ M(C)[<u>let</u> (f=λx.e, memo1) <u>in</u> M(E)[f(c0) <u>valueof</u> f <u>at</u> c0]] (6)

$\longrightarrow^+$ M(C)[<u>let</u> (f=λx.e, memo2) <u>in</u> M(F)[c1 <u>valueof</u> f <u>at</u> c0]] (7)

The steps from (4) to (5) are the ones made from (0) to (2). The steps from (5) to (6) are analogous to those made from (1) to (2). (The context M(E)[...] and memo1 are not changed because all memoupdatings were done while deriving (5) from (4).)

We derive (7) from (6) in the same way as we derived (3) from (2).

Now there are two cases:

(i) $\langle c0, c1\rangle \in$ memo2.

(ii) $\langle c0, c1\rangle \notin$ memo2.

(No other cases occur, because f is a deterministic function, and therefore, if $\langle c0, c\rangle \in$memo2 then c=c1.)

<u>Case 1</u>. From (7) we get, using VF4:

$\rightarrow$ M(C)[<u>let</u> {f=λx. e, memo2} <u>in</u> M(F)[c1]] (8.1)

which is equal to (3).

Therefore, $N(\rho) \vdash (3) \approx (8.1)$.

<u>Case 2</u>. From (7) we get, using VF3 and MPr:

$\rightarrow^{+}$ M(C)[<u>let</u> {f=λx. e, {$\langle c0, c1\rangle$}Umemo2} <u>in</u> M(F)[c1]]. (8.2)

By computational induction on the number of the recursive calls of f, Lemma 4, Lemma 5 and Church-Rosser property we have:

$N(\rho) \vdash (3) \approx (8.2)$.

(Recall that $\rho \vdash$<u>let</u> ρ_0 <u>in</u> c $\rightarrow$ c if c$\in$Con, for any ρ_0.) □

The extension of the Theorem 6 to the case in which functions have more than one argument is immediate.

The following Theorem establishes the commutativity for the diagram of fig. 2-11 for closed expressions.

Theorem 7: $\forall$ closed expression e $\in$ elExp. $\emptyset \vdash$ e (L)$\rightarrow^{*}$ t and t is irreducible iff $\emptyset \vdash$ M(e) (memoL)$\rightarrow^{*}$ t and t is irreducible.

Proof: Corollary of the Theorem 6. □

2.4.5 Extending the results on the consistency of the semantics for the languages L and memoL

We will now extend the results presented in the subsection 2.4.4. Let us first make some remarks and give some examples which elucidate the difference between the semantics of the language L and the one of the language memoL.

Remark 1. The semantics of the language L is according to the "static binding" mechanism.
This is the binding method used in λ-calculus [Barendregt 81] and it is based on the textual occurrences of the variables before evaluation. The bindings for the language L can be determined by the following equivalence:

let x=e in e′ = (λx. e′) e.

(See also some examples below.)

Remark 2. The semantics of the language memoL is according to the "dynamic binding" mechanism.
It means that the binding of a variable f is delayed until f is used and it is made by using the innermost definition of f in whose scope that f occurrence exists at run time.

Example 23.

$\emptyset \vdash$ let y=3 in let f(x)=x+y in let y=2 in f(2) (L) $\longrightarrow^*$ 5,

because by the static binding y in x+y is bound to 3.

$\emptyset \vdash$ let y=3 in let f(x)=x+y in let y=2 in f(2) (memoL) $\longrightarrow^*$ 4,

because by the dynamic binding, when f is called the innermost binding for y gives it the value 2. □

Example 24.

$\emptyset \vdash$ let $f(x)=3$ in (let $f(x)=f(x)$ in $f(2)$) (L) $\longrightarrow^*$ 3.

$\emptyset \vdash$ let $f(x)=3$ in (let $f(x)=f(x)$ in $f(2)$) in memoL fails to have normal form.

Notice that let $f(x)=3$ in let rec $f(x)=f(x)$ in $f(2)$ in L fails to have normal form, while it is not an expression in memoL. □

Example 25.

$\emptyset \vdash$ let $g(x)=3$ in [let $f(y)=($let $g(x)=x+g(x)$ in $g(y))$ in $f(2)$] (L) $\longrightarrow^*$ 5.

because the given expression, using the static binding is equivalent to:

let $f(y)$ = (let $g(x)$ = $x+3$ in $g(y)$) in $f(2)$ and therefore to:

let $f(y)$ = $y+3$ in $f(2)$.

The given expression in memoL fails to have normal form because $f(2)$ is replaced, during the evaluation, by let $g(x)$ = $x+g(x)$ in $g(2)$ which, using the dynamic binding, has no normal form. □

Therefore, looking for an improvement of the results of the previous theorems 6 and 7 (see Section 2.4.4) while preserving the commutativity of the diagram in fig. 2-11, we have at least to satisfy the conditions under which the dynamic binding mechanism evaluates expressions in the same way as the static binding mechanism does.

We will be able to achieve only partial results because, in general, the problem of knowing whether an expression evaluates to the same value in a static binding regime and in a dynamic binding regime is unsolvable. This is due to the fact that the only way of checking that equivalence of regimes for expressions with procedures as parameters of procedures, is to evaluate the expressions themselves.

We can extend the results of the previous subsection as follows.

Let us give first the following definitions with their explanation.

Definition 8: We say that a set of variables is simple iff they are of ground type. (i.e. either bool or int for our language L.) □

Definition 9: e is an SDExpression (and we write $e \in$ SDExp)
iff $e \in$ Exp (for the language L) and

1. for each variable there is only one occurrence as a defined variable or as an argument variable, and
2. e is a closed expression (i.e. without free variables) and
3. any definition occurring in e is of one the following kinds only:
   - 3.1 $x{:}\,\tau$=e with FV(e) simple or
   - 3.2 $f(x{:}\,\tau_0){:}\,\tau_1$=e with FV(e) simple or
   - 3.3 rec $f(x{:}\,\tau_0){:}\,\tau_1$=e with FV(e)\{f} simple or
   - 3.4 d1 and d2 where d1 and d2 are of kind 3.1,....,3.4. □

Example 26.

1. let y: int=5 in let f(y: int) : int = 2+y in f(4) is not an SDExp because y is used both as a defined variable and as an argument variable.
2. let y: int=3 in let f(x: int) : int = x+y in f(2) is in SDExp and not in elExp.
3. let rec x: int=x+1 in x+5 is not an SDExpression.
4. let x: int=2 in let x: int=3 in x is in elExp but not in SDExp, because two bindings for x occur.
5. let y: int=3 in let rec f(x: int) : int = if x=0 then y else y•f(x-1) in f(y) is in SDExp, but not in elExp.

Expressions 2. and 5. show that SDExp is not included in elExp.

6. Here is an SDExpression for computing $0^2+1^2+...+7^2$ (for simplicity we do not write the types in definitions):

let rec sq(x) = if x=0 then 0 else sq(x-1)+2x-1
in let rec sumsq(y) = if y=0 then 0 else sq(y)+sumsq(y-1)
in sumsq(7). □

The basic idea of the above definition of SDExp is to avoid those expressions which may have different value when evaluated with the static binding or dynamic binding mechanism and for which the presence of rec's is significant. (The name SDExp suggests the invariance of the value w.r.t. the static or dynamic binding regime.)

For instance, we do not allow in SDExp the following expressions E1 and E2 (where "...f..." stands for any expression with a free occurrence of f, and the subscripts are used for distinguishing the different occurrences):

$\underline{let}\ f_{(1)}(x)=e\ \underline{in}\ (\underline{let}\ f_{(2)}(x)=\ldots f_{(3)}\ldots\ \underline{in}\ \ldots f_{(4)}\ldots)$ (E1)

$\underline{let}\ f_{(5)}(x)=(\underline{let}\ f_{(6)}(x)=\ldots f_{(7)}\ldots\underline{in}\ldots f_{(8)}\ldots)\ \underline{in}\ f_{(9)}\ldots$ (E2)

in which

(i) a variable occurs free in the scope of its own definition (see occurrence $f_{(3)}$ in E1, and $f_{(7)}$ in E2) and

(ii) it is used in expressions to be evaluated (see occurrence $f_{(4)}$ in E1 and $f_{(8)}$ and $f_{(9)}$ in E2) and

(iii) it is "enclosed" in the scope of different bindings for it (see bindings $f_{(1)}$ and $f_{(2)}$ for $f_{(3)}$ in E1, and bindings $f_{(5)}$ and $f_{(6)}$ for $f_{(8)}$ in E2).

For the above expressions the presence of rec is significant. Indeed rec in front of $f_{(2)}(x)$ in E1 would make the binding of $f_{(3)}$ in E1 to be $f_{(2)}$ itself and not $f_{(1)}$ (analogously for rec in front of $f_{(6)}$).

Let M(e), for any $e \in Exp$ (in L), denote the expression e after erasing all rec occurrences and replacing some occurrences of "f(e1)" by "f(e1) valueof f at e1".

Theorem 10: $\forall \rho \in Env_V.\ \forall e \in Exp$ s.t. $\underline{let}\ \rho\ \underline{in}\ e \in SDExp$,

$\rho \vdash e\ (L) \longrightarrow^{*} t$ and t is irreducible iff

$N(\rho) \vdash M(e)\ (memoL) \longrightarrow^{*} t$ and t is irreducible.

Proof: Similar to the proof of Theorem 6.

We do not present it here because it is a bit lenghty. The proof is based on the fact that free variables in SDExp are simple. They can only be bound once and they cannot interfere with the argument variables. □

The following Theorem gives us the main result, i.e. the diagram of fig. 2-11 commutes for programs denoted by expressions in SDExp.

Theorem 11: $\forall e \in$ SDExp $\quad \phi \vdash e \; (L) \longrightarrow^{*} t$ and t is irreducible iff $\phi \vdash M(e) \; (\text{memoL}) \longrightarrow^{*} t$ and t is irreducible.

Proof: Corollary of Theorem 10. $\square$

2.4.6 Main Results and Some Final Examples

We have proved (see Theorems 7 and 11) that the commutativity of the diagram 2-11 holds for:

- M being a translation which erases all occurrences of rec and it introduces "f(e) valueof f at e" expressions instead of some occurrences of "f(e)" where f is any recursively defined function;
- C(L) being a class of programs described by Elementary Expressions (see Definition 2) or by SDExpressions (see Definition 9).

From those results we proved the correctness of the program annotation methodology when we perform the translation M on programs denoted by Elementary Expressions or SDExpressions, and the evaluator of the annotated programs implements the rules of the operational semantics of memoL as given in Section 2.3.

One can extend those results by allowing programs to be made of expressions whose closed subexpressions are either Elementary Expression or SDExpressions.

For instance, by the above results, the following annotated program, using the M translation, computes the binomial coefficients:

```
bin(n,m) = if m=0 or n=m then 1
                 else(bin(n-1,m-1) valueof bin at (<n-1,m-1> and <n-1,n-m>)) +
                     (bin(n-1,m) valueof bin at (<n-1,m> and <n-1,n-m-1>))
```

and avoids all redundant function evaluations.

We recall that redundant calls in this case cannot be avoided "at compile time" by using Cohen's methods nor the tupling strategy. (In [Cohen 83] pages 293–294 it is given a "run time" method by which the derived program depends on the values of n and m.) □

Another familiar example is the Fibonacci program. By the translation M we can obtain the following expression:

```
let mfib(n) = if n≤1 then 1 else  mfib(n-1) valueof mfib at n-1
                                 + mfib(n-2) valueof mfib at n-2
in mfib(m).
```

It avoids redundant recursive calls and it achieves the same linear time performances of the program given in [Burstall and Darlington 77] page 49, which was derived by transformation at the expense of a "eureka step". □

The following annotated program computes $0^2+1^2+...+n^2$, in the case where multiplications by 2 only are allowed. Using the <u>valueof at</u> construct the repeated evaluation of recursive calls is avoided, and a linear running time algorithm is obtained.

```
let sq(x) = if x=0 then 0 else sq(x-1)+2x-1
in let sumsq(y) = if y=0 then 0 else [sq(y) valueof f at y] + sumsq(y-1)
in sumsq(n).
```

The same linear time performance is achieved by the following program, written in the recursive equations style:

```
┌ sumsq(n) = π1(t(n))
│ t(0)     = <0,0>
│ t(n+1)   = <sq(n+1), sumsq(n+1)>
│          = <sq(n)+2n+1, sq(n)+2n+1+sumsq(n)>
│          = <a+v, a+v+b> where <a,b>=t(n)
└                               v=2n+1
```

which can be obtained by using the tupling strategy. Indeed one can easily see that the pair <sq(n), sumsq(n)> determines a cut in the m-dag for the function sumsq. □

Notice that the memo approach for saving redundant evaluations we have presented, works for first order functions. An extension to the case in which one deals with higher-order functions, is possible, provided that the values of the arguments range over domains for which the equality predicate is decidable. In that case one can effectively test whether or not an entry of the memo-table is relevant for the function call to be evaluated.
We leave that extension to elsewhere.

2.4.7 On the position of the memofunctions definitions

Before closing this section we will now make a few remarks on where one should place the memofunction definitions.

The memo component of the environments avoids redundant computations of recursive calls when those calls are depending on a unique "father call" with which the memo is associated (see in section 2.4.6 the example of the binomial coefficients program).
Avoidance of redundant computations is also achieved when the memo is associated to a definition whose scope includes recursive calls, as the following example shows.

Example 27. In the expression:

let f(x) = if p(x) then a(x) else b(x, f(cx), f(dx)) in t(f(rx), f(sx))

if we change "f(cx)" into "f(cx) valueof f at cx" (and analogously for f(dx)) repeated evaluations of the function f are avoided not only inside each of the calls f(rx) and f(sx), but also outside them.

For instance, given let f(x)=x+1 valueof f at x in plus(f(2), f(2)) we get 6, by evaluating only once f(2) according to the definition "f(x)=x+1". For the second call of f(2) we use the memo information. □

The "communication" outside the scope of the "father-call" shown in the above example could not have been achieved via the methods indicated in [Cohen 83]. Moreover, having the memo component associated with the definition of the functional variable to which it refers, has the advantage of being able to discard, during computation, the relevant memo information when no longer needed.

2.5 Formal Theories for Structural Operational Semantics

In this section we will see how one can interpret the structural operational semantics rules in formal theories and we will present two different approaches. They allow us to translate in two different ways the operational semantics rules into a deductive system à la Prolog [Roussel 75, Warren 77]. The language we will consider here is a very simple one. It is made out of expressions only. They are defined as follows:

$e ::= 0 \mid S(e) \mid e1+e2 \qquad e \in Exp$

We leave to the reader the extension of our results to a more realistic programming language. We give the operational semantics by the following axioms and rules:

S1. $e+0 \longrightarrow e$

S2. $e1+S(e2) \longrightarrow S(e1+e2)$

S3. $$\frac{e \longrightarrow e'}{S(e) \longrightarrow S(e')}$$

S4. $$\frac{e1 \longrightarrow e1'}{e1+e2 \longrightarrow e1'+e2}$$ S5. $$\frac{e2 \longrightarrow e2'}{e1+e2 \longrightarrow e1+e2'}$$

We look for a theory T and an interpretation function t: $Exp \times Exp \longrightarrow$ Formulas s.t. $\forall e, e' \in Exp$. $R(e,e')$ iff $T \vdash t(e,e')$ where $R(e,e')$ is the relation $e \longrightarrow e'$ (or its reflexive transitive closure $\longrightarrow^*$) as introduced in the operational semantics. We will give two solutions to that problem.

Our first solution is for the case:

$R(e,e') \equiv e \longrightarrow e'$ and $t(e,e') \equiv e \longrightarrow e'$.

Our second solution is for the case:

$R(e,e') \equiv e \longrightarrow^* e'$ and $t(e,e') \equiv F(e) \supset F(e')$ where F is a unary relation symbol and $\supset$ is the usual implication for logical theories (see Appendix C).

2.5.1 First Interpretation

We consider a first order language L as a quadruple (S, V, O, R), where:

$S = \{\neg, \supset, \forall, =\}$ is a set of logical connectives.

$V = \{x, y, z, \dots\}$ is a countable set of variables.

$O = \{op1, op2, \dots\}$ is a set of operators with arity.

$R = \{r1, r2, \dots\}$ is a set of relation symbols with arity.

Parentheses are used as auxiliary symbols.

For the notions which are not defined here, the reader may refer to [Shoenfield 67, Monk 76].

We choose: $O = \{0_0, S_1, +_2\}$, $V = \{\}$ and $R = \{\longrightarrow_2\}$, where the subscripts denote the arity.

Terms are built from the operators in O as usual. We have: Terms = Exp.

Atomic formulas are of the form $e1 \longrightarrow e2$, where $e1, e2 \in$ Exp.

The equality symbol $=$, and the universal quantifier symbol $\forall$ will not be used.

Formulas are defined as the smallest set containing atomic formulas and closed w.r.t. $\neg$ and $\supset$. Closure w.r.t. $\forall v$ for any $v \in V$, is not necessary here, because $V = \{\}$.

As logical rules we consider, as usual, the modus ponens rule MP:

$$\frac{A, \quad A \supset B}{B}.$$

We do not need the generalization rule G:

$$\frac{A}{\forall v_i\ A} \quad \text{for } v_i \in V \quad \text{because } V=\{\}.$$

We can dispense with nonlogical rules by adopting the following nonlogical axioms S′ [Shoenfield 67]:

S1′. $e+0 \longrightarrow e$.

S2′. $e1+S(e2) \longrightarrow S(e1+e2)$.

S3'. $(e \longrightarrow e') \supset (S(e) \longrightarrow S(e'))$.

S4'. $(e1 \longrightarrow e1') \supset (e1+e2 \longrightarrow e1'+e2)$.

S5'. $(e2 \longrightarrow e2') \supset (e1+e2 \longrightarrow e1+e2')$.

We can easily build a term model for this theory.
We will use the standard notion of models for first order theories [Monk 76].
In particular, if U is a model and ϕ is a formula $U \models \phi$ means that ϕ is true in U. If Γ is a set of formulas, $U \models \Gamma$ means that $\forall \phi \in \Gamma.\ U \models \phi$. If K is a class of models $K \models \Gamma$ means that $\forall\ U \in K.\quad U \models \Gamma$.
Let us consider the structure U = (Terms, f, R), where f is such that f_0 is the term 0, f_S: Terms$\longrightarrow$Terms is s.t. $f_S(t)=S(t)$ and f_+: Terms$^2\longrightarrow$Terms is s.t. $f_+(t1,t2) = t1+t2$.
R associates with the binary relation symbol $\longrightarrow$ the binary relation $\Longrightarrow \subseteq$ Terms$\times$Terms defined as follows:
t1 $\Longrightarrow$ t2 iff t1 $\longrightarrow$ t2 can be derived from the set of axioms and rules S={S1, . . . , S5}.
As usual the satisfiability relation $M \models \phi$ for a formula ϕ in the structure M is defined, by induction on the formulas as follows:

$M \models t1 \longrightarrow t2$	iff $t1 \Longrightarrow t2$	for any term t1 and t2.
$M \models \neg\phi$	iff not$(M \models \phi)$	for any formula ϕ
$M \models \phi \supset \psi$	iff either not$(M \models \phi)$ or $(M \models \psi)$	for any formula ϕ and ψ.

As for any first order theory the completeness theorem holds and we have:
$S' \vdash \phi$ iff $S' \models \phi$.

Theorem 12: $e \longrightarrow e'$ using S1, . . . , S5 iff $S' \vdash e \longrightarrow e'$.

Proof: (only if part) By induction of the length of the derivation of $e \longrightarrow e'$ from the axioms and rules S1, . . . , S5.
(if part) $S' \vdash e \longrightarrow e'$ iff $S' \models e \longrightarrow e'$. Since $U \models S'$ (as one can easily verify), $U \models e \longrightarrow e'$. By definition of U we have $e \longrightarrow e'$. □

The logical theory we have presented allows a direct implementation of the operational semantics definitions using a first order deduction system. In particular one can use PROLOG, and in the next section we will indeed present such approach.

The Second Interpretation of the operational semantics rules into formal theories is presented in Appendix C. We confined it there because it is a study more related to the formalism we are using, i.e. the generalized deductive definitions, rather than the operational semantics of memofunctions.

2.6 Prolog implementations of Structural Operational Semantics

We implemented the structural operational semantics definitions both for the language L (see Section 2.1) and the language memoL (see Section 2.3) using the DEC-10 Prolog [Byrd et al. 80, Clocksin and Mellish 81].
Let us first consider the implementation of the semantics for the language L.

2.6.1 Implementation Syntax for L

Expressions	e $\in$ Expressions
e ::= 0 | 1 |. . .	Natural Numbers
| true | false	Boolean Values
| v(x) | . . . | v(f) . . .	Individual and Functional Variables
| succ | plus(e1, e2) | . . .	Successor and Arithmetic Functions
| eq(e1, e2) | if(e0, e1, e2)	Equality Predicate and Conditional
| let(d, e)	Let-expression
| fapp(v(f), e)	Function Application.

Types	typ, typ0, typ1 $\in$ Types
typ ::= int | bool	integer and boolean types
| typ0 →$ typ1	functional types

Definitions	df $\in$ Definitions
df ::= def(v(x), typ, exp)	simple variable definition
| deff(v(f), typ, v(x), exp)	function variable definition
| rec(df)	recursive definition
| andd(df1, df2)	simultaneous definition
| ρ	environments

Environments	ρ, $\rho 1$ $\in$ Environments
ρ ::= nil_env	empty environment
| env(bind(v(x), val), ρ1)	generic environment

Values	val $\in$ Values
val ::= 0 | 1 | ...	Natural Numbers
| true | false	Boolean Values
| lambda(v(x), typ, e)	Lambda Expressions

Notes

i) Constants are either Natural Numbers or Boolean Values.

ii) Terminal Expressions are either Natural Numbers or Boolean Values. or Variables.

iii) The only Terminal Definitions are the Environments.

iv) Only first-order functions are allowed.

Auxiliary definitions for the Dynamic Semantics of L

1. Operators

$\longrightarrow$	rewriting of configurations
$\longrightarrow^*$	reflexive transitive closure of $\longrightarrow$
$\longrightarrow\$$	functional types
:	association of environments and transitions, e.g. Ro : t1 $\longrightarrow$ t2

All operators are left associative.

2. Predefined Predicate and Functions

Goal:	Success if:
is_const(X)	X is a constant.
terminal(X)	X is a terminal expression or a terminal definition.
free_var(X, Y)	X is an expression or a definition and Y is the list of its free variables.

defined_var(X, Y)	as for free_var. Y is the list of defined variables in X.

3. Set Manipulation

Sets are represented as lists.

Goal	Success if:
setunion(X, Y, Z)	$Z = X \cup Y$
member(X, Y)	$X \in Y$
setminus(X, Y, Z)	$Z = X - Y$
setinters(X, Y, Z)	$Z = X \cap Y$

4. Environment Manipulation

update_env : Environments $\times$ Environments $\times$ Environments $\longrightarrow$ goal

avoid_set : Environments $\times$ Set(of Variables) $\times$ Environments $\longrightarrow$ goal

restrict_to_set : Environments $\times$ Set(of Variables) $\times$ Environments $\longrightarrow$ goal

apply_env : Environments $\times$ Variables $\times$ Values $\longrightarrow$ goal

recur_env : Environments $\times$ Environments $\times$ Environments $\longrightarrow$ goal

Goal:	Success if:
update_env(Ro0, Ro1, FinRo)	FinRo = Ro0[Ro1] (where $\cdot[\cdot]$ denotes the usual function updating so that Ro1 supersedes Ro0).
avoid_set(Ro, X, LeftRo)	LeftRo is s.t. dom(LeftRo) = dom(Ro)−X and $\forall$x $\in$ dom(LeftRo). LeftRo(x)=Ro(x).
restrict_to_set(Ro, X, ReRo)	ReRo is s.t. dom(ReRo)=dom(Ro) $\cap$ X and $\forall$x $\in$ dom(ReRo). ReRo(x)=Ro(x)
apply_env(Ro, X, Val)	X is bound to Val in the environment Ro.
recur_env(Ro, Ro, RecRo)	$\rho \vdash$ rec Ro$\longrightarrow$RecRo (see rule R2).

2.6.2 Dynamic Semantics for L

In the implementation Ro: A $\longrightarrow$ B denotes $\rho \vdash$ A $\longrightarrow$ B. For instance, we will write Ro: let(R,C) $\longrightarrow$ C instead of: $\rho \vdash$ <u>let</u> R <u>in</u> C $\longrightarrow$ C.

In the implementation $\longrightarrow^*$ denotes the "terminal reflexive transitive closure" of $\longrightarrow$ in the sense that Ro: E $\longrightarrow^*$ Efin holds iff:

1) $\exists$ $E_0, E_1, \ldots, E_n$ for $n \geq 0$ and Ro: $E_0 \longrightarrow E_1, \ldots,$ Ro: $E_{n-1} \longrightarrow E_n$ and E_0=E, E_n=Efin hold and

2) Efin is a terminal expression (or a terminal definition).

Lines starting with % are comment lines.

A:-B,C,...F . means "if B and C and ... F succeed then A succeeds".

A:-. means that A succeeds.

More details on the PROLOG syntax and the PROLOG system we used may be found in [Clocksin and Mellish 81]. In particular the reader may refer to that book for the explanation of the semantics of the goal "!" which controls the backtracking mechanism. For understanding its use the following may suffice: A:-B,C,....,F,! means that <u>if</u> B <u>and</u> C <u>and</u>...<u>and</u> F succeed <u>then</u> A succeeds and it is not possible to have a success for A in any other way (even if other clauses of the form A:-. or A:- .... exist).

%<u>Evaluation to terminal (or normal) form</u>.

Ro: E $\longrightarrow^*$ E :- terminal(E), !.

Ro: E $\longrightarrow^*$ E2 :- Ro: E $\longrightarrow$ E1, Ro: E1 $\longrightarrow^*$ E2.

%<u>Variables</u>.

Ro: v(X) $\longrightarrow$ V :- apply_env(Ro,X,V), !.

%<u>Arithmetics</u>.

Ro: succ(N) $\longrightarrow$ M :- integer(N), M is N + 1, !.

Ro: succ(E) $\longrightarrow$ succ(E1) Ro: E $\longrightarrow$ E1.

Ro: plus(M,N) $\longrightarrow$ S :- integer(M), integer(N), S is M + N, !.

Ro: plus(E1,E2) $\longrightarrow$ plus(E11,E2) :- Ro: E1 $\longrightarrow$ E11.

```
Ro: plus(E1,E2) --> plus(E1,E21) :- Ro: E2 --> E21.
(Analogously for other arithmetic functions)
Ro: eq(M,N) --> true              :- integer(M),integer(N), M = N,!.
Ro: eq(M,N) --> false             :- integer(M),integer(N), not(M = N),!.
Ro: eq(E1,E2) --> eq(E11,E2)      :- Ro: E1 --> E11.
Ro: eq(E1,E2) --> eq(E1,E21)      :- Ro: E2 --> E21.
%Conditional.
Ro: if(E0,E1,E2) --> E1                :- Ro: E0 --> true,!.
Ro: if(E0,E1,E2) --> E2                :- Ro: E0 --> false,!.
Ro: if(E0,E1,E2) --> if(E01,E1,E2)     :- Ro: E0 --> E01, !.
%Let clause.
Ro: let(nil_env,E) --> E          :- !. % inserted for efficiency.
Ro: let(D0,E) --> let(D1,E)       :- Ro: D0 --> D1,!.
Ro: let(D,E) --> let(D,E1)        :- terminal(D), update_env(Ro,D,Ro1),
                                     Ro1: E --> E1, !.
Ro: let(D,E) --> E                :- terminal(E),!.
%Function application.
Ro: fapp(v(F),E) --> fapp(v(F),E1) :- Ro: E --> E1,!.
Ro: fapp(v(F),C) --> let(env(bind(v(X),C),nil_env),Body)
                          :- is_const(C),
                             apply_env(Ro,F,lambda(v(X),_,Body)),!.
%Simple definitions.
Ro: def(v(X),T,E) --> def(v(X),T,E1)                :- Ro: E --> E1.
Ro: def(v(X),T,E) --> env(bind(v(X),E),nil_env)     :- is_const(E),!.
%Function definitions.
Ro: deff(v(F),T,v(X),E) --> env(bind(v(F), lambda(v(X),T,let(d,E))),
                                   nil_env)
                               :- free_var(E,FVE), setminus(FVE,[X],V),
                                  restrict_to_set(Ro,V,D).
```

```
%Recursive functions.
Ro: rec(D) --> rec(D1):- defined_var(D,DVD), free_var(D,FVD),
                         setinters(DVD,FVD,V0), avoid_set(Ro,V0,Ro1),
                         Ro1: D --> D1, !.
Ro: rec(D) --> Ro1      :- terminal(D), recur_env(D,D,Ro1).
%And definitions.
Ro: andd(D0,D1) --> Ro1             :- terminal(D0),terminal(D1),
                                       update_env(D0,D1,Ro1),!.
Ro: andd(D0,D1) --> andd(D01,D1)    :- Ro: D0 --> D01.
Ro: andd(D0,D1) --> andd(D0,D11)    :- Ro: D1 --> D11.
%Failure.
Ro: E -->* E1:- nl, write(' *** Evaluation fails for '),
                print(E), write(' in the environment:'),
                print(Ro), fail.
%Evaluation in a given environment.
eeval(Ro,E)  :- Ro: E -->* Enf, nl, write('In the environment: '),
                print(Ro), write(' the expression <or definition>: '),
                nl,print(E), write(' reduces to: '), print(Enf).
%Evaluation.
eval(E)      :- eeval(nil_env,E).
```

The !'s make the system more space efficient. The failure clause is for debugging purposes.

2.6.3 Pragmatics for L

Here is an example session for the language L using the DEC-10 Prolog system at Edinburgh University. The user should have access to the files *.pro[3120,3134].

```
.run prolog
```

```
Edinburgh DEC-10 Prolog
    . . .
| ?- ['sys00.pro'].
    . . .
 yes
| ?- eval(let(def(v(x),int,2),plus(minus(v(x),1),4))).
 In the environment:  { }  the expression <or definition>:
 let x:int==2 in x-1+4    reduces to:      5
 yes
| ?-                  and the system waits for another query.
```

There is a tracing facility to show the "surface transitions". It can be switched on by typing: step. and it can be switched off by: no_step.

```
| ?- step.
 yes
| ?- eval(let(def(v(x),int,2),plus(minus(v(x),1),4))).
 let x:int==2 in x-1+4   -->  let {x==2} in x-1+4  -->
 let {x==2} in 2-1+4     -->  let {x==2} in 1+4  -->
 let {x==2} in 5         -->  5
 yes
| ?-
```

Notice the "pretty print" let x:int==2 in x-1+4 instead of let(def(v(x),int,2),plus(minus(v(x),1),4)). We will not describe here the details of the pretty print syntax.

Using the backtracking facility of PROLOG we can obtain from a given expression (or definition) all the rewriting sequences which are possible. Backtracking is activated while looking for a new binding of a given variable. Let us consider the following example. Suppose we have the goal:
go(X) :- eval(plus(minus(2,1),plus(3,1))), where the variable X occurs.

Then if we ask: go(X). we get the answer:

$2-1+3+1 \rightarrow 1+3+1 \rightarrow 1+4 \rightarrow 5$ X = . . .

and by typing ";" and RETURN we can get another reduction sequence for the given expression. We can continue that process and we can get all the reduction sequences. A general formula for computing their number will be given in subsection 2.6.7.

In our case we have the two sequences: the one given above and

$(2-1)+(3+1) \rightarrow (2-1)+4 \rightarrow 1+4 \rightarrow 5$.

Since the failure clause interacts with the backtracking mechanism, in order to observe the various reduction sequences we provide the user with a system called "syspar.pro" (which should be used instead of "sys00.pro".)

Let us turn now to the implementation of operational semantics for the language memoL. Since it is very similar to the language L, we will mention only the features which are different.

2.6.4 Implementation Syntax for memoL

Expressions $\qquad$ e ∈ Expressions

e ::= . . . (as for L)

| vof(e0, v(f), e1) $\qquad$ valueof expression

| letm(memel, e) $\qquad$ letmemo expression

Definitions

(as for the language L without recursive definitions)

Constants $\qquad$ con ∈ Constants

con ::= 0 | 1 | . . . | true | false

Environments $\qquad$ ρ ∈ Environments

ρ : : = nil_env	empty_environment
| env(bind(v(x), val), ρ)	binding a simple variable
| env(bind(v(f), ie(val, μ)), ρ)	binding a function variable

Memos	$\mu \in$ Memos
μ: : = nil_mem	empty-memo
| mem(pair(con0, con1), μ)	memo with <con0, con1>

Memoelements	memel $\in$ Memoelements
memel : : = mel(v(f), con0, con1)	remembering f(con0)=con1

In the implementation we did not use MemoEnvironments and we used instead Memoelements. The relationship among the two notions is that a memoelement is a memoenvironment where there is only one function symbol and it is bound to an argument-value pair only. The possibility of using in our implementation Memoelements instead of MemoEnvironments is due to the fact that we give priority to the application of the "memopropagation" and "memoupdating" rules. In that case, in fact, only one memoelement at a time is propagated towards the environment where it should be stored and therefore in any "<u>letmemo</u> μ <u>in</u> e" expression we have:
$\mu \equiv \{f(c_0)=c_1\}$ for some f $\in$ V, c_0 and c_1 $\in$ Con.

<u>Manipulation of Memos</u>.

memo_update : Environments × Memoelements × Environments $\longrightarrow$ goal

in_memo : Environments × Variables × Constants × Constants $\longrightarrow$ goal

memo_update(Ro0, mel(v(F), C0, C1), RoU) : this goal succeeds if RoU is an environment where the memo for the function F has been updated with the pair <C0, C1>.

in_memo(Ro,F,C0,C1): this goal succeeds if in the environment Ro, the memo for the variable F has the pair <C0,C1>.

2.6.5 Dynamic Semantics for memoL

The Dynamic Semantics for the language memoL is very similar to the one for the language L. The differences are the following ones:

```
%Memo updating and Memo propagation. Letmemo expressions.
% (Initial clauses of the program).
Ro: E --> E1 :- propmemo,
                E =.. [ let, Env, letm(mel(v(F),C0,C1),Exp)],
                Env = env(A,B), defined_var(Env,DVE), member(F,DVE),
                memo_update(Env, mel(v(F),C0,C1), NewEnv),
                E1 =.. [ let, NewEnv, Exp], retract((propmemo)), !.
Ro: E --> E1  :- propmemo,
                E =.. [ Fu | Argl], memberpos(letm(M,Exp),Argl,P),
                repl(P,Argl,Exp, NewArgl), NewExp =.. [ Fu | NewArgl],
                E1 =.. [ letm, M, NewExp] , !.
%Function application.
Ro: fapp(v(F),E) --> fapp(v(F),E1)          :- Ro: E --> E1,!.
Ro: fapp(v(F),C0) --> C1                    :- is_const(C0),
                                               in_memo(Ro,F,C0,C1),!.
Ro: fapp(v(F),C0) --> let(env(bind(v(X),C0),nil_env),Body)  :-
                      is_const(C0),
                      not(in_memo(Ro,F,C0,_)),
                      apply_env(Ro,F,ie(lambda(v(X),_,Body),_)), !.
%valueof at .
Ro: vof(E1,v(F),E0) --> vof(E1,v(F),E01)  :- Ro: E0 --> E01.
Ro: vof(E1,v(F),E0) --> vof(E11,v(F),E0)  :- Ro: E1 --> E11.
Ro: vof(C1,v(F),C0) --> letm(mel(v(F),C0,C1)
                              :- is_const(c0), is_const(C1),
                              not(in_memo(Ro,F,C0,C1)),asserta((propmemo)),!.
Ro: vof(E,v(F),C0) → C1      :- is_const(C0), in_memo(Ro,F,C0,C1),!.
```

%<u>Function definitions</u>.

```
Ro: deff(v(F),T,v(X),E) → env(bind(v(F), ie(lambda(v(X),T,E),
                                          nil_mem)),nil_env)  :- !.
```

<u>Remarks</u>.

1) <u>not(X)</u> is a goal which succeeds if the goal X fails, and viceversa.

2) When an expression of the form <u>letmemo</u> {f(c0)=c1} <u>in</u> c1 is generated (by rule VF3) we will propagate the information "f(c0)=c1" by inserting it in the memopart of the environment where f is bound. This is done by asserting the goal "propmemo" via asserta((propmemo)) and invoking the memopropagation clauses as often as possible, by locating them at the beginning of the program. (Indeed Prolog gives priority to the rules one writes first.)

3) We will not explain in detail here the clauses for memopropagation and memoupdating. They implement the rules MU and MPr, without introducing the notion of atomic expression-contexts. We did so for reasons of efficiency. We analyzed and manipulated the structure of the expression where the memopropagation and memoupdating should take place by using the PROLOG built-in predicate =.. [Clocksin and Mellish 81]. It provides a way of "visiting a structure" by giving its "functor" and its "arguments". For instance, +(2,3) =.. [+,2,3] and letm(memel,e) =.. [letm,memel,e].

Two functions are used:

<u>memberpos</u>: expressions $\times$ list of expressions $\times$ integer $\rightarrow$ goal.

<u>repl</u> : integer $\times$ list of expressions $\times$ expression $\times$ list of expressions $\rightarrow$ goal.

memberpos(ei,[e1,....,en],k) succeeds if k=i, i.e. the expression ei is the i-th expression in the given list.

repl(k,[e1,....,ek,....,en],ek1,E) succeeds if E = [e1,....,ek1,....,en], where the k-th element is replaced by ek1.

4) A list of the form [e1, e2, . . . , en] is also represented as [e1|A] where A = [e2, . . . , en].

5) When function definitions are evaluated we do not build closures, but we bind the functions to their corresponding lambda expressions. That is correct because for $f(x{:}\ \tau_0) : \tau_1 = e$ definitions we assume that: $FV(e) \subseteq \{x, f\}$. Indeed correctness holds under weaker hypotheses on FV(e) (see Sect. 2.4).

2.6.6 Pragmatics for memoL

In order to evaluate expressions or definitions in the language memoL the user of the DEC-10 Prolog system should have access to the files *.pr1[3120,3134] and type:

['sys01.pr1']. The system answers as follows:

```
  . . .
  sys01.pr1
| ?-
```

then one could ask, for instance, for the evaluation of the fibonacci function using memo information by typing: mfib(4). and one gets:

In the environment { } the expression <or definition>:

<u>let</u> mfib(x)==<u>if</u> x=0 <u>then</u> 1 <u>else</u> <u>if</u> x=1 <u>then</u> 1 <u>else</u>

[mfib(x-1) <u>valueof</u> mfib <u>at</u> x-1] + [mfib(x-2) <u>valueof</u> mfib <u>at</u> x-2]

<u>in</u> mfib(4) reduces to: 5.

In our Prolog implementation fibonacci(5) is computed in about 14 seconds using the L version and about 9 seconds using the memoL version. Indeed the direct implementation of the operational semantics rules make the computations to be very slow. That is due to the fact that structural rules often force "unnecessary" deduction steps (as shown in Example 18 of the previous section 2.1) and the fact that, in general, Prolog works as a blind theorem prover. Despite the heavy inefficiency problems our implementations were of great help in checking the correctness of our semantics rules.

2.6.7 On the number of different reduction sequences

As we mentioned in the previous section, our Prolog implementation of the operational semantics rules provides also a way of computing all reduction sequences of a given expression to be evaluated. In this section we give a formula for counting their number. Through this analysis we can measure the "parallelism" inherent in the evaluation of an expression and we also have an indirect proof of the correctness of our Prolog implementation.

Let us first consider the simple case of expressions with binary operators only. This simple case allows us to study the reduction sequences of arithmetic expressions, like for instance, (5+3)-(4•1). This expression can be reduced in two different ways:

1. $(5+3)-(4\cdot 1) \rightarrow 8-(4\cdot 1) \rightarrow 8-4 \rightarrow 4$ and
2. $(5+3)-(4\cdot 1) \rightarrow (5+3)-4 \rightarrow 8-4 \rightarrow 4$.

Let us define the following rewriting systems on Terms. $t \in$ Terms.

t is either c or | t1 t2 where t1, t2 $\in$ Terms.

The only rewriting rule is: | c c $\rightarrow$ c.

The stroke operator | (analogous to the one introduced by [Rosenbloom 50]), stands for any binary operator (as a prefix).

Example 28. The term |||cc|cc|cc can be reduced in 8 ways.

$$\begin{tikzcd}
 & ||c|cc|cc \arrow[r] \arrow[dr] & |cc|cc \arrow[r] \arrow[dr] & |c|cc \arrow[dr] & & \\
|||cc|cc|cc \arrow[ur] \arrow[r] \arrow[dr] & |||ccc|cc \arrow[ur] \arrow[dr] & ||c|ccc \arrow[r] & ||ccc \arrow[r] & |cc \arrow[r] & c \\
 & |||cc|ccc \arrow[ur] \arrow[r] & |||cccc \arrow[ur] & & &
\end{tikzcd}$$

Lemma 13: Given two ordered sequences of $\ell 1$ and $\ell 2$ objects, the numbers of different ordered sequences derived by merging them and preserving their suborders is $[\ell 1+\ell 2,\ \ell 1]$.

Proof: After merging the two given sequences, the resulting one has $\ell 1+\ell 2$ objects. The number of ways of picking from it $\ell 1$ objects is equal to the number of ways of merging those $\ell 1$ objects in the sequence with $\ell 2$ objects. □

Let us associate to each term t a pair of numbers $\langle n, \ell\rangle$ where n is the number of different reduction sequences in which we can reduce t, and ℓ is their common length. (Obviously, for any given term t all reduction sequences of t have the same length.) With the irreducible term c we associate the pair $\langle 1, 0\rangle$.

Theorem 14: If t = I t1 t2 and $\langle n1, \ell 1\rangle$ and $\langle n2, \ell 2\rangle$ are the pairs associated with t1 and t2 respectively, then $\langle n1 \cdot n2 \cdot [\ell 1+\ell 2, \ell 1],\ \ell 1+\ell 2+1\rangle$ is the pair associated with t.

Proof: The first component of the pair associated with t is obtained as follows. There are n_i reduction sequences of length ℓ_i when reducing ti to c, for i=1,2. A reduction sequence for t can be obtained by interleaving a reduction sequence of t1 with a reduction sequence of t2 and finally reducing Icc to c (and this last reduction can be done in a unique way). The number of those interleavings is given by the previous lemma. The second component of the pair associated with t is $\ell 1+\ell 2+1$ because for reducing t to c, we may reduce t1 to c in $\ell 1$ steps, t2 to c in $\ell 2$ steps and then Icc is reduced to c in 1 step. □

In general one can prove that when m-ary operators (with $m \geqslant 0$) are involved we can associate with $t = I\ t_1 \ldots t_m$ the pair:

$$\langle n_1 \cdot n_2 \cdot \ldots \cdot n_m \cdot [\ell_1 + \ldots + \ell_m, \ell_1] \cdot [\ell_2 + \ldots + \ell_m, \ell_2] \cdot \ldots \cdot [\ell_m, \ell_m],\ \sum_{i=1}^{m} \ell i + 1\rangle$$

where $\langle n_i, \ell_i\rangle$ is the pair associated with t_i for i=1, . . . , m.
This result derives from the previous result and a simple generalization of the binomial coefficients [Cohen 78].

2.7 Annotations denoting communications and communicating agents

This section outlines two other approaches to communications and parallelism in recursive equation programs, which I have pursued after the work described above. To keep the length of the thesis within reasonable bounds I give only a brief account of them here. The interested reader may refer to [Pettorossi and Skowron 82b, Pettorossi and Skowron 82a, Pettorossi 80b, Pettorossi and Skowron 83].
The first approach uses annotations [Schwarz 78] to denote communications, the second one introduces communicating agents.

2.7.1 Annotations denoting communications

Some of the ideas for the annotation language introduced in [Pettorossi 80b] are taken from [Dennis 74, Kahn and MacQueen 77, Hewitt and Baker 78, Hoare 78, Lauer, Torrigiani and Shields 79, Milner 80]. The major features of our approach are:
i) communications may be optional (written between slashes, i.e. /.../) or compulsory (written between exclamation marks, i.e. !...!). If optional they do not influence program correctness, but they improve program efficiency.
ii) There are either broadcast communications, or point-to-point communications. Point-to-point communications are messages sent to queues, which are available (for reading and writing) to all computing agents, i.e. all processes which perform the applications of recursive equations.
Queues are not "private" as in [Kahn and MacQueen 77], and they have names.
There are mechanisms for serializing read and write operations on queues, so that agents may be delayed when they refer to the queues used by writer processes.
iii) Communications denote facts, i.e. pieces of truth. The semantics of the data in the queues does not depend on when data are written or read, nor on the agents which operate on them.

More details can be found in [Pettorossi 80b].

We give here a simple example.

```
                                                            Program F1
fib(n) = if n=0 or n=1 then 1  !send(1,queue 0), send(1,queue 1)!
       else z  !send(z,queue n)!
            where z=fib(n-1)+b  !receivetoken(b,queue n-2)!
```

In Program F1, which compute the familiar Fibonacci function, there are queues whose names are natural numbers. The annotation !receivetoken(b,queue n-2)! means that a compulsory communication must take place between the computing agent which evaluates b and the queue n-2. That agent is blocked until a "token" (i.e. an integer) is placed into that queue, so that it can receive it by performing a receivetoken operation.
!send(1,queue 0), send(1,queue 1)! means that, after the evaluation of fib(0) [or fib(1)], the computing agent has to send to both queues 0 and 1 the value of fib(0) [or fib(1)]. The meaning of !send(z,queue n)! is analogous.
Program F1 has linear running time and it avoids repeated evaluations of recursive calls. Therefore it is an improvement upon the exponential running time program derivable from the Fibonacci definition. □

We give now another example where the use of compulsory communications avoids the repeated evaluation of function calls.
Let us consider the program:

```
                                                         Program F2.1
  sumsq(n) = if n=0 then 0 else sq(n)+sumsq(n-1)
     sq(n) = if n=0 then 0 else sq(n-1)+ 2n-1
```

where sumsq(n)=$0^2+1^2+...+n^2$ (see Sect. 2.4).
Obviously the running time for the above program is proportional to n^2. We can make it linear by adding some compulsory communications and using an auxiliary function as follows:

Program F2. 2

```
sumsq(n)  = if n=0 then 0 else sq(n)+sumsq1(n-1)
sumsq1(n) = if n=0 then 0 else z+sumsq1(n-1)  !receivetoken(z, queue n)!
     sq(n) = if n=0 then 0  !send(0, queue 0)!
                else  z !send(z, queue n)!
                      where z = sq(n-1)+2n-1
```

The function sumsq(n) is only required for the activation of the function sq(n), while sumsq1(n) does the job of summing up the squares produced by sq(n).

A receivetoken operation does not erase the received value from the corresponding queue: we say that it is not erasing communication. There are also other kinds of annotations denoting erasing communications: they are useful for solving synchronization problems [Pettorossi 80b]. □

A final example shows the use of optional and broadcast communications. Suppose we are given the following program, written in a HOPE-like notation, which computes the set of all leaves of a binary tree.

Program F3. 1

```
data btree(num) == niltree ++ tip(num) ++ btree(num)Δbtree(num)
dec leaves : btree(num) → set(num)
--- leaves(niltree) = {}
--- leaves(tip(ℓ)) = {ℓ}
--- leaves(t1Δt2)  = leaves(t1) U leaves(t2)
```

In order to compute the result we need in any case to perform the complete visit of the given btree, but an increase of efficiency is possible if we speed up the set-union operation. No matter what algorithm we choose for implementing that operation, it will require less time if we keep the sets involved as small as possible. This can be achieved by establishing some communications.

As soon as a leaf value, say ℓ, has been found, we can broadcast that information. In this way, when the value ℓ is encountered again, we may return {} instead of {ℓ}.

We can annotate program F3.1 and obtain the following one:

Program F3.2

```
data btree(num) == niltree ++ tip(num) ++ btree(num)Δbtree(num)
dec fastleaves : btree(num) → set(num)
--- fastleaves(niltree) = {}
--- fastleaves(tip(ℓ)) = if ℓ∈z /received(z)/    then {}
                         else {ℓ} /broadcast(ℓ)/
--- fastleaves(t1Δt2)  = fastleaves(t1) U fastleaves(t2)
```

where we used the following primitive operations for broadcast communications:

broadcast(n) where n is a data value, and

received(z) where z is a (local) variable.

Their meaning can be explained by assuming that there exists a global variable SET, initialized to the empty set, which can be updated and looked up during computation.

(i) broadcast(ℓ) makes SET to become SET U {ℓ};

(ii) received(z) binds z to the current value of SET.

By convention, the local variable z is bound to the empty set, if received(z) is not executed.

(Recall that communications between slashes are optional.)

Program F3.2 is an improvement upon program F3.1 because, by referring to the global variable SET, we do not collect leaves values already considered.

One can easily verify that, as usual, optional communications in program F3.2 do not affect program correctness but only program efficiency. □

2.7.2 Communicating Agents

This approach to communications in applicative languages is presented in [Pettorossi and Skowron 82b]. That work is motivated by the need of having a way of controlling point-to-point communications, so that they take place only "locally", and their routing, when many communications must occur, will not require much computational overhead.
For that purpose we introduced the notion of computing agent as a triple:

< agn, msg > :: expr,

where agn is an agentname, msg is a message associated with the agent and expr is an expression which should be worked on by the agent agn.
A recursive equation program is a set of rules for transforming sets of agents (written on the left hand side of the rules) into new sets of agents (written on the right hand side of those rules). The computation proceeds by nondeterministic application of a rule at a time, using a pattern matching mechanism à la Hope [Burstall, MacQueen and Sannella 80].

Here is a simple example.
Program F4, written in our language which we call Hope-C, sorts two bags (i.e. multisets) S and T of integers using two computing agents P and Q, associated with S and T, respectively. (This problem is due to Prof. Dijkstra.)
P and Q stop when S and T are transformed into S' and T' such that:
$|S|=|S'|$, $|T|=|T'|$, $S \cup T = S' \cup T'$, $a \leq b$ for any $a \in S'$ and any $b \in T'$.

Program F4

```
a. {<P,∇>::S} ⇐ {<P,max(S)>::S}
b. {<Q,∇>::T} ⇐ {<Q,min(T)>::T}
c. {<P,M>::S, <Q,m>::T} ⇐ {<P,∇>::S-M+m,
                             <Q,∇>::T-m+M}   if M > m
```

∇ (signifying 'no value') does not match any integer.

S−n means "subtract integer n from bag S"; S+n means "add integer n to bag S".

Rule a. says that the agent P computes the maximum integer in the bag S and stores it in its message. Analogously for rule b.
Rule c. says that whenever the agents P and Q have computed the values of their messages, they can swap them and update their bags accordingly.
The computation goes on by successive swappings of elements between P and Q, so that eventually each element in S is not greater of each element in T.

Program F2 evokes, for instance, the following computation:
$\{\langle P,\nabla\rangle::\{1,1,3\},\ \langle Q,\nabla\rangle::\{2,2\}\}$
$\rightarrow$(by a) $\{\langle P,3\rangle::\{1,1,3\},\ \langle Q,\nabla\rangle::\{2,2\}\}$ P computed max({1,1,3})
$\rightarrow$(by b) $\{\langle P,3\rangle::\{1,1,3\},\ \langle Q,2\rangle::\{2,2\}\}$ Q computed min({2,2})
$\rightarrow$(by c) $\{\langle P,\nabla\rangle::\{1,1,2\},\ \langle Q,\nabla\rangle::\{2,3\}\}$
P and Q swapped their messages and updated their bags.
$\rightarrow$(by b) $\{\langle P,\nabla\rangle::\{1,1,2\},\ \langle Q,2\rangle::\{2,3\}\}$ Q computed min({2,3})
$\rightarrow$(by a) $\{\langle P,2\rangle::\{1,1,2\},\ \langle Q,2\rangle::\{2,3\}\}$ P computed max({1,1,2}).
The computation halts here because no rule can be applied.

In general, program rules can be partitioned into two groups:
(i) rules concerning <u>one agent</u> only: they specify the agent behaviour "in isolation", i.e. when no interaction from other agents should be taken into account (see rules a. and b. of Program F4);
(ii) rules concerning <u>more than one agent</u>: they specify the interaction among agents and the way they cooperate with each other (see rule c.).

The evaluation process may be performed in a parallel way. Indeed, for each computation step, we can apply <u>many</u> rules at a time, provided that for any pair of rules the corresponding pair of sets of agents to which they are applied, are disjoint.
In particular in Program F4 we can reduce in one step

$\{\langle P,\nabla\rangle::(1,1,3),\ \langle Q,\nabla\rangle::(2,2)\}$ to $\{\langle P,3\rangle::(1,1,3),\ \langle Q,2\rangle::(2,2)\}$
by a parallel application of rules a. and b.

In order to compare our approach with the one in [Milner 80], we give here the CCS program corresponding to program F4.

Program F4.1

```
P(S) ⇐ αx.β̄(max(S)). if max(S) >x then P (S-max(S)+(x))
                                  else γ̄(S). NIL
Q(T) ⇐ ᾱ(min(T)).βy. if y>min(T) then Q (T-min(T)+(y))
                                  else δ̄(T). NIL
```

NIL denotes the CCS agent which "does nothing", and the communications between P and Q occur through the ports α and β, while ports γ and δ are used for sending out the resulting bags.
In program F4.1 the final result is obtained by one extra exchange of values w.r.t. the ones required in Program F4.

Program F4 can be improved by observing that elements can be swapped only once. Therefore we can save in separate subbags the elements received from the other agent and, in that way, we can speed up the computation of max and min. The improved rules a. and c. are given below. The bag of the agent P is split into bags S0 and Sr, where Sr stores the elements received from agent Q. Initially, S0=S and Sr=ϕ.
(Analogously for the bag of the agent Q.)

a'. $\{\langle P,\nabla\rangle::(S \mid Sr)\} \Leftarrow \{\langle P,\max(S)\rangle::(S \mid Sr)\}$

. . . . .

c'. $\{\langle P,M\rangle::(S \mid Sr),\ \langle Q,m\rangle::(T \mid Tr)\} \Leftarrow \{\langle P,\nabla\rangle::(S-M \mid Sr+m),$
$\langle Q,\nabla\rangle::(T-m \mid Tr+M)\}$ <u>if</u> $M>m$

□

As a final example let us consider again the program for computing $1^2+...+n^2$, when multiplications by 2 only are allowed (see section 2.4 and program F2.1 in this section).

We have the following computing agents:

(i) <sumsq, res>: : k where res stores the sum of the squares already computed and k is such that $(1^2+...+k^2)+res=1^2+...+n^2$;

(ii) for any k, $0 \leq k \leq n$, <sq•k, p>: : k where p is either the "empty value" ∇ or k^2.

Notice that the name of the agent is parametrized by "•k". This parametrization technique is useful when recursive calls have to be implemented via computing agents. For details see [Pettorossi and Skowron 82b].

As usual ∇ does not match any integer.

Program F5

```
1.   {<sumsq,∇>::k} ⟸ {<sumsq,0>::k, <sq•k,∇>::k}                    if k≥0
2.1  {<sq•k,∇>::k}  ⟸ {<sq•k,∇>::k, <sq•k-1,∇>::k-1}                 if k>0
2.2  {<sq•0,∇>::0}  ⟸ {<sq•0,0>::0}
2.3  {<sq•k+1,∇>::k+1, <sq•k,kto2>::k} ⟸ {<sq•k+1,kto2+2k+1>::k+1,
                                          <sq•k,kto2>::k}
3.   {<sumsq,res>::k, <sq•k,kto2>::k} ⟸ {<sumsq,res+kto2>::k-1}
```

Initially there exists only the agent <sumsq, ∇>: : n.

The result of the computation is given by the message component of the agent sumsq when it cannot longer be rewritten (i.e. when its expression is -1).

Rule 1. initializes the computation. Rules 2.1, 2.2 and 2.3 compute the values of k^2 for k=0,...,n by storing them in the message component msg of the agents <sq•k, msg>: : k. Notice the way in which recursion is implemented, and recursive calls are suspended, using the value ∇.

Rule 3. realizes the interaction between the agent sumsq and the various agents sq•k's. sumsq reads their messages and garbage-collects them. Indeed sq•k does not occurs in the r.h.s. of rule 3.
At the end of the computation there exists the agent sumsq only.

The recursive generation of the agents sq•k's can be avoided using the following program where there exists only one agent sq, together with sumsq, and it stores in its message the list of pairs $[k, k^2]$ for k=0,...,n.
: denotes list concatenation. During the computation, sq satisfies the following "agent invariant":

$<sq, [k, k^2]: ... : [0,0]>::n-k$ for k=0,...,n.

Program F5.1

```
1'. {<sumsq,∇>::n} ⇐ {<sumsq,0>::n, <sq,[0,0]>::n}                          if n>0
2'. {<sq,[k,kto2]:ℓ>::m} ⇐ {<sq,[k+1,kto2+2k+1]:[k,kto2]:ℓ>::m-1}            if m>0
3'. {<sumsq,res>::k, <sq,[k,kto2]:ℓ>::p} ⇐ {<sumsq,res+kto2>::k-1,
                                              <sq,ℓ>::p}
```

At the end of the computation the desired result is given by the message component of the agent sumsq. The agent sq is not garbage-collected. □

We are currently working on the development of the computing agents approach, and on the formalization of methods for proving the correctness of their behaviours.
In [Pettorossi and Skowron 83] complete logical theories are given for reasoning about parallel computations evoked by sets of agents.

2.8 Conclusions

In the first part of the thesis we analyzed a method for deriving program by transformations [Burstall and Darlington 77].
When using that approach the programmer first writes a very simple and maybe inefficient program, whose correctness can easily be shown, and then he tries to transform it in a more efficient one, by applying transformation rules which preserve correctness.
We devoted our attention to functional languages and, in particular, to recursive equation languages. They have been advocated because they allow easy proofs of program correctness and they are quite suitable for program transformation. We were able to convert many "ad hoc" heuristics and techniques, previously described in the literature, to a powerful strategy, called "tupling strategy".
The main idea consists in the "synchronization" of function evaluations, which share common subcomputations. Once the commom expressions are evaluated, their values are "sent" to the functions which need them. In that way, improvements in efficiency can be achieved, and we showed that optimal algorithms can be derived for recurrence relations evaluation.

The gain in efficiency relies on the fact that, using the tupling strategy, "general recursive" programs can often be transformed into "linear recursive" ones. We can then apply various techniques, already studied by other authors, for efficiently implementing linear recursion without using stacks.
The tupling strategy also allows tupled functions to be evaluated in a concurrent way because the various components can be independently evaluated. However, they need to be synchronized, so that the computation of the common subexpressions is not repeated.

We compared the power of the tupling strategy with other methods for eliminating redundancy in recursive programs, as the ones recently published in [Cohen 83]. It turns out that they are all particular cases of the tupling strategy.

At the end of the first part of the thesis we showed a "limitation result". We presented a class of recursive program schemas for which the tupling strategy cannot eliminate all redundant computations (if the number of the tupled functions is fixed at compile time). That result suggested us to look for ways of realizing communications among function calls "at run time".

In the second part of the thesis we looked at the memofunctions approach [Michie 68] as a first method of realizing run time communications among functions calls. We provided their operational semantics, and we presented their first formal treatment.
We gave an operational semantics definition for a language with memofunctions and a corresponding one for a language without memofunctions and we proved their consistency, in the sense that memoing improves efficiency and it preserves correctness.
We implemented our operational semantics definitions in Prolog and we studied various formal theories in which one can embed them.

Finally we mentioned some other approaches of allowing communications among functions calls. Those ideas of ours are still under development and they will be the object of our future study.

Appendix A

THE STATIC SEMANTICS FOR RECURSIVE EQUATIONS PROGRAMS

Let us first define the following three kinds of <u>formulas</u>:

1. $p \vdash e$ meaning that the expression e is well-formed w.r.t. the program p;

2. $p \vdash d$ meaning that the definition d is well-formed w.r.t. the program p;

3. $p \vdash req$ meaning that the recursive equation req is well-formed w.r.t. the program p.

We also need the following definitions:

1. the set of the <u>free variables</u> occurring in an expression e (denoted by FV(e)) and in a definition d (denoted by FV(d));

2. the set of the <u>defined variables</u> occurring in a definition d (denoted by DV(d)) and in a program p (denoted by DV(p)).

The following tables define FV(e) for any $e \in$ Exp, FV(d) and DV(d) for any $d \in$ Def, and DV(p) for any program p.

e	m	t	x	$bop_i(\ldots, e_i, \ldots)$	<u>if</u> e_0 <u>then</u> e_1 <u>else</u> e_2
FV(e)	ϕ	ϕ	$\{x\}$	$\cup_i \ FV(e_i)$	$FV(e_0) \cup FV(e_1) \cup FV(e_2)$

e	e <u>where</u> d	$f(\ldots,e_i,\ldots)$
FV(e)	[FV(e)\DV(d)] U FV(d)	$\{f\} \cup (\cup_i \; FV(e_i))$

Notice that the FV(cons(. . . ,be, . . .)) can be derived using the definition of $FV(bop_i(\ldots,e_i,\ldots))$ because cons $\in$ Bop.

d	x=e	$\langle x_1,\ldots,x_n\rangle$=e
FV(d)	FV(e)	FV(e)
DV(d)	$\{x\}$	$\{x_1,\ldots,x_n\}$

Given $p = \{f_1(\ldots) \Leftarrow e_1,\ldots,f_n(\ldots) \Leftarrow e_n\}$, we define: $DV(p) = \{f_1,\ldots,f_n\}$.

Now we can state <u>axioms</u> and <u>derivation rules</u> for well-formedness for basic expressions, expressions, definitions and recursive equations. We will state those notions with respect to a given program p.
This will allow us to define the notion of well-formedness of a program p as follows:

Definition 1: A program $p = \{req_1,\ldots,req_n\}$ is well-formed iff $\forall i \; 1 \leq i \leq n$ req_i is well-formed with respect to p.

For basic expressions and expressions we have:

$p \vdash m$ for any $m \in N$

$p \vdash t$ for any $t \in T$

$p \vdash x$ for any individual variable $x \in IVar$

$$\frac{p \vdash be_1,\ldots,p \vdash be_n}{p \vdash cons(be_1,\ldots be_n)} \qquad \text{for } be_1,\ldots,be_n \in BasExp$$

$$\frac{p \vdash e_1, \ldots, p \vdash e_n}{p \vdash bop(e_1, \ldots, e_n)} \qquad \text{for } e_1, \ldots, e_n \in Exp$$

$$\frac{p \vdash e_0,\ p \vdash e_1,\ p \vdash e_2}{p \vdash \underline{if}\ e_0\ \underline{then}\ e_1\ \underline{else}\ e_2} \qquad \text{for } e_1, \ldots, e_n \in Exp$$

Notice that, in order to conclude the well-formedness of the expression $\underline{if}$ e_0 $\underline{then}$ e_1 $\underline{else}$ e_2 we do not check whether or not e_0 is a boolean expression. Indeed we do not include any typechecking consideration in the definition of well-formedness.

We could have easily extended our rules to incorporate such typechecking, as we have done elsewhere in the thesis.

We omitted it here because now we are only interested in the issue of how free variables and defined variables effect well-formedness of recursive equations programs.

$$\frac{p \vdash e, \quad p \vdash d}{p \vdash e\ \underline{where}\ d} \qquad \text{for } e \in Exp,\ d \in Def,\ DV(d) \subseteq FV(e)$$

$$\frac{p \vdash e_1, \ldots, p \vdash e_n}{p \vdash f(e_1, \ldots, e_n)} \qquad \text{for } e_1, \ldots, e_n \in Exp \text{ and } f \in FVar$$

For definitions:.

$$\frac{p \vdash e}{p \vdash x{=}e} \qquad \text{for } e \in Exp \text{ and } x \notin FV(e)$$

$$\frac{p \vdash e}{p \vdash \langle x_1, \ldots, x_n \rangle{=}e} \qquad \text{for } e \in Exp \text{ and } \{x_1, \ldots, x_n\} \cap FV(e) = \emptyset$$

For recursive equations:

$$\frac{p \vdash be_1, \ldots, p \vdash be_n, p \vdash e}{p \vdash f(be_1, \ldots, be_n) \Leftarrow e} \quad \text{if } f \in DV(p) \text{ and } FV(e) \subseteq [DV(p) \cup (\cup_i FV(be_i))]$$

Notice that we did not allow recursion in definitions, so that for instance $x = x+2$ is not a well-formed definition. Notice also that if the recursive equation $f(\ldots) \Leftarrow e$ is well-formed w.r.t. the program p then $f \in DV(p)$.

Some other constraints can be introduced into the static semantics. In particular we can consider as well formed only recursive equation programs, whose functions are defined in such a way that they provide a "disjoint and exhaustive" case analysis on the structure of their data domains [Darlington 78].

For example, we can consider as well-formed only those programs whose definition for a function f defined on natural numbers, provides the values of f(0) and f(succ(n)). Analogously, for a function g operating on lists, we can consider as well-formed only those programs which have the following definitions:

$g(nil) \Leftarrow e_1$ and $g(a::\ell) \Leftarrow e_2$.

These constraints can be incorporated in the static semantics, but we did not do so, because it goes beyond our objectives here.

Appendix B
OPERATIONAL SEMANTICS FOR N-ARY FUNCTIONS

In this Appendix we would like to present an extension of the structural operational semantics definitions for including functions with $n \geq 0$ arguments. This extension is significant because in Chapter 2 we did not allow higher order functions, and therefore we could not use "curried functions". This appendix is an improved version of what is presented in [Plotkin 81] pages 117-121.

We need some extra auxiliary sets:

1) a set of Formals, called Forms, ranged over by form.

form ::= • | x: τ, form

where $\tau \in$ {int, bool} and x is an individual variable.

2) a set of Actual Expressions, called AcExp, ranged over by ae.

ae ::= • | e, ae

3) a set of Actual Expressible Types, called AcETypes, ranged over by eat.

aet ::= • | τ, aet   where $\tau \in$ {int, bool}

Notice that • and _,_ are overloaded symbols. They will be used in what follows as "nil" and "cons" for lists.

We also need to change some of the definitions we have already given for the language L.

(1) In Expressions instead of f(e) we have: f(ae)

(2) In Definitions we have:

d ::= ...(as for the language L)... | form=ae | f(form): τ=e

(3) Denotable Types, ranged over by dt $\in$ DTypes, are modified as follows:

dt ::= τ | aet $\rightarrow \tau$ where $\tau \in$ {int, bool}

(4) We need to change the notion of Abstracts, as well.

Let us consider the function T: Forms $\rightarrow$ AcEtypes s.t.

i) T($\bullet$) = $\bullet$ and ii) T(x: τ, form) = (τ, T(form)).

We define the predicate "form: : β" with form$\in$Forms and $\beta\in$TEnv as follows:

i) $\bullet :: \emptyset$ and ii) $$\frac{\text{form} :: \beta}{(x{:}\ \tau, \text{form}) :: \{x=\tau\} \cup \beta} \text{ if } x \notin DV(\text{form}) \cup \text{domain}(\beta).$$

We have: Abstracts = $\{ \lambda$form.e: τ | $\alpha[\beta] \vdash_{V \cup DV(form)}$ e: τ for some α: V and FV(e) $\subseteq$ V and form: : β $\}$

(5) The definitions of the free variables and defined variables are as follows:

For Expressions:

FV($\bullet$) = $\emptyset$ FV(e, ae) = FV(e) $\cup$ FV(ae)

For Abstracts: FV(λform.e: τ) = FV(e)\DV(form)

For Definitions:

	form=ae	f(form) : τ=e
FV	FV(ae)	FV(e)\DV(form)
DV	DV(form)	{f}

where DV : Forms $\rightarrow 2^{Var}$ is defined as follows:

DV($\bullet$) = $\emptyset$ and   DV(x: τ, form) = {x} $\cup$ DV(form).

Here are the rules for the <u>Static Semantics</u>, where $\alpha\vdash$. . . stands for $\alpha\vdash_V$. . . and $\alpha[\beta]\vdash$. . . stands for $\alpha[\beta]\vdash_{V \cup DV(form)}$. . . where form: : β.

Actual Expressions.

AE1 $\alpha \vdash \bullet : \bullet$.

AE2 $$\frac{\alpha \vdash e{:}\ \tau \qquad \alpha \vdash ae{:}\ aet}{\alpha \vdash (e, ae) : (\tau, aet)}$$

Application (updated)

A. $$\frac{\alpha \vdash ae: aet}{\alpha \vdash f(ae): et} \qquad \text{if } \alpha(f)=aet \rightarrow et$$

Abstracts (updated)

Ab. $$\frac{\alpha[\beta] \vdash e: \tau}{\alpha \vdash (\lambda form.\, e: \tau) \ : \ (T(form) \rightarrow \tau)} \qquad \text{if } form::\beta$$

<u>Definitions</u>

[form=ae] 1.1 $\alpha \vdash \bullet = \bullet$

1.2 $$\frac{\alpha \vdash (x: \tau)=e, \quad \alpha \vdash form=ae}{\alpha \vdash (x: \tau, form) = (e, ae)}$$

2. $$\frac{form::\beta \qquad \alpha \vdash ae: T(form)}{\vdash (form=ae): \beta}$$

[f(form) : τ=e]

1. $$\frac{\alpha[\beta] \vdash e: \tau}{\alpha \vdash f(form): \tau=e} \qquad \text{if } form::\beta$$

2. $\vdash (f(form): \tau=e): \ \{f=T(form) \rightarrow \tau\}$

Note. DV(form) = domain(β).

For the extended definition of the Dynamic Operational Semantics we need to introduce the following notions as well.

Given an actual expression ae and a type environment $\alpha \in TEnv_V$ s.t. $FV(ae) \subseteq V$, we define the following well-formedness formula:

$W_V(ae, \alpha)$ iff $\exists$ aet $\in$ AcETypes. $\alpha \vdash ae: aet$.

The set of Actual Expression Configurations w.r.t. a type environment $\alpha: V$ is:

$AE\Gamma_\alpha = \{\langle\rho, ae\rangle \mid W_V(ae, \alpha)$ and $\rho: \alpha$ and $FV(ae) \subseteq V\}$.

The set of Terminal Actual Expression Configurations, $TAE\Gamma$, is

a subset of $\bigcup_{\alpha \in TEnv_V} AE\Gamma_\alpha$ s.t. $\gamma \in TAE\Gamma$ iff the second component of γ

is of the form $\gamma 2 ::= \bullet \mid c, \gamma 2$ where $c \in DVal = N+T+Abstracts$.

We define a transition relation $\rightarrow_\alpha \subseteq \Gamma_\alpha \times \Gamma_\alpha$ where $\Gamma_\alpha = E\Gamma_\alpha \cup AE\Gamma_\alpha \cup D\Gamma_\alpha$ and we can partition that relation in the three subrelations $\rightarrow_{\alpha,e}$, $\rightarrow_{\alpha,ae}$ and $\rightarrow_{\alpha,d}$.

We allow ourselves to use the usual abbreviations. In particular, an element of $\rightarrow_{\alpha,ae}$ will also be written as $\vdash_\alpha \langle\rho 0, ae0\rangle \rightarrow \langle\rho 1, ae1\rangle$. If $\rho 0=\rho 1$ we also write $\rho 0 \vdash_\alpha ae0 \rightarrow ae1$.

We define:

1. the set of Actual Constants, ranged over by acon $\in$ ACon.

   acon ::= $\bullet$ | con, acon where con $\in$ N+T

2. an auxiliary function mkenv: Forms $\times$ Acon $\rightarrow$ Env s.t.

   mkenv($\bullet$, $\bullet$) = $\emptyset$

   mkenv((x: τ, form), (con, acon)) = {x=con} $\cup$ mkenv(form, acon).

The transition rules for the Dynamic Semantics definition are modified as follows. We write $\rho \vdash \ldots$ instead of $\rho \vdash_{\alpha} \ldots$

Actual Expressions:

AE1. $$\frac{\rho \vdash e \longrightarrow e'}{\rho \vdash (e, ae) \longrightarrow (e', ae)}$$

AE2. $$\frac{\rho \vdash ae \longrightarrow ae'}{\rho \vdash (e, ae) \longrightarrow (e, ae')}$$

Notice that, contrary to what is done in [Plotkin 81] page 121, we allow for nondeterministic evaluation of the expressions in actual expressions.

Application

A1. $$\frac{\rho \vdash ae \longrightarrow ae'}{\rho \vdash f(ae) \longrightarrow f(ae')}$$

A2. $\rho \vdash f(acon) \longrightarrow \underline{let}\ form = acon\ \underline{in}\ e \qquad \text{if } \rho(f) = \lambda form.\, e : \tau$

Rules A1 and A2 determine call-by-value evaluations.

Definitions

[form = ae] 1. $$\frac{\rho \vdash ae \longrightarrow ae'}{\rho \vdash form = ae \longrightarrow form = ae'}$$

2. $\rho \vdash form = acon \longrightarrow \rho_0$ where $\rho_0 = mkenv(form, acon)$

[f(form) : τ = e] $\rho \vdash f(form) : \tau = e \longrightarrow \{f = \lambda form. (\underline{let}\ \rho \lceil V\ \underline{in}\ e) : \tau\}$

where $V = FV(e) \backslash DV(form)$.

Appendix C
FORMAL THEORIES FOR OPERATIONAL SEMANTICS

In this appendix we consider a second approach for embedding structural operational semantics definitions into formal theories. The first one was presented in Section 2.5.
This approach is motivated by the desire of identifying the rewriting relation (denoted by $\rightarrow$) for terms and the implication relation (denoted by $\supset$) for formulas. Having one notion only, instead of two distinct ones, the deduction process may be made more efficient, because we can realize the improvements mentioned at the end of section 2.1 (where we suggested the use of rule $B1^+$ instead of rule B1). As we will see, we will find useful to introduce the reflexive transitive closure (denoted by $\rightarrow^*$) of the "one step" rewriting relation $\rightarrow$.

We will define two versions of this second approach and we will compare them. In both versions the meaning of the symbol $\supset$ (which we will call "hook" for avoiding confusion) is similar to the one of the logical implication. The second version differs from the first one in that the reflexivity and transitivity properties hold for hook.
For studies on logics with non-standard notions of implication the interested reader may refer to [Anderson, Belnap and Wallace 60, Anderson and Belnap 62]. In this appendix, $\supset$ denotes also the computation process of rewriting which was denoted by $\rightarrow$ in Section 2.5.

We consider a first order language where we choose $O = \{0_0, S_1, +_2\}$, $V = \{\}$, and $R = \{F_1\}$.

As usual, the subscripts denote the arities.

Terms are built as in the first interpretation (see Section 2.5), i.e. Terms=Exp, $e \in Exp$.

$$e ::= 0 \mid S(e) \mid e1 + e2$$

Atomic formulas are of the form: Fe, where $e \in Exp$, i.e. terms are injected in the atomic formulas via the unary relation symbol F.

The set of Formulas (denoted by Flma) is defined as the smallest set containing atomic formulas and closed w.r.t. $\supset$ (i.e. hook) and $\neg$.

Instead of writing atomic formulas as "Fe" we will often write them as "e" only: the reader will understand from the context whether "e" stands for the term "e" or the formula "Fe" (because, obviously, arithmetical operators require terms as operands, while $\neg$ and $\supset$ require formulas).

The modus ponens (MP), i.e. $\dfrac{\phi,\ \phi\supset\psi}{\psi}$ for any formula ϕ and ψ, is the only rule of inference we have.

We will consider the theory T with the following set A of axioms:

A1. $F(x+0) \supset F(x)$

A2. $F(x+S(y)) \supset F(S(x+y))$

A3. $(F(x) \supset F(y)) \supset (F(S(x)) \supset F(S(y)))$

A4. $(F(x) \supset F(y)) \supset (F(x+z) \supset F(y+z))$

A5. $(F(x) \supset F(y)) \supset (F(z+x) \supset F(z+y))$

Let Γ be a set of formulas s.t. $A \subseteq \Gamma$. By Gen(Γ) we denote the intersection of all sets Δ of formulas s.t. i) $\Gamma \subseteq \Delta$, ii) each instance of one of the nonlogical axioms A1,....,A5 is in Δ and iii) $\psi \in \Delta$ whenever $\phi \in \Delta$ and $\phi\supset\psi \in \Delta$.

We write $\Gamma \rhd \phi$ or $(\Gamma-A) \rhd \phi$ to denote that $\phi \in Gen(\Gamma)$.

If Γ=A we will also write $\rhd\phi$ instead of $A\rhd\phi$.

If $\Gamma\rhd\phi$ we will say that ϕ is a proper Γ-theorem.

If $\rhd\phi$ we will also say that ϕ is a proper theorem, instead of a proper A-theorem.

We can define the notion of a <u>proper Γ-proof</u>, as follows:

Definition 1: A finite sequence $\langle\psi_0,\ldots,\psi_{m-1}\rangle$ of formulas, with $m>0$, s.t. $\psi_{m-1}\equiv\phi$ and for each $i<m$ one of the following holds: (i) $\psi_i \in \Gamma$, (ii) ψ_i is an instance of one of the axioms A1,...,A5, (iii) $\psi_k \equiv \psi_j \supset \psi_i$ for some $j<i$ and $k<i$, is called a <u>proper Γ-proof</u> for ϕ.

Notice that we do <u>not</u> include in point (ii) of the above definition the instances of <u>logical axioms</u>, as, for example $\phi\supset(\psi\supset\phi)$. This limitation is necessary for establishing the connection between the deduction process (using "hook" as implication) and the process of computing the value of a given term (using "hook" as rewriting), as the following Theorem 5 shows.

One can prove the following properties for $\triangleright$, as usually done in a first order theory:

Theorem 2: Let $\Gamma,\Delta \subseteq$ Fima such that $A \subseteq \Gamma\cap\Delta$ and $\phi,\psi \in$ Fima.

(i) if $\Delta\subseteq\Gamma$ and $\Delta\triangleright\phi$ then $\Gamma\triangleright\phi$;

(ii) if $\Gamma\triangleright\phi$ then $\Theta\triangleright\phi$ for some finite subset Θ of Γ;

(iii) if $\Gamma\triangleright\chi$ for each $\chi\in\Delta$ and $\Delta\triangleright\phi$, then $\Gamma\triangleright\phi$;

(iv) if $\Gamma\triangleright\phi$ and $\Gamma\triangleright(\phi\supset\psi)$, then $\Gamma\triangleright\psi$;

(v) if $\Gamma\triangleright\phi$ then $\Gamma\vdash\phi$, where $\vdash$ denotes the usual "logical consequence" notion (where logical axioms also are allowed in proofs).

Here is presentation à la Hilbert of the proof that $\Gamma \triangleright F(S(S(S(0))))$ where $\Gamma = A \cup \{F(S(0) + S(S(0)))\}$. We will feel free to omit the symbol F for formulas and the parentheses for the argument of S.

1.	S0+SS0	given
2.	(S0+SS0) ⊃ S(S0+S0)	A2
3.	S(S0+S0)	1, 2, MP
4.	(S0+S0) ⊃ S(S0+0)	A2
5.	(S0+S0 ⊃ S(S0+0)) ⊃ (S(S0+S0) ⊃ SS(S0+0))	A3
6.	S(S0+S0) ⊃ SS(S0+0)	4, 5, MP
7.	SS(S0+0)	3, 6, MP

8.	$S0+0 \supset S0$	A1
9.	$(S0+0 \supset S0) \supset (S(S0+0) \supset SS0)$	A3
10.	$S(S0+0) \supset SS0$	8, 9, MP
11.	$(S(S0+0) \supset SS0) \supset (SS(S0+0) \supset SSS0)$	A3
12.	$SS(S0+0) \supset SSS0$	10, 11, MP
13.	$SSS0$	7, 12, MP

Now we will prove some theorems about the theory we have introduced. Let us first prove the following lemmas.

Let $\rightarrow$ denote the relation defined by axioms and rules S1, . . . ,S5 (see Section 2.5).

Lemma 3: If $a \rightarrow b$ then $a \neq b$.

Proof: By induction on the length of the derivation of $a \rightarrow b$. □

Lemma 4: Any formula ϕ s.t. $\triangleright\phi$ has one of the following forms:

either $F(e1) \supset F(e2)$

or $(F(e1) \supset F(e2)) \supset (F(S(e1)) \supset F(S(e2)))$

or $(F(e1) \supset F(e2)) \supset (F(e1+e) \supset F(e2+e))$

or $(F(e1) \supset F(e2)) \supset (F(e+e1) \supset F(e+e2))$

for some $e, e1, e2 \in Exp$.

Proof: By induction on the number m of applications of the MP rule.

<u>Base</u> If m=0 the lemma is obvious.

<u>Step</u> Suppose the lemma valid for m. If we apply the MP rule to any two formulas of the above specified forms we get a formula of the form $F(e1) \supset F(e2)$. □

Theorem 5: $\forall a, b \in Exp$ $a \rightarrow b$ iff $\triangleright$ $F(a) \supset F(b)$

Proof: (<u>only if part</u>). By induction on the length m of the deduction of $a \rightarrow b$.

<u>Base</u> If m=1 $a \rightarrow b$ is an instance of S1 (or S2). Using axiom A1 (or A2) we can derive $\triangleright$ $F(a) \supset F(b)$.

Step Suppose the theorem valid for a length m s.t. $m \geq 1$. Suppose that a $\longrightarrow$ b has been derived in m steps by using in the m-th step rule S4. (The other cases in which in the m-th step we used axiom S1 or S2 or rules S3 or S5 are analogous.) Therefore a $\longrightarrow$ b is of the form a1+a2 $\longrightarrow$ a1'+a2 and we have a derivation of a1 $\longrightarrow$ a1' in n<m steps. By induction hypothesis we have: $\triangleright$ F(a1) $\supset$ F(a1'). By axiom A4 we have: (F(a1) $\supset$ F(a1')) $\supset$ (F(a1+a2) $\supset$ F(a1'+a2)) and by modus ponens we derive: F(a1+a2) $\triangleright$ F(a1'+a2).

(if part). By induction on the length m of the deduction of F(a) $\supset$ F(b).

Base If m=1 and $\triangleright$ F(a) $\supset$ F(b) then F(a) $\supset$ F(b) is an instance of a non-logical axiom A1 or A2. Using S1 or S2 we have a $\longrightarrow$ b.

Step Suppose "if $\triangleright$F(a)$\supset$F(b) then a$\longrightarrow$b" valid for any deduction $\triangleright$ F(a) $\supset$ F(b) with length m s.t. $m > n \geq 1$ for some n. There are 2 cases:

(i) F(a) $\supset$ F(b) is an instance of axiom A1 (or A2). By axiom S1 (or S2) we can derive a $\longrightarrow$ b.

(ii) F(a) $\supset$ F(b) is derived by modus ponens from ϕ and $\phi \supset$ (F(a) $\supset$ F(b)) for some formula ϕ. By Lemma 4 ϕ is of the form F(c) $\supset$ F(d) where c,d $\in$ Exp. By induction hypothesis we have c $\longrightarrow$ d. There are 3 subcases:

1). a=S(c) and b=S(d). By applying rule S3 we can derive a $\longrightarrow$ b from c $\longrightarrow$ d.

2). a=c+e and b=d+e for some e $\in$ Exp. As subcase 1), applying rule S4.

3). a=e+c and b=e+d for some e $\in$ Exp. As subcase 1), applying rule S5. □

Proposition 6: If $\triangleright$ F(e1) $\supset$ F(e2) then e1$\neq$e2.

Proof: From the if-part of Theorem 5 and Lemma 3. □

Theorem 5 allows for the "hook" symbol, denoted by $\supset$, the same

interpretation given to $\longrightarrow$, i.e. we can interpret $F(a) \supset F(b)$ as "in one computation step from the expression a we can derive the expression b".
Theorem 5 holds because in the definition of the "proper proof of ϕ from A", i.e. "$\triangleright \phi$", we excluded the use of logical axioms. (Otherwise Lemma 4 on which Theorem 5 relies, would not hold.)
Indeed, if we allow logical axioms in the proofs we had to consider formulas as, for instance, $F(3) \supset (F(7) \supset F(3))$, (from the logical axiom $\phi \supset (\psi \supset \phi)$) for which the interpretation of $\supset$ as $\longrightarrow$ does not make sense.
Here is another example. For any formula ϕ we can prove that $\vdash \phi \supset \phi$ (where $\vdash$ denotes logical theorems) and, obviously, the interpretation of $\supset$ as $\longrightarrow$, i.e. "reduction in one step of the computation", would not be acceptable.

Unfortunately, for the notion of a proper proof we have introduced, the analogous of the Deduction Theorem is not valid. Indeed the following implication does not hold: for any $e1, e2 \in Exp$.
<u>If</u> $A \cup \{F(e1)\} \triangleright F(e2)$ <u>then</u> $A \triangleright F(e1) \supset F(e2)$.
A class of counterexamples can be given because for any $e \in Exp$. Indeed $A \cup \{F(e)\} \triangleright F(e)$ obviously holds, while $A \triangleright F(e) \supset F(e)$ does not hold (see Proposition 6).
However, one may want to have the Deduction Theorem valid for keeping for the 'hook' symbol that basic property of the logical implication. For regaining the validity of the Deduction Theorem we will introduce a slightly different notion of "proper proof" (thereby defining the second version of our interpretation of the operational semantics), while maintaining a simple computational interpretation for the 'hook' symbol $\supset$.

Definition 7: We define $\longrightarrow^* \subseteq Exp \times Exp$ as the reflexive transitive closure of $\longrightarrow$, i.e. $a \longrightarrow b$ iff ($\exists n \geq 0$ s.t. $a \longrightarrow b_0$ and $b_0 \longrightarrow b_1$ and ... and $b_{n-1} \longrightarrow b_n$ and $b_n = b$) or $a = b$.

Let us consider the following two axiom schemata (valid for any $e, e1, e2, e3 \in Exp$):

RA. <u>Reflexivity axiom</u> : $F(e) \supset F(e)$

TA. <u>Transitivity axiom</u>: $(F(e1) \supset F(e2)) \supset ((F(e2) \supset F(e3)) \supset (F(e1) \supset F(e3)))$

If $\Gamma = A \cup \{RA, TA\}$, instead of $\Gamma \rhd \phi$ we will also write $\rhd^* \phi$.

If $\Gamma = A \cup \{RA, TA\} \cup B$, instead of $\Gamma \rhd \phi$ we will also write $B \rhd^* \phi$.

Analogous properties to the ones stated for Theorem 2 hold also for $\rhd^*$. Notice that also for $\rhd^*$ we do not allow instances of logical axioms.

Lemma 8: Any formula ϕ s.t. $\rhd^* \phi$ has one of the following forms:

either (i) $F(e1) \supset F(e2)$

or (ii) $(F(e1) \supset F(e2)) \supset (F(e3) \supset F(e4))$

or (iii) $(F(e1) \supset F(e2)) \supset ((F(e2) \supset F(e3)) \supset (F(e1) \supset F(e3)))$,

for some $e1, e2, e3, e4 \in Exp$.

Proof: Analogous to the proof of Lemma 4. Notice that any instance of the RA (or TA) is of the form (i) (or (iii)).

Theorem 9: $\forall a, b \in Exp$ $a \longrightarrow^* b$ iff $\rhd^* F(a) \supset F(b)$.

Proof: (<u>only if part</u>). If $a=b$ then $\rhd^* F(a) \supset F(b)$ by RA. If $a \neq b$ then we can reason by induction on the number n of contraction steps. Suppose $a \longrightarrow b_0$ and $b_0 \longrightarrow b_1$ and... and $b_{n-1} \longrightarrow b_n = b$, i.e. we derived b from a in n+1 contraction steps. By induction hypothesis we have $\rhd^* F(a) \supset F(b_0)$ and $\rhd^* F(b_0) \supset F(b_1)$ ... and $\rhd^* F(b_{n-1})$. Using TA and MP we get $\rhd^* F(a) \supset F(b_n)$.

(<u>if part</u>). By induction on the length m of the proof $\rhd^* F(a) \supset F(b)$.

If $m=0$ and $F(a) \supset F(b)$ and $a=b$ we have $a \longrightarrow^* b$ by reflexivity.

If $m=0$ and $F(a) \supset F(b)$ and $a \neq b$ we have $a \longrightarrow^* b$ by axioms A1 or A2.

If $m=n+1$, $F(a) \supset F(b)$ is derived from ϕ and $\phi \supset (F(a) \supset F(b))$ where $\phi \equiv (F(c) \supset F(d))$ by Lemma 8. By induction hypothesis we have: $c \longrightarrow^* d$. Suppose $\rhd^* (F(c) \supset F(d)) \supset (F(a) \supset F(b))$ is a proof of length k with $k \leq n$. Let us reason by induction on k.

If k=0 the given formula is an instance of axiom A3 or A4 or A5. There are two cases:

- if $c=d$ then $a=b$ and we get $a \longrightarrow^* b$ by reflexivity;

- if $c \neq d$, from $c \longrightarrow^* d$ we get $c \longrightarrow^+ d$. Since k=0, a=C[c] and b=C[d] for some context C. By applications of axioms S3 or S4 or S5 we get $a \longrightarrow^* b$.
If $k=h+1$ for some $h \geq 0$, $(F(c) \supset F(d)) \supset (F(a) \supset F(b))$ is obtained from $(F(a) \supset F(c)) \supset ((F(c) \supset F(d)) \supset (F(a) \supset F(b)))$ as an instance of TA (and therefore d=b) and $F(a) \supset F(c)$.
By induction hypothesis on m we have $a \longrightarrow^* c$. Since $c \longrightarrow^* d$, we get: $a \longrightarrow^* d$ and $a \longrightarrow^* b$, because d=b. □

Lemma 10: For any $e \in$ Exp, any formula ϕ s.t. $\{F(e)\} \triangleright^* \phi$, i.e. ϕ is derived from F(e), or from any instance of the axioms A1, ..., A5, RA and TA, has one of the following forms: for some $a, b, c, d \in$ Exp
(i) $F(a)$, (ii) $F(a) \supset F(b)$, (iii) $(F(a) \supset F(b)) \supset (F(c) \supset F(d))$
(iv) $(F(a) \supset F(b)) \supset ((F(b) \supset F(c)) \supset (F(a) \supset F(c)))$.

Proof: Analogous to the proof of Lemma 4. Formulas of form (i) can be derived by MP from the ones of form (ii) using F(e) (or other formulas of form (i) previously derived). □

Lemma 11: For any $e, e1, e2 \in$ Exp, <u>if</u> $\{F(e1)\} \triangleright^* F(e) \supset F(e2)$ <u>then</u> $\triangleright^* F(e) \supset F(e2)$.

Proof: In the proof $\{F(e1)\} \triangleright^* F(a) \supset F(e2)$ every formula derived from F(e1) by MP is of the form F(a). Let C be the set of formulas $\{F(c)\}$ s.t. $c \in$ Exp and F(c) occurs as an element in the proof $\{F(e1)\} \triangleright^* F(e) \supset F(e2)$. By Lemma 10 any application of MP involving a formula in C produces a formula which also is in C. A formula of the form $F(e) \supset F(e2)$ can be obtained only from formulas of the form (ii), (iii) or (iv) (as given in Lemma 10). Therefore the formulas in C can be discarded from the proof $\{F(e1)\} \triangleright^* F(e) \supset F(e2)$, thereby obtaining the proof $\triangleright^* F(e) \supset F(e2)$. □

The following theorem holds for $\triangleright^*$.

Theorem 12: <u>Deduction Theorem for $\triangleright^*$</u>. $\forall\ e1, e2 \in$ Exp. <u>if</u> $\{F(e1)\} \triangleright^* F(e2)$ <u>then</u> $\triangleright^* F(e1) \supset F(e2)$.

Proof: If e1=e2, $\triangleright^*$ F(e1) $\supset$ F(e1) because F(e1) $\supset$ F(e1) is an instance of the axiom RA.

If e1$\neq$e2 we reason by induction on the length m of the formal proof $\langle\phi_0,\ldots,\phi_{m-1}\rangle$ of F(e2) $\equiv \phi_{m-1}$ from {F(e)}.

By Lemma 10 F(e2) is derived by modus ponens MP from {F(e1)} $\triangleright^*$ F(a) and {F(e1)} $\triangleright^*$ F(a) $\supset$ F(e2) for some a $\in$ Exp. By induction hypothesis from {F(e1)} $\triangleright^*$ F(a) we get: $\triangleright^*$ F(e1) $\supset$ F(a).

We also have, as an instance of axiom TA:

$\triangleright^*$ (F(e1) $\supset$ F(a)) $\supset$ ((F(a) $\supset$ F(e2)) $\supset$ (F(e1) $\supset$ F(e2)) for any a, e1, e2 $\in$ Exp.

By MP we have: $\triangleright^*$ ((F(a) $\supset$ F(e2) $\supset$ (F(e1) $\supset$ F(e2)). (1)

By Lemma 11 from {F(e1)} $\triangleright^*$ F(a) $\supset$ F(e2) we get:

$\triangleright^*$ F(a) $\supset$ F(e2). (2)

and by MP from (1) and (2) we obtain: $\triangleright^*$ F(e1) $\supset$ F(e2). □

The converse of the Deduction Theorem is stated by the following proposition.

Proposition 13: $\forall$ e1, e2 $\in$ Exp. <u>if</u> $\triangleright^*$ F(e1) $\supset$ F(e2) <u>then</u> {F(e1)} $\triangleright^*$ F(e2).

Proof: Obviously if $\triangleright^*$ ϕ then {F(e1)} $\triangleright^*$ ϕ. Thus, from $\triangleright^*$ F(e1) $\supset$ F(e2) we have: {F(e1)} $\triangleright^*$ F(e1) $\supset$ F(e2). Since {F(e1)} $\triangleright^*$ F(e1), we get the thesis by MP. □

Moreover, as one can easily see, $\forall$ e1, e2 $\in$ Exp <u>if</u> {F(e1)} $\triangleright$ F(e2) <u>then</u> {F(e1)} $\triangleright^*$ F(e2), and <u>if</u> {F(e1)} $\triangleright^*$ F(e2) <u>then</u> {F(e1)} $\vdash$ F(e2), where $\Gamma \vdash \phi$ denotes the usual notion of Γ-theorem in a first order theory.

Theorem 14: $\forall$e, e1 $\in$ Exp e$\rightarrow^*$e1 iff {F(e)} $\triangleright^*$ F(e1).

Proof: $\forall$e, e1 $\in$ Exp. e $\rightarrow^*$ e1 iff $\triangleright^*$ F(e) $\supset$ F(e1) by Theorem 9.

$\triangleright^*$ F(e) $\supset$ F(e1) iff {F(e)} $\triangleright^*$ F(e1) by Theorem 12 and Proposition 13.

□

Therefore the notion of computation according to our operational semantics rules corresponds to the notion of $\triangleright^*$ proof.

In general $\forall e, e' \in Exp \triangleright F(e) \supset F(e')$ iff $\triangleright^* F(e) \supset F(e')$ does not hold (e.g. consider the case $e=e'$), however the following theorem holds.

Theorem 15: $\{F(e)\} \triangleright F(e')$ iff $\{F(e)\} \triangleright^* F(e')$.

Proof: (<u>only if part</u>). Obvious.

(<u>if part</u>). By induction on the length ℓ of the derivation of $e \longrightarrow e'$.

If $\ell=0$: $e=e'$ and obviously $\{F(e)\} \triangleright F(e)$.

Assume the theorem true for $\ell=k$.

Suppose that $e \longrightarrow e_1 \longrightarrow \ldots \longrightarrow e_k \longrightarrow e_{k+1}=e'$.

We have $\{F(e)\} \triangleright F(e_k)$. We also have $\triangleright F(e_k) \supset F(e_{k+1})$ by Theorem 5.

Therefore $\{F(e)\} \triangleright F(e_k) \supset F(e_{k+1})$. By modus ponens we get the thesis. $\square$

This second version of the second interpretation for the operational semantics rules has a particular merit w.r.t. the first version. In order to illustrate it, let us consider the following "tree presentation" of the proof of $\{S0+SS0\} \triangleright SSS0$, we have given earlier.

$$\begin{tikzcd}
S0+SS0 & \supset & S(S0+S0) & \supset & SS(S0+0) & \supset & SSS0 \\
 & A2 \arrow[u] & & S0+S0 \supset S(S0+0) \arrow[u] & & S(S0+0) \supset SS0 \arrow[u] & \\
 & & & A2 \arrow[u] & & S0+0 \supset S0 \arrow[u] & \\
 & & & & & A1 \arrow[u] &
\end{tikzcd}$$

The arrow $\uparrow$ shows an application of the MP rule.

Here is a proof of $\{S0+SS0\} \triangleright^* SSS0$.

1.	$S0+SS0$	given
2.	$(S0+SS0) \supset S(S0+S0)$	A2

3.	$S(S0+S0)$	1, 2, MP
4.	$S0+S0 \supset S(S0+0)$	A2
5.	$S0+0 \supset S0$	A1
6.	$(S0+0 \supset S0) \supset (S(S0+0) \supset SS0)$	A3
7.	$S(S0+0) \supset SS0$	5, 6, MP
8.	$(S0+S0 \supset S(S0+0)) \supset ((S(S0+0) \supset SS0) \supset (S0+S0 \supset SS0))$	TA
9.	$(S(S0+0) \supset SS0) \supset (S0+S0 \supset SS0)$	4, 8, MP
10.	$S0+S0 \supset SS0$	7, 9, MP
11.	$(S0+S0 \supset SS0) \supset (S(S0+S0) \supset SSS0)$	A3
12.	$S(S0+S0) \supset SSS0$	10, 11, MP
13.	$SSS0$	3, 12, MP

It can be depicted as follows:

$$\begin{tikzcd}
S0+SS0 & \supset & S(S0+S0) & & \supset & SSS0 \\
& A2 \arrow[u] & & & S0+S0 \supset SS0 \arrow[u] & \\
& & & S0+S0 \supset S(S0+0) \arrow[ur, "\text{MP twice from TA}"] & & S(S0+0) \supset SS0 \arrow[ul] \\
& & & A2 \arrow[u] & & S0+0 \supset S0 \arrow[u] \\
& & & & & A1 \arrow[u]
\end{tikzcd}$$

In that proof we see that the "surface expression" (the one in which we are interested) is transformed only twice, instead of three times, as in $\{S0+SS0\} \rhd SSS0$, at the expense of making the "supporting proofs" more elaborate by using the transitivity axiom.

The number of applications of the modus ponens rule (MP) in the two cases is the same. Regarding the modus ponens application as the time unit, we could say that the two proofs have the same "time complexity". But those proofs have different "time×space complexity": the surface expression, in fact, which is usually bigger than the other expressions, is transformed fewer times in the $\triangleright^*$ proof w.r.t. the $\triangleright$ proof. Therefore we could say that in $\triangleright^*$ proofs have better time$\times$space efficiency.

The result of Theorem 14 can be extended to the case in which the set of premises for $\triangleright^*$ is not a singleton set, but any <u>set of atomic formulas</u>. That case provides a model for the <u>concurrent evaluation</u> of expressions. Notice, however, that particular care should be taken because <u>if</u> $\{F(e1), F(e2)\} \triangleright^* F(e)$ <u>then</u> $\{F(e1)\} \triangleright^* F(e)$ or $\{F(e2)\} \triangleright^* F(e)$, but in general not both, because the arithmetic value of e1 may be different from the one of e2. Therefore, from Theorem 14 we will have that: $e1 \longrightarrow^* e$ or $e2 \longrightarrow^* e$, but we do not know which of the two subformulas holds.
In order to avoid that ambiguity, we may associate the tag '1' with F(e1) and the tag '2' with F(e2). We will then associate with any formula of the form F(e) s.t. $\{F(e1), F(e2)\} \triangleright^* F(e)$ the tag '1' (or '2') according to the fact that the final application of MP for deriving the F(e) in question, was made by using $F(e_k) \supset F(e)$ where '1' (or '2') was the tag of $F(e_k)$.

Therefore, the second version we presented for the interpretation of the operational semantics (i.e. the one which uses the extra axioms RA and TA) allows the parallel evaluation of various expressions, with the advantage of making the deductions needed for evaluating an expression, available also for other evaluations.
For instance, if we have to evaluate both 5+(3+2+1) and (3+3)+7 we may use, during the evaluation of the second expression, the fact "$\triangleright^* F(3+3) \supset F(6)$" derived in evaluating the first expression. In that way in parallel computations we may eliminate redundant deductions, at the expense of keeping track of the already proved facts.
In order to use the result of Theorem 9 for evaluating expressions, we need to

be sure that $D^* F(a) \supset F(b)$ holds for $a \neq b$. Only in that case, in fact, we know that at least one computation step has been performed when replacing a by b. By iterating this derivation procedure, we eventually get the value of the given expression a.

C.1 Models

We would like to present the models for the theories defined by the axioms A and the axioms A $\cup$ $\{RA, TA\}$.

We first consider the <u>theory for the A axioms</u>.

Let us define a class K of models U s.t. $U = \{\text{Terms}, f, R\}$ where Terms=Exp, f is defined as in the first interpretation (see Sect. 2.5.1) and for the unary predicate symbol F the relation R has the property that $\forall e, e' \in \text{Exp}$ the following two conditions are equivalent:

(Ri) if $F_R(e)$ then $F_R(e')$ and (Rii) $e \longrightarrow e'$,

where as usual, $\longrightarrow$ is the relation derived from axioms and rules S1, . . . , S5 (see section 2.5).

The satisfiability relation $U \models \phi$ for a formula ϕ in a structure U is defined as follows:

$U \models F(e)$ iff $F_R(e)$

$U \models \phi \supset \psi$ iff <u>either</u> not($U \models \phi$) <u>or</u> $U \models \psi$.

We will say that U is a model of ϕ if $U \models \phi$.

$K \models \phi$ means that $\forall\ U \in K.\ \ U \models \phi$.

Lemma 16: $K \models A$.

Proof: For A1: we need to show: $K \models F(x+0) \supset F(x)$.

Suppose: $U \not\models F(x+0) \supset F(x)$ for some $U \in K$. We have:

$U \models F(x+0)$ iff $F_R(x+0)$ and $U \not\models F(x)$ iff $\neg F_R(x)$.

Thus $\neg(x+0 \longrightarrow x)$ but this contradicts S1.

For A3: we need to show: $K \models (Fx \supset Fy) \supset (FSx \supset FSy)$.

Suppose: (i) $U \models Fx \supset Fy$ and (ii) $U \not\models FSx \supset FSy$ for some $U \in K$.

By (i) we have: $x \longrightarrow y$ and therefore $Sx \longrightarrow Sy$ by S3. This contradicts (ii).

Analogously for the axioms A2, A4 and A5. $\square$

Theorem 17: $\forall e, e' \in \mathrm{Exp}$. $\rhd F(e) \supset F(e')$ iff $K \models F(e) \supset F(e')$.

Proof: (<u>only if part</u>). By Theorem 5 $\forall e, e' \in \mathrm{Exp}$. $\rhd F(e) \supset F(e')$ iff $e \rightarrow e'$.

Suppose $K \not\models F(e) \supset F(e')$. For some $U \in K$. $U \not\models F(e) \supset F(e')$. We get: $U \models F(e)$ and $U \not\models F(e')$. Thus $F_R(e)$ and $\neg F_R(e')$. This contradicts the definition of R in K.

(<u>if part</u>). Assume $K \models F(e) \supset F(e')$. To show: $e \rightarrow e'$.

Assume $\neg(e \rightarrow e')$. By definition of K: $e \rightarrow e'$ iff (if $F_R(e)$ then $F_R(e')$) for any model $U \in K$. By $\neg(e \rightarrow e')$ we have $F_R(e)$ and $\neg F_R(e')$ for any model $U \in K$. Contradiction. □

Theorem 18: Let Γ be a set of atomic formulas. If $\Gamma \rhd \phi$ then $A \cup \Gamma \models \phi$.

Proof: If $\Gamma \rhd \phi$ then $\Gamma \cup A \vdash \phi$. By Completeness Theorem $A \cup \Gamma \models \phi$.
□

Theorem 19: $\forall e, e' \in \mathrm{Exp}$. <u>if</u> $\{F(e)\} \rhd F(e')$ <u>then</u> $A \cup \{F(e)\} \models F(e')$.

Proof: It is a corollary of Theorem 18. □

The reverse implication of Theorem 19 is also true, as we will show in Theorem 24.

Now we turn to the definition of the models of the <u>theory for A ∪ {RA, TA} axioms</u>.

We will characterize the relation $\rhd^*$ using the class K′ of models.
A model $U \in K'$ is a triple { Terms, f, R } where Terms and f are defined as in K and R has the property that for all $e, e' \in \mathrm{Exp}$ the following two conditions are equivalent:

(Ri) if $R_R(e)$ then $F_R(e')$ and (Rii) $e \rightarrow^* e'$

where $\rightarrow^*$ is the reflexive transitive closure of the relation $\rightarrow$ as derived from the axioms S1, ..., S5 (see section 2.5).

Lemma 20: $K' \models A \cup \{RA, TA\}$.

Proof: Analoguos to the proof of Lemma 16. $\Box$

Theorem 21: $\forall e, e' \in Exp.$ $\rhd^* F(e) \supset F(e')$ iff $K' \models F(e) \supset F(e')$.

Proof: (<u>only if part</u>). Assume $\rhd^* F(e) \supset F(e')$. We have: $A \cup \{RA, TA\} \vdash F(e) \supset F(e')$ where $\Gamma \vdash \phi$ denotes the usual notion of Γ-theorem in a first order theory (including logical axioms in proofs).

By the Completeness Theorem: $A \cup \{RA, TA\} \models F(e) \supset (e')$.

By Lemma 20 we get: $K' \models F(e) \supset F(e')$.

(<u>if part</u>). Assume $K' \models F(e) \supset F(e')$.

For any $U \in K'$, $U \models F(e) \supset F(e')$ iff $e \longrightarrow^* e'$ (by definition of K') iff $\rhd^* F(e) \supset F(e')$ (by Theorem 9). $\Box$

Theorem 22: Let Γ be a set of atomic formulas and A be the set of axioms A1,....,A5. If $\Gamma \rhd^* \phi$ then $A \cup \{RA, TA\} \cup \Gamma \models \phi$.

Proof: If $\Gamma \rhd^* \phi$ then $\Gamma \cup A \cup \{RA, TA\} \vdash \phi$. By the Completeness Theorem we get the thesis. $\Box$

Theorem 23: $\forall e, e' \in Exp.$ $\{F(e)\} \rhd^* F(e')$ iff $A \cup \{RA, TA\} \cup \{F(e)\} \models F(e')$.

Proof: (<u>only if part</u>). See theorem 22.

(<u>if part</u>). Assume $A \cup \{RA, TA\} \cup \{F(e)\} \models F(e')$.

From the hypothesis we get: $A \cup \{RA, TA, F(e)\} \vdash F(e')$ by the Completeness Theorem.

By the Deduction Theorem we have: $A \cup \{RA, TA\} \vdash F(e) \supset F(e')$.

By the Completeness Theorem we get: $A \cup \{RA, TA\} \models F(e) \supset F(e')$.

Since $K' \models A \cup \{RA, TA\}$ we get: $K' \models F(e) \supset F(e')$, then by Theorem 21 we have: $\rhd^* F(e) \supset F(e')$. Since $\{F(e)\} \rhd^* F(e)$ and $\{F(e)\} \rhd^* F(e) \supset F(e')$, by MP we get: $\{F(e)\} \rhd^* F(e')$. $\Box$

Theorem 24: $\forall e, e' \in Exp.$ $\{F(e)\} \rhd F(e')$ iff $A \cup \{F(e)\} \models F(e')$.

Proof: Since $\{F(e)\} \rhd F(e')$ implies $A \cup \{F(e)\} \models F(e')$ (see Theorem 19), it is enough to show that: if $A \cup \{F(e)\} \models F(e')$ then $A \cup \{F(e), RA, TA\} \models F(e')$, because $A \cup \{RA, TA, F(e)\} \models F(e')$ iff $\{F(e)\} \rhd^* F(e')$ by Theorem 23 iff $\{F(e)\} \rhd F(e')$ by Theorem 15.

Assume $A \cup \{F(e)\} \models F(e')$ and $A \cup \{F(e), RA, TA\} \not\models F(e')$.

Thus for some U s.t. $U \models A \cup \{F(e), RA, TA\}$ we have $U \not\models F(e')$

We have: $U \models A \cup \{Fe\}$ and therefore, since $A \cup \{F(e)\} \models F(e')$, we get $U \models Fe'$. Contradiction. $\square$

C. 2 Summary of results

The following tables summarize the main definitions and results of Section 2. 5 and this Appendix.

A. OPERATIONAL SEMANTICS

$e ::= 0 \mid S(e) \mid e1+e2$ $\qquad$ $e \in Exp$

Axioms and rules S:

S1. $e+0 \longrightarrow e$ $\qquad$ S2. $e1+S(e2) \longrightarrow S(e1+e2)$

S3. $\dfrac{e \longrightarrow e'}{S(e) \longrightarrow S(e')}$

S4. $\dfrac{e1 \longrightarrow e1'}{e1+e2 \longrightarrow e1'+e2}$ $\qquad$ S5. $\dfrac{e2 \longrightarrow e2'}{e1+e2 \longrightarrow e1+e2'}$

B. FIRST INTERPRETATION IN A FIRST ORDER THEORY. (Section 2. 5)

Atomic formulas: $e \longrightarrow e'$ $\qquad$ $e, e' \in Exp$

Formulas: the least set containing atomic formulas and closed w. r. t. $\neg$ and $\supset$.

Axioms: logical axioms and nonlogical axioms S':

S1'. $e+0 \longrightarrow e$

S2'. $e1+S(e2) \longrightarrow S(e1+e2)$

S3'. $(e \longrightarrow e') \supset (S(e) \longrightarrow S(e'))$

S4'. $(e1 \longrightarrow e1') \supset (e1+e2 \longrightarrow e1'+e2)$

S5'. $(e2 \longrightarrow e2') \supset (e1+e2 \longrightarrow e1+e2')$

Inference rule: modus ponens (MP) $\dfrac{\phi, \quad \phi \supset \psi}{\psi}$

Theorem. $e \longrightarrow e'$ in S iff $S' \vdash e \longrightarrow e'$.

C. SECOND INTERPRETATION IN A FORMAL THEORY

First version:

Atomic formulas: $F(e)$ $\qquad e \in \mathrm{Exp}$

Formulas: the least set containing atomic formulas and closed w.r.t. $\supset$ and $\neg$.

Axioms A: A1. $F(e+0) \supset F(e)$

A2. $F(e1+S(e2)) \supset F(S(e1+e2))$

A3. $(Fe \supset Fe') \supset (F(S(e)) \supset F(S(e')))$

A4. $(Fe1 \supset Fe1') \supset (F(e1+e2) \supset F(e1'+e2))$

A5. $(Fe2 \supset Fe2') \supset (F(e1+e2) \supset F(e1+e2'))$

(For simplicity reasons we did not write some obvious parentheses).

Inference rule: modus ponens (MP) $\dfrac{\phi, \quad \phi \supset \psi}{\psi}$

$\rhd\ \phi$ means that ϕ can be derived from the A axioms using MP (logical axioms are not allowed). If $\rhd\ \phi$ we say that ϕ is a proper theorem (in the first version). We will also write $A \rhd \phi$ instead of simply $\rhd\ \phi$.

$\Gamma \rhd \phi$ means that ϕ can be derived from the formulas Γ, the axioms A and MP (logical axioms are not allowed). In that case we say that ϕ is a proper Γ-theorem (in the first version).

Theorem. $\forall e, e' \in \mathrm{Exp}.$ $e \longrightarrow e'$ iff $\rhd\ F(e) \supset F(e')$.

Note. The analogous of the Deduction Theorem does not hold:

If $\{F(e)\} \rhd F(e')$ it does not follow that $\rhd\ F(e) \supset F(e')$.

Second version: as in the 1st version, except the following 2 extra axioms:

RA. $\forall e \in \mathrm{Exp}.$ $F(e) \supset F(e)$

TA. $\forall e1, e2, e3 \in \mathrm{Exp}.$

$(F(e1) \supset F(e2)) \supset ((F(e2) \supset F(e3)) \supset (F(e1) \supset F(e3)))$

$\triangleright^* \phi$ means that ϕ can be derived from the axioms $A \cup \{RA, TA\}$ using MP (logical axioms are not allowed). If $\triangleright^* \phi$ we say that ϕ is a proper theorem (in the second version).

$\Gamma \triangleright^* \phi$ means that ϕ can be derived from the formulas Γ, the axioms $A \cup \{RA \cup TA\}$ using MP (logical axioms are not allowed). In that case we say that ϕ is a proper Γ-theorem (in the second version).

<u>Theorem</u>. $\forall e, e' \in Exp.$ $e \rightarrow^* e'$ iff $\triangleright^* F(e) \supset F(e')$ iff $\{F(e)\} \triangleright^* F(e')$.

$\rightarrow^*$ denotes the reflexive transitive closure of $\rightarrow$.

<u>Note</u>. The analogous of the Deduction Theorem does hold.

<u>Theorem</u>. $\{F(e)\} \triangleright F(e')$ iff $\{F(e)\} \triangleright^* F(e')$ iff $A \cup \{F(e)\} \models F(e')$ iff $A \cup \{RA, TA\} \cup \{F(e)\} \models F(e')$.

Appendix D
TABLE OF SYMBOLS

domain(f)	domain of the function f, also written as dom(f).
$A \setminus B$	$x \in A \setminus B$ iff $x \in A$ and $x \notin B$.
$f[a \mid b]$	function updating. $f[a\mid b](x) = f(x)$ if $x \neq a$. $f[a\mid b](a) = b$.
$f \upharpoonright A$	function restriction. $\text{domain}(f \upharpoonright A) = \text{domain}(f) \cap A$.
$f \setminus A$	function difference. $\text{domain}(f \setminus A) = \text{domain}(f) - A$.
$f[g]$	updating the function f by the function g. <u>if</u> $x \in \text{dom}(g)$ <u>then</u> $f[g](x) = g(x)$ <u>else</u> $f[g](x) = f(x)$.
+	disjoint union of sets or integer addition.
FV(e), FV(d)	free variables in an expression e and a definition d (Sect. 2.1).
DV(e), DV(d)	defined variables in an expression e and a definition d (Sect. 2.1).
M(e)	translation from an expression e to M(e) by erasing <u>all</u> keywords <u>rec</u>, and replacing <u>some</u> occurrences of "f(e1)", where f is recursively defined, by "f(e1) <u>valueof</u> f <u>at</u> e1". (Sect. 2.4)
E[...]	an atomic expression context (Sect. 2.3) or an expression context (Sect. 2.4)
$\rightarrow$	term rewriting ($t1 \rightarrow t2$) or function definition ($f : A \rightarrow B$) or logical implication.
$\rightarrow^{+}$	transitive closure of $\rightarrow$.

$\rightarrow^*$	reflexive transitive closure of $\rightarrow$ or terminal reflexive transitive closure of $\rightarrow$ (see the Prolog implementation: Sect. 2.6).
$\rightarrow\$$	functional types (see the Prolog implementation).
$(L)\rightarrow$	rewritings of expressions in the language L.
$(memoL)\rightarrow$	rewritings of expressions in the language memoL.
$e\,[\,e1\,/\,x\,]$	substitution of the free occurrences of the variable x in an expression e by the expression e1.
$\rho\vdash_{\alpha} e1 \approx e2$	equivalence of two expressions (see Sect. 2.4). (We may omit to write α).
$\rho\vdash_{\alpha} e1 \rightarrow e2$	rewriting of the expression e1 into e2 in the environment ρ and the type-environment α (see Chap. 2). (We may omit to write α).
$\alpha \vdash_V e:t$	well-formedness for the expression e (Sect. 2.1).
$\alpha \vdash_V d$	well defined definition d (Sect. 2.1)
$\vdash_V d:\beta$	(strong) agreement of a definition d and a type-environment β.
$\rho:\alpha$	agreement of an environment ρ and a type-environment α.
$\mu:\alpha$	agreement of the memo environment μ with the type-environment α.
$\Gamma \rhd \xi$	ξ is a proper Γ-theorem. (Logical axioms are <u>not</u> allowed in proofs).
$\rhd^* \xi$	it means: $\{A1,\ldots,A5,RA,TA\} \rhd \xi$.
$B \rhd^* \xi$	it means: $B \cup \{A1,\ldots,A5,RA,TA\} \rhd \xi$.
$\Gamma \vdash \xi$	usual notion of Γ-theorem in first order theories. (Logical axioms are allowed in proofs).
$U \models \xi$	satisfiability relation for a formula ξ in a structure U
$\xi \supset \psi$	logical implication or "hook".

References

[Anderson and Belnap 62]
Anderson, A. R. and Belnap, N. D. Jr.
The Pure Calculus of Entailment.
The Journal of Symbolic Logic 27: 19-52, 1962.

[Anderson, Belnap and Wallace 60]
Anderson, A. R., Belnap, N. D. Jr. and Wallace, J. R.
Independent Axiom Schemata for the Pure Theory of Entailment.
Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 6: 93-95, 1960.

[Arsac and Kodratoff 82]
Arsac, J. and Kodratoff, Y.
Some techniques for recursion removal from recursive functions.
ACM Trans. Progr. Lang. Syst. 4(2): 295-322, 1982.

[Ashcroft and Wadge 77]
Ashcroft, E. A. and Wadge, W. W.
Lucid, a Nonprocedural Language with Iteration.
Communication ACM 20(7): 519-526, 1977.

[Atkinson 81] Atkinson, M. D.
The Cyclic Towers of Hanoi.
Information Processing Letters 13(3): 118-119, 1981.

[Aubin 79] Aubin, A.
Mechanizing structural induction: Part I and II.
Theor. Comput. Sci. 9(3): 329-362, 1979.

[Backus 78] Backus, J.
Can programming be liberated from the von Neumann style?
Communications of the ACM 21(8):613-641, 1978.

[Barendregt 81] Barendregt, H.P.
The lambda calculus, its syntax and semantics.
North Holland, Amsterdam, 1981.

[Bauer and Wössner 81]
Bauer, F.L. and Wössner, H.
Algorithmic language and program development.
Berlin, Heidelberg, New York: Springer, 1981.

[Bauer et al. 77] Bauer, F.L., Partsch, H., Pepper, P. and Wössner, H.
Notes on the project CIP: outline of a transformation system.
Technical Report TUM-INFO-7729, Institut für Informatik, der Technischen Universität München, 1977.

[Bauer et al. 78] Bauer, F., Broy, M., Partsch, H., Pepper, P. and Wössner, H.
Systematics of transformation rules.
Technical Report TUM-INT-BER-77-12-0350, Institut für Informatik, der Technischen Universität München, 1978.

[Bird 80] Bird, R.S.
Tabulation techniques for recursive programs.
ACM Comp. Surv. 12(4):403-418, December, 1980.

[Boudol 83] Boudol, G.
Computational Semantics of Term Rewriting Systems.
Technical Report 192, INRIA Rocquencourt Le Chesnay (France), 1983.

[Boyer and Moore 75]
Boyer, R.S. and Moore J S.
Proving theorems about LISP functions.
Journal of the ACM 22(1):129-144, 1975.

[Broy and Krieg-Brückner 80]
Broy, M. and Krieg-Brückner, B.
Derivation of invariant assertions during program development by transformation.
ACM Trans. Progr. Lang. Syst. 2(3):321-337, 1980.

[Burstall 69] Burstall, R. M.
Proving properties of programs by structural induction.
Computing Journal 12(1):41-48, 1969.

[Burstall 77] Burstall, R. M.
Design considerations for a functional programming language.
In Infotech State of the Art Conference. The Software Revolution, pages 45-57. Infotech, Copenhagen, 1977.

[Burstall and Darlington 77]
Burstall, R. M. and Darlington, J.
A transformation system for developing recursive programs.
Journal of the ACM 24(1):44-67, 1977.

[Burstall and Feather 77]
Burstall, R. M. and Feather, M.
Program development by transformation: an overview.
In Amirchahy, M. and Neel, D. (editors), Les Fondements de la Programmation, pages 45-55. IRIA-SEFI, Toulouse, France, 1977.

[Burstall and Lampson 84]
Burstall, R. M. and Lampson, B.
A Kernel Language for Abstract Data Types and Modules.
In Intern. Symp. Semantics of Data Types. INRIA, 1984.

[Burstall, Collins and Popplestone 71]
Burstall, R. M., Collins, J. S. and Popplestone, R. J.
Programming in Pop-2.
Edinburgh University Press 22 George Sq. Edinburgh, 1971.

[Burstall, MacQueen and Sannella 80]
Burstall, R.M., MacQueen, D.B. and Sannella, D.T.
HOPE: An experimental applicative language.
In Proc. 1980 LISP Conference, pages 136–143. Stanford, Stanford, California, 1980.

[Byrd et al. 80] Byrd, L., Pereira, F. and Warren, D.
Guide to Version 3 of DEC-10 Prolog and Prolog Debugging Facilities.
Technical Report DAI Occasional Paper 19, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

[Cardelli 83] Cardelli, L.
The Functional Abstract Machine.
Technical Report TR-107, Bell Labs Technical Memorandum TM-83-11271-1, 1983.

[Chandra 73] Chandra, A.K.
Efficient Compilation of Linear Recursive Programs.
In 14th Annual Symp. on Switching and Automata Theory, pages 16–25. IEEE, Iowa City, Iowa , October, 1973.

[Chatelin 77] Chatelin, P.
Self-redefinition as a program manipulation strategy. Proc. Symp. Artif. Intellig. Progr. Lang.
ACM SIGPLAN Notices and SIGART Newsletter : 174–179, 1977.

[Clocksin and Mellish 81]
Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, Berlin,Heidelberg,New York, 1981.

[Cohen 78] Cohen, W.A.
Basic Techniques of Combinatorial Theory.
J. Wiley, 1978.

[Cohen 80] Cohen, N.H.
Source-to-source improvement of recursive programs.
PhD thesis, Harvard University, 1980.
TR. 13-80, Aiken Computation Lab., Cambridge, MA.

[Cohen 83] Cohen, N. H.
Eliminating Redundant Recursive Calls.
ACM Transactions on Programming Languages and Systems 5:265–299, 1983.

[Colmerauer et al. 72]
Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P.
Un système de communication homme–machine en francais.
Technical Report, Groupe d'Intelligence Artificielle, U.E.R. de Luminy Universitè d'Aix–Marseille, Luminy, 1972.

[Dahl et al. 72] Dahl, O–J., Dijkstra, E.W. and Hoare, C.A.R.
Structured Programming.
London: Academic Press, 1972.

[Darlington 78] Darlington, J.
A synthesis of several sorting algorithms.
Acta Informatica 11:1–30, 1978.

[Darlington 81] Darlington, J.
The Structured Description of Algorithm Derivations.
In de Bakker, J.W. and van Vliet, H. (editors), Algorithmic Languages, pages 221–250. Elsevier North–Holland, 1981.

[Darlington and Burstall 76]
Darlington, J. and Burstall, R.M.
A system which automatically improves programs.
Acta Informat. 6:41–60, 1976.

[Darlington and Reeve 81]
Darlington, J. and Reeve, M.
ALICE: A Multi–Processor Reduction Machine for the Parallel Evalution of applicative Languages.
In Proc. 1981 Conference on Functional Programming Languages and Computer Architecture, pages 65–75. ACM, Portsmouth, New Hampshire, October 18–22, 1981.

[Darlington, Henderson and Turner 82]
Darlington, J., Henderson, P. and Turner, D.A. (eds.).
Functional Programming and its Applications: an Advanced Course.
Cambridge University Press, 1982.

[Dennis 74] Dennis, J.B.
First Version of a Data Flow Procedure Language.
In Programming Symposium. Lecture Notes in Computer Science n.19. Springer Verlag, Paris France, 1974.

[Dershowitz 83] Dershowitz, N.
Computing with rewrite systems.
Technical Report Aerospace no.ATR83 (8478)-1, The Aerospace Corporation El Segundo California 90245, 1983.

[Dijkstra 82] Dijkstra, E.W.
Selected Writings on Computing: A Personal Perspective.
Springer-Verlag, 1982.

[Er 82] Er, M.C.
An Iterative Solution to the Generalized Towers of Hanoi Problem.
BIT 23:295-302, 1982.

[Feather 78] Feather, M.S.
ZAP program transformation system: Primer and user manual.
Technical Report DAI Research Report No. 54, Dept. of Artificial Intelligence, University of Edinburgh, 1978.

[Feather 79] Feather, M.S.
A system for developing programs by transformation.
PhD thesis, University of Edinburgh, 1979.

[Feather 82] Feather, M.S.
A system for assisting program transformation.
ACM Trans. Progr. Lang. Syst. 4(1):1-20, 1982.

[Floyd 79] Floyd, R.W.
The paradigms of programming.
Communications of the ACM 22(8):455-460, 1979.

[Gordon 75] Gordon, M.
Towards a Semantic Theory of Dynamic Binding.
Technical Report STAN-CS-75-507, Computer Science Dept. Stanford University, 1975.

[Gordon, Milner and Wadsworth 79]
Gordon, M., Milner, R. and Wadsworth, C.
Edinburgh LCF. Lecture Notes in Computer Science n. 78.
Springer-Verlag, 1979.

[Gries and Levin 80]
Gries, D. and Levin, G.
Computing Fibonacci Numbers (and similarly defined functions) in log time.
Infomation Processing Letters 10:68-75, 1980.

[Hayes 77] Hayes, P.J.
A Note on the Towers of Hanoi Problem.
The Computer Journal 20(3):282-285, 1977.

[Henderson 80] Henderson, P.
Functional Programming, Applications and Implementation.
Prentice-Hall International Englewood Cliffs, N.J., 1980.

[Hennessy and Wei 82]
Hennessy, M.C.B. and Wei, Li.
Translating a subset of ADA into CCS.
In D. Bjőrner (editor), Proc. Formal Description of Programming Concepts II. IFIP TC-2, Garmisch-Partenkirchen, 1982.

[Hewitt and Baker 78]
Hewitt, C. and Baker, H. Jr.
Actors and Continuous Functionals.
In Neuhold, E.J. (editor), Formal Descriptions of Programming Languages. North Holland, Amsterdam, 1978.

[Hilden 76] Hilden, J.
Elimination of Recursive Calls using a Small Table of 'randomly' Selected Function Values.
BIT 16(1):60-73, 1976.

[Hoare 78] Hoare, C.A.R.
Communicating Sequential Processes.
Communications ACM 21(8):666-677, 1978.

[Hoffmann and O'Donnell 82]
Hoffmann, C.M. and O'Donnell, M.
Programming with Equations.
ACM Transactions on Programming Languages and Systems 4(1):83-112, 1982.

[Hoggatt 69] Hoggatt, V. E. Jr.
Fibonacci and Lucas Numbers.
Boston, Houghton Mifflin Co., 1969.

[Hopcroft and Ullman 79]
Hopcroft, J.E. and Ullman, J.D.
Introduction to Automata Theory Languages and Computation.
Addison-Wesley Publ. Co., 1979.

[Huet 80] Huet, G.
Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems.
Journal of A.C.M. 27(4):797-821, October, 1980.

[Huet and Lèvy 79]
Huet, G. and Lèvy, J.J.
Call-by-need Computations in Non-ambiguous Linear Term Rewriting Systems.
Technical Report 359, INRIA Rocquencourt Le Chesnay (France), 1979.

[Huet and Lang 78]
Huet, G. and Lang, B.
Proving and applying program transformations expressed with second-order patterns.
Acta Informat. 11:31-55, 1978.

[Huet and Oppen 80]
Huet, G. and Oppen, D.
Equations and Rewrite Rules: A Survey.
In Book, R. (editor), Formal Language Theory: Perspectives and Open Problems. Academic Press, 1980.

[Jerrum and Snir 82]
Jerrum, M. and Snir, M.
Some exact complexity results for straight-line computations on semirings.
Journal of the ACM 29(3):874-897, July, 1982.

[Kahn and MacQueen 77]
Kahn, G. and MacQueen, D.B.
Coroutines and Network of Parallel Processes.
In Gilchrist, B. (editor), Proceedings IFIP 77, pages 993-998. North Holland Amsterdam, 1977.

[Kleene 67] Kleene, S.C.
Introduction to Metamathematics.
North-Holland Publ. Co., 1967.

[Kott 78] Kott, L.
About transformation system: A theoretical study.
In 3ème Colloque International sur la Programmation, pages 232-247. Dunod, Paris, 1978.

[Kowalski 74] Kowalski R. A.
Predicate Logic as a Programming Language.
In Proceedings IFIP 74, pages 569-574. IFIP, North Holland Publishing Co., Amsterdam, 1974.

[Landin 64] Landin, P.J.
The Mechanical Evaluation of Expressions.
Computer Journal 6(4):308-320, 1964.

[Landin 66] Landin, P.J.
A lambda-calculus approach.
In L. Fox (editor), Advances in Programming and Non-Numerical Computation, pages 97-141. Pergamon Press, 1966.

[Lauer, Torrigiani and Shields 79]
Lauer, P.E., Torrigiani, P.R. and Shields, M.W.
COSY: A System Specification Language Based on Paths and Processes.
Acta Informatica 12: 109-158, 1979.

[Lévy 80] Lévy, J.J.
Optimal reduction in the lambda-calculus.
In J.P. Seldin and J.R. Hindley (editors), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, 1980.

[Liu 68] Liu, C.L.
Introduction to Combinatorial Mathematics.
McGraw-Hill, 1968.

[Manna and Waldinger 79]
Manna, Z. and Waldinger, R.
Synthesis: Dreams => Programs.
IEEE Transactions on Software Engineering SE-5(4), July 1979.

[McCarthy et al. 62]
McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I.
The Lisp 1.5 Programmer's Manual
MIT Press, Cambridge, Mass., 1962.

[Michie 68] Michie, D.
'Memo' functions and machine learning.
Nature 218(5136): 19-22, April, 1968.

[Miller and Brown 66]
Miller, J. and Brown, S.
An algorithm for evaluation of remote terms in a linear recurrence sequence.
The Computer Journal 9: 188-190, 1966.

[Milner 80] Milner, R.
A Calculus for Communicating Systems.
Springer Verlag Lecture Notes in Computer Science n. 92, 1980.

[Monk 76] Monk, J. D.
Mathematical Logic. Graduate Texts in Mathematics.
Springer-Verlag, 1976.

[Mycroft 81] Mycroft, A.
Abstract interpretation and optimising transformations for applicative programs.
PhD thesis, Computer Science Dept., University of Edinburgh, 1981.

[O'Donnell 77] O'Donnell, M. J.
Computing in Systems Described by Equations.
Springer Verlag Lecture Notes in Computer Science n. 58, 1977.

[Partsch and Pepper 77]
Partsch, H. and Pepper, P.
Program transformations on different levels of programming.
Technical Report TUM-INFO-7715, Institut für Informatik, der Technischen Universität München, 1977.

[Partsch and Steinbrüggen 81]
Partsch, H. and Steinbrüggen, R.
A comprehensive survey on program transformation systems.
Technical Report TUM I8108, Institut für Informatik, der Technischen Universität München, 1981.

[Partsch and Steinbrüggen 83]
Partsch, H. and Steinbrüggen, R.
Program Transformation Systems.
ACM Computing Surveys 15(3): 199-236, 1983.

[Paterson and Hewitt 70]
Paterson, M.S. and Hewitt, C.E.
Comparative Schematology.
In Conference on Concurrent Systems and Parallel Computation, pages 119-127. Project MAC, Wood's Hole, Mass., June 2-5, 1970.

[Pettorossi 77] Pettorossi, A.
Transformation of programs and use of 'tupling strategy'.
In Proc. of Informatica '77 Conference, pages 1-6. Informatica '77, Bled, Yugoslavia, 1977.

[Pettorossi 78] Pettorossi, A.
Improving memory utilization in transforming programs.
In MFCS, Zakopane, Poland. Lecture Notes in Computer Science n. 64, pages 416-425. Berlin-Heidelberg- New York: Springer, 1978.

[Pettorossi 80a] Pettorossi, A.
Derivation of an $O(k^2 \log n)$ algorithm for computing order-k Fibonacci numbers from the $O(k^3 \log n)$ matrix multiplication method.
Info. Proc. Lett. 11(4,5):172-179, 1980.

[Pettorossi 80b] Pettorossi, A.
Towards a Theory of Parallelism and Communications for Increasing Efficiency in Applicative Languages.
In Salwicki, A. (editor), Proceedings Logic of Programs and Their Applications, pages 224-249. Lecture Notes in Computer Science n. 148 Springer Verlag, Poznań Poland, 1980.

[Pettorossi 81a] Pettorossi, A.
An Approach to Communications and Parallelism in Applicative Languages.
In Diaz, J. and Ramos, I. (editors), International Colloquium on Formalization of Programming Concepts, pages 432-446. Lecture Notes in Computer Science n. 107 Springer Verlag, Peñíscola Spain, 1981.

[Pettorossi 81b] Pettorossi, A.
A Transformational Approach for Developing Parallel Programs.
In Händler, W. (editor), CONPAR 81, pages 245–258. Lecture Notes in Computer Science n. 111 Springer Verlag, 1981.

[Pettorossi 84] Pettorossi, A.
A Powerful Strategy for Deriving Very Efficient Programs by Transformation.
In Symp. on LISP and Functional Programming. Austin, Texas. ACM, Aug. 1984.

[Pettorossi and Burstall 82]
Pettorossi, A. and Burstall, R. M.
Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique.
Acta Informatica 18: 181–206, 1982.

[Pettorossi and Skowron 82a]
Pettorossi, A. and Skowron, A.
Efficient Execution of Parallel Programs and Proofs of Their Correctness Using Computing Agents and Communications.
In Les Mathématiques de l'Infomatique. Symposium AFCET, Paris France, 1982.

[Pettorossi and Skowron 82b]
Pettorossi, A. and Skowron, A.
Communicating agents for applicative concurrent programming.
In Dezani–Ciancaglini and Montanari (editors), 5th International Symposium on Programming. Lecture Notes in Computer Science n. 137, pages 305–322. Springer-Verlag, Turin Italy, 1982.

[Pettorossi and Skowron 83]
Pettorossi, A. and Skowron, A.
Complete Modal Theories for Verifying Communicating Agents Behaviour in Recursive Equations Programs.
Technical Report CSR-128-83, Computer Science Department Edinburgh University, 1983.

[Plotkin 81] Plotkin, G. D.
A structural approach to operational semantics.
Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

[Plotkin 82] Plotkin, G. D.
An Operational Semantics for CSP.
In D. Bjørner (editor), Proc. Formal Description of Programming Concepts II, pages 199-223. IFIP TC-2, Garmisch-Partenkirchen, 1982.

[Reif and Scherlis 82]
Reif, J. H. and Scherlis, W. L.
Deriving Efficient Graph Algorithms.
Technical Report CMU-CS-82-155, Carnegie Mellon University, Dec. 1982.

[Rosenbloom 50] Rosenbloom, P.
The Elements of Mathematical Logic.
Dover, 1950.

[Roussel 75] Roussel, P.
PROLOG: Manuel de Reference et d'Utilisation
Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, France, 1975.

[Scherlis 80] Scherlis, W. L.
Expression Procedures and Program Derivation.
PhD thesis, Stanford University, August 1980.
Computer Science Report STAN-CS-80-818.

[Schwarz 78] Schwarz, J.
Verifying the safe use of destructive opertions in applicative programs.
In 3ème Colloque International sur la Programmation, pages 394–410. Dunod, Paris, 1978.

[Schwarz 82] Schwarz, J.
Using Annotations to Make Recursive Equations Behave.
IEEE Transactions on Software Engineering SE-8(1), January, 1982.

[Shoenfield 67] Shoenfield, J. R.
Mathematical Logic.
Addison-Wesley, 1967.

[Shortt 78] Shortt, J.
An iterative algorithm to calculate Fibonacci numbers in O(log n) arithmetic operations.
Information Processing Letters 7:299–303, 1978.

[Steinbrüggen 77]Steinbrüggen, R.
Equivalent recursive definitions of certain number theoretical functions.
Technical Report TUM-INFO-7714, Institut für Informatik, der Technischen Universität München, 1977.

[Strong 71] Strong, H. R.
Translating Recursion Equations into Flowcharts.
Journal of Computer and System Science 5(3):254–285, June, 1971.

[Strong, Maggiolo-Schettini and Rosen 75]
Strong, H. R., Maggiolo-Schettini, A., and Rosen, B. K.
Recursion Structure Simplification.
SIAM J. Computing 4(3):307–320, September, 1975.

[Swamy and Savage 79]
Swamy, S. and Savage, J. E.
Space-time tradeoffs for linear recursion.
In 6th Annual Symp. on Principles of Programming Languages, pages 135-142. ACM, San Antonio, Texas, Jan 29-31, 1979.

[Turner 76] Turner, D. A.
SASL Language Manual.
Technical Report, St. Andrews University Technical Report, Scotland, 1976.

[Turner 81a] Turner, D. A.
KRC Language Manual.
Technical Report, University of Kent at Canterbury, Department of Computer Science, 1981.

[Turner 81b] Turner, D. A.
The Semantic Elegance of Applicative Languages.
In Conference on Functional Programming Languages and Computer Architecture, pages 85-92. ACM, Portsmouth, New Hampshire, Oct. 18-22, 1981.

[Urbanek 80] Urbanek, F. J.
An O(log n) algorithm for computing the nth element of the solution of a difference equation.
Info. Proc. Lett. 11(2):66-67, 1980.

[Wadler 81] Wadler, P.
Applicative Style Programming, Program Transformation, and List Operators.
In Conference on Functional Programming Languages and Computer Architecture, pages 25-32. ACM, Portsmouth, New Hampshire, Oct. 18-22, 1981.

[Walker and Strong 72]
Walker, S. A. and Strong, H. R.
Characterizations of flowchartable recursions.
In Proc. 4th ACM Conference on Theory of Computing, pages 18-34. ACM, Denver, Colorado, 1972.

[Walker and Strong 73]
Walker, S. A. and Strong, H. R.
Characterizations of Flowchartable Recursions.
Journal of Computer and System Sciences 7(4):404-447, August, 1973.

[Wand 80] Wand, M.
Continuation-based program transformation strategies.
Journal of the ACM 27(1):164-180, 1980.

[Warren 77] Warren, D. H. D.
Implementing Prolog.
Technical Report N. 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[Wegbreit 76] Wegbreit, B.
Goal-directed program transformation.
IEEE Trans. Software Eng. SE-2:69-79, 1976.

[Wei 82a] Wei, Li.
An operational semantics of tasking and exception handling in Ada.
Technical Report CSR-99-82, Computer Science Department, Edinburgh University, 1982.

[Wei 82b] Wei, Li.
An operational approach to semantics and translation for concurrent programming languages.
PhD thesis, University of Edinburgh, 1982.

[Wilson and Shortt 80]
Wilson, T. C. and Shortt, J.
An O(log n) algorithm for computing general order-k Fibonacci numbers.
Info. Proc. Lett. 10(2):68-75, 1980.

[Wirth 71] Wirth, N.
Program development by stepwise refinement.
Communications of the ACM 14(4):221-227, 1971.