A Historical Account of My Early Research Interests [*]

Alberto Pettorossi
DICII, University of Rome Tor Vergata, Rome, Italy
IASI-CNR, Rome, Italy
adp@iasi.cnr.it

This paper presents a brief account of some of the my early research interests. This historical account starts from my laurea thesis on Signal Theory and my master thesis on Computation Theory. It recalls some results in Combinatory Logic and Term Rewriting Systems. Some other results concern Program Transformation, Parallel Computation, Theory of Concurrency, and Proof of Program Properties. My early research activity has been mainly done in cooperation with Andrzej Skowron, Anna Labella, and Maurizio Proietti.

1 From Signal Theory to Combinatory Logic

Since my childhood I very much liked Arithmetic and Mathematics. The formal reasoning always attracted my spirit and I always felt a special interest for numbers and geometrical patterns. Maybe this was due to the fact that I thought that Mathematics is a way of establishing 'truth beyond any doubt'. As Plato says: 'Truth becomes manifest in the mathematical process' (Phaedo). (The actual word used by Plato for 'mathematical process' comes from $\lambda o\gamma i\zeta o\mu\alpha\iota$ which means: I compute, I deduce.) During my high school I attended the Classical Lyceum. Perhaps, for me the Scientific Lyceum would have been a better school to attend, but the Scientific Lyceum was located too far away from my home town.

At the age of nineteen, I began my university studies in Rome as a student of Engineering. I was in doubt whether or not to enrol myself as a Mathematics student, but eventually I followed my father's suggestion to study Engineering because, as he said: "If you study Mathematics, you will have no other choice in life than to become a teacher." My thesis work was in Telecommunication and, in particular, I studied the problem of how to pre-distort an electric signal which encodes a sequence of symbols, each one being 0 or 1, via a sequence of impulses. The pre-distortion of the electric signal should minimize the effect of a Gaussian white noise (which would require a reduction of bandwidth) and the interference between symbols (which would require an increase of bandwidth). A theoretical solution to this problem is not easy to find. Thus, I was suggested to look for a practical solution via a numerical simulation of the transmission channel and the construction of the so called *eye pattern* [45]. In the numerical simulation, which uses the Fast Fourier Transform algorithm, one could easily modify the various parameters of the pre-distortion for minimizing the errors in the output sequence of 0's and 1's. The thesis work was done under the patient guidance of my supervisors Professors Bruno Peroni and Paolo Mandarini.

After getting the laurea degree, I attended during 1972 at Rome University a course in Engineering of Control and Computation Systems. During that year I read the book entitled

[*]This work has been partially supported by GNCS-INdAM, Italy.

Mathematical Theory of Computation written by Professor Zohar Manna (1939–2018) (at that time that book was nothing more than a thick technical report of Stanford University, California). I wrote my master thesis on the "Automatic Derivation of Control Flow Graphs of Fortran Programs", under the guidance of Professor Vincenzo Falzone and Professor Paolo Ercoli [46]. In particular, I wrote a Fortran program which derives control flow graphs of Fortran programs. That program ran on a UNIVAC 1108 computer with the EXEC 8 Operating System. The main memory had 128k words. The program I wrote was a bit naive, but at that time I was not familiar with efficient parsing techniques. I also studied various kinds of program schemas and, in particular, those introduced by Lavrov [37], Yanov [74], and Martynuk [39]. Having constructed the control flow graph of a given Fortran program, one could transform that program into an equivalent one with better computational properties (such as smaller time or space complexities) by applying a set of schema transformations [32] which are guaranteed to preserve semantical equivalence. Schema transformations are part of the research area in which I have been interested for some years afterwards.

During that period, which overlapped with my military service in the Italian Air Force, I also read a book on Combinatory Logic (actually, not the entire book) by J. R. Hindley, B. Lercher and J. P. Seldin [27, 28]. I read the Italian edition of the book, which was emended of some inaccuracies with respect to the previous English edition (as Roger Hindley himself told me later). Under the guidance of Professor Giorgio Ausiello and the great help of my colleague Carlo Batini, I studied various properties of subbases in Weak Combinatory Logic (WCL) [3].

WCL is an applicative system whose terms, called *combinators*, can be defined as follows: (i) K and S are atomic terms, and (ii) if t_1 and t_2 are terms, then $(t_1\ t_2)$ is a term. When parentheses are missing, left associativity is assumed. A notion of *reduction*, denoted $>$, is introduced as follows: for all terms x, y, z, $Sxyz > xz(yz)$ and $Kxy > x$. Thus, for instance, $SKKS > KS(KS) > S$. WCL is a Turing complete system as every partial recursive function can be represented as a combinator in WCL. A *subbase* in WCL is a set of terms which can be constructed starting a fixed set of (possibly non-atomic) combinators. For instance, the subbase $\{B\}$, where B is a combinator defined by the following reduction: $Bxyz > x(yz)$, is made out of all terms which are constructed by B's (and parentheses) only. These terms are called B-combinators. One can show that B can be expressed in the subbase $\{S, K\}$ by $S(KS)K$. Indeed, $S(KS)Kxyz >^* x(yz)$, where $>^*$ denotes the reflexive, transitive closure of $>$. The various subbases provide a way of partitioning the set of computable functions into various sets, according to the features of the combinators in the subbases. This should be contrasted with other stratifications of the set of computable functions one could define and, among them, the stratifications based on complexity classes or on the Chomsky hierarchy [30] with the *type* i (for $i = 0, 1, 2, 3$) classes of languages.

Among other subbases, we studied the subbase $\{B\}$ and we showed how to construct the shortest B-combinator for constructing bracketed terms out of sequences of atomic subterms. For instance, $B(B(BB)B)(BB)$ is the shortest B-combinator X such that: $Xx_1x_2x_3x_4x_5x_6 >^* x_1(x_2(x_3x_4))(x_5x_6)$.

During 1975, while attending in Rome the conference on λ-calculus and Computer Science Theory, where our results on subbases were presented [3], I heard from Professor Henk Barendregt of an open problem concerning the existence of a combinator $\widetilde{X}$ made of only S's (and parentheses), having no weak normal form. A combinator T is said to be *in weak normal form* if no combinator T' exists such that $T > T'$. X is said to have *weak normal form* if there exists a combinator T such that $X >^* T$ and T is in weak normal form.

It was not hard to show that one such combinator $\widetilde{X}$ is $SAA(SAA)$, where A denotes $((SS)S)$. I send the result to Henk Barendregt (by surface mail, of course). Some years later I was happy to see that an exercise about that problem and its solution was included in Barendregt's book on λ-calculus [2, page 162].

2 Finite and Infinite Computations

While studying Combinatory Logic, I became interested in terms viewed as trees and tree transformers. Indeed, combinators can be considered both as trees and tree transformers at the same time. This area was also related to the research on Term Rewriting Systems which was going to be one of my interests for a few years later. The search for a non-terminating combinator stimulated my studies on infinite, non-terminating computations.

In 1979 I introduced a hierarchy of infinite computations within WCL (and other Turing complete systems) which is related to the Chomsky hierarchy of languages [48]. That definition uses the notion of a *sampling function* s which is a total function from the set of natural numbers to $\{true, false\}$, which from an infinite sequence $\sigma = \langle w_0, w_1, w_2, \ldots \rangle$ of finite words constructed by an infinite computation, selects an infinite subsequence σ_s whose words are the elements of a (finite or infinite) language L_s. We state that $L_s =_{def} \{w_j \mid j \geq 0 \wedge w_j \textit{ occurs in } \sigma \wedge s(j) = \textit{true}\}$. Let us assume that L_s is generated by a grammar G_s. In this case we say that also the subsequence σ_s is generated by the grammar G_s. Given a sequence σ, by varying the sampling function s we have different languages L_s and different generating grammars G_s. For $i = 0, 1, 2, 3$, we say that the infinite computation which generates σ *is of type* i if there exists a sampling function s selecting a subsequence σ_s generated by a grammar of type i, and no sampling function s' exists such that the subsequence selected by s' is generated by a grammar of type $(i+1)$.

For instance, let us consider the following program P:

$w = $"$a$"; `while` *true* `do` *print* w; $w = $"$b$"$w$"$c$"; π_0 `od`

where a, b, c are characters, w is a string of characters, and π_0 is a terminating program fragment associated with a type 0 language L_0, such that: (i) L_0 is not of type 1, (ii) π_0 does not modify w, (iii) at each loop body execution, π_0 prints only one word of L_0, and (iv) for every word $v \in L_0$ there is exactly one body execution in which π_0 prints v. We have that P evokes an infinite computation of type 2, as the grammar with axiom S and productions: $S \rightarrow a \mid b\,S\,c$ is a type 2 (context free) grammar.

When I first presented this hierarchy definition at a conference, I met my dear colleague Philippe Flajolet (1948-2011) and he said to me: "I have already studied these topics [24]. You should look at the immune sets." That remark motivated my first encounter with Roger's book on recursivity [66] where immune sets are defined and analyzed. Then also Professor Maurice Nivat (1937-2017) came to me and said: "It is a nice piece of work,... but you should rewrite the paper in a better way!". I was very glad that Nivat showed interest in my work. He was right in asking me to rewrite it and improve it. Unfortunately, I did not follow his suggestion. Not even when, a few years later, Professor Tony Hoare told me: "I like writing and rewriting my papers."

Looking for terms with infinite behaviour in WCL, in 1980 I wrote a paper on the automatic construction of combinators having no normal form by using the so called *accumulation* method and the *pattern matching and hereditary embedding* method [49]. The solutions of some equations between terms would guarantee the existence of the combinators with the desired properties.

On the other side of the camp, that is, considering the finite behaviours, many people at that time were studying properties of Term Rewriting Systems (TRSs) which would guarantee termination. Among them, Nachum Dershowitz, Samuel Kamin, Jean-Jacques Lévy, and David Plaisted. In 1981 I wrote a paper introducing the *non-ascending property* [50]. In that paper I related the various techniques which were proposed, including recursive path orderings, simplification ordering, and bounded lexicographic orderings. I thank Nachum for pointing out to me some errors in that paper and, in particular, a missing left-linearity hypothesis about the TRS under consideration [20]. A TRS is said to be *left linear* if the variable occurrences on the left hand side of every rule are all distinct. For instance, $f(x,y,z) \to f(y,z,x)$ is a left linear rule, while $f(x,y,x) \to g(y,x)$ is not. During a conference coffee-break, Jean-Jacques showed me a simple inductive proof of Fact 1 [50, pages 436–437] using bounded lexicographic orderings (actually, that proof is based on a non-predicative definition of the non-ascending rewriting rules).

3 Program Transformation

During the years 1977–1981 I visited Edinburgh University. I was supported by the British Council organization and the Italian National Research Council. I did my Ph.D. thesis work on program transformation under the guidance of Professor Rod Burstall and also Professor Robin Milner, during Rod's visit to Inria in Paris for some months. I met Rod in person for the first time at the Artificial Intelligence Department, in Hope Park Square at Edinburgh. I addressed him by saying: "Professor Burstall,...". I do not remember my subsequent words, but I do remember what he said to me in answering: "Alberto, this is the last time you call me 'professor'. Please, call me Rod." He introduced me to functional programming and he wrote 'for me', as he said, a compiler for a new functional language, called NPL [9] he was developing at that time. The language NPL later evolved into Hope [11]. While at Edinburgh, I wrote a paper [47] on the automatic annotation of functional programs for improving memory utilization. Functions could destroy the value of their arguments whenever they were no longer needed for subsequent computations. I apologize for not having Rod as co-author of that paper.

My Ph.D. thesis work was mainly on program transformation starting from the seminal paper by Rod and John Darlington [10]. Some time before, Rod had received a letter from Professor Edger W. Dijkstra (1930-2002) proposing the following 'exercise' in program transformation: the derivation of an iterative program for the `fusc` function [22, pages 215–216, 230–232]:

$\mathtt{fusc(0)} = \mathtt{0} \qquad \mathtt{fusc(1)} = \mathtt{1}$

$\mathtt{fusc(2n)} = \mathtt{fusc(n)} \qquad \mathtt{fusc(2n{+}1)} = \mathtt{fusc(n{+}1)} + \mathtt{fusc(n)} \qquad \text{for } \mathtt{n} \geq \mathtt{0}$

In one of my scientific conversations with Rod, he told me about his research interests and he also mentioned the above exercise. The difficult part of the exercise was how to motivate the 'invention' of the new function definitions to be introduced during program transformation in the so called *eureka steps* [10].

To do the same exercise Bauer and Wössner [4, page 288] use an embedding into a linear combination, that is, they define the function $\mathtt{F(n,a,b)} =_{def} \mathtt{a}\times\mathtt{fusc(n)} + \mathtt{b}\times\mathtt{fusc(n{+}1)}$. Using that function, they are able to derive for `fusc` a program that is linear recursive and also tail-recursive. Then, from that program they easily derive an iterative program. But, where the function F comes from? I wanted to do the exercise using the unfolding/folding rules only [10] and, at the same time, I wanted to give a somewhat mechanizable account of the definition the new functions to be introduced.

Now, the unfolding rule allows one to unroll (upto a specified depth) the recursive calls thereby generating a directed acyclic graph of distinct calls. I called that graph the *m-dag*. The prefix *m* (short for minimal) tells us that in an *m-dag* identical function calls are denoted by a single node. Then, I used the so called *tupling strategy* that allows one to define new functions as the result of tupling together function calls which share common subcalls, that is, calls which have common descendants in the m-dag. Note that to check this sharing property requires syntactic operations only on the m-dags. By using the tupling strategy, looking at the m-dag for fusc, we introduce the tuple function $\mathtt{t(n)} =_{\mathtt{def}} \langle \mathtt{fusc(n)},\ \mathtt{fusc(n{+}1)}\rangle$ and we get the following recursive equations for fusc:

$$
\begin{array}{l}
\mathtt{fusc(n)} = \mathtt{u} \qquad \mathtt{where}\ \langle \mathtt{u,v}\rangle = \mathtt{t(n)} \quad \text{for } \mathtt{n} \geq 0\\
\mathtt{t(0)} = \{\text{by unfolding}\} = \langle \mathtt{fusc(0)},\ \mathtt{fusc(1)}\rangle = \{\text{by unfolding}\} = \langle 0,1\rangle\\
\mathtt{t(2n)} = \{\text{by unfolding}\} = \langle \mathtt{fusc(2n)},\ \mathtt{fusc(2n{+}1)}\rangle = \{\text{by unfolding}\} =\\
\qquad = \langle \mathtt{fusc(n)},\ \mathtt{fusc(n{+}1){+}fusc(n)}\rangle = \{\text{by } \mathtt{where} \text{ abstraction [10]}\} =\\
\qquad = \langle \mathtt{u,u{+}v}\rangle \qquad \mathtt{where} \langle \mathtt{u,v}\rangle = \langle \mathtt{fusc(n)},\ \mathtt{fusc(n{+}1)}\rangle = \{\text{by folding}\} =\\
\qquad = \langle \mathtt{u,u{+}v}\rangle \qquad \mathtt{where}\ \langle \mathtt{u,v}\rangle = \mathtt{t(n)} \quad \text{for } \mathtt{n} > 0\\
\mathtt{t(2n{+}1)} = \langle \mathtt{u{+}v,v}\rangle \quad \mathtt{where}\ \langle \mathtt{u,v}\rangle = \mathtt{t(n)} \quad \text{for } \mathtt{n} \geq 0 \qquad \text{(by a derivation similar to that of } \mathtt{t(2n)}\text{)}
\end{array}
$$

Now a last step is needed to get the iterative program desired by Dijkstra's exercise.

I used the following schema equivalence (such as the ones in [69]) stating that t(n) defined by the non-tail recursive equations:

$$
\begin{array}{ll}
\mathtt{t(0)} = \mathtt{a} & \\
\mathtt{t(2n)} = \mathtt{b(t(n))} & \text{for } \mathtt{n} > 0\\
\mathtt{t(2n+1)} = \mathtt{c(t(n))} & \text{for } \mathtt{n} \geq 0
\end{array}
$$

is equal to the value of res returned by the following program, where $\mathtt{B}[\ell..0]$ stores the binary expansion of m, the most significant bit being at position ℓ (obviously, $\mathtt{B}[\ell..0]$ can be computed by performing $O(\log \mathtt{m})$ successive integer divisions by 2):

```
res = a;  p = ℓ;  while p ≥ 0 do if  B[p] = 0  then  res = b(res)  else  res = c(res);  p = p−1  od
```

By using this schema equivalence we derive from the above linear, non-tail recursive program for fusc the following iterative program:

```
        {n ≥ 0  ∧  n = Σ_{p=0}^{ℓ} B[p]·2^p}
⟨u,v⟩ = ⟨0,1⟩;      p = ℓ;
while p ≥ 0 do if  B[p] = 0  then  v = u+v  else  u = u+v;   p = p−1 od
        {⟨u,v⟩ = t(n)  ∧  u = fusc(n)}
```

Note that we do not need to state the somewhat intricate invariant of the while-loop for showing the correctness of the derived iterative program, as Dijkstra's methodology for program construction would have required us to do. The derived program, which is correct by construction, uses an $O(\log \mathtt{n})$ number of operations for computing fusc(n) as Dijkstra's program reported in [22, page 215–216][1]. We have only to show by induction, once and for all, the validity of the schema equivalence we have used.

[1]In order to get exactly Dijkstra's program, one should perform a generalization step as indicated in [52].

Having derived an iterative program for the `fusc` function, I faced the problem of deriving by transformation an iterative program, such as the one suggested by [41], which computes the Fibonacci function $\mathtt{fib(n)}$ using an $O(\log \mathtt{n})$ number of arithmetic operations. Here is the definition of the Fibonacci function:

$$\mathtt{fib(0)=0} \qquad \mathtt{fib(1)=1}$$
$$\mathtt{fib(n+2) = fib(n+1) + fib(n)} \quad \text{for } \mathtt{n \geq 0} \qquad (\dagger 1)$$

By using the tupling strategy the function $\mathtt{g(n)} =_{\mathtt{def}} \langle \mathtt{fib(n)}, \mathtt{fib(n-1)} \rangle$ is introduced and the following program is derived:

$$\mathtt{fib(0)=0} \qquad \mathtt{fib(1)=1}$$
$$\mathtt{fib(n+2)=u \ \ where\ \langle u,v \rangle = g(n+2)} \qquad \text{for } \mathtt{n \geq 0}$$
$$\mathtt{g(1) = \langle 1,0 \rangle}$$
$$\mathtt{g(n+2) = \langle u+v,u \rangle \ \ where\ \langle u,v \rangle = g(n+1)} \qquad \text{for } \mathtt{n \geq 0}$$

The iterative program for `fib` can be obtained by applying the following schema equivalence stating that $\mathtt{g(n)}$ defined by the equations:

$$\mathtt{g(0)=a} \qquad \mathtt{g(n+1)=b(g(n))} \quad \text{for } \mathtt{n \geq 0}$$

is equal to the value of `res` returned by the program:

```
res = a;    while n > 0 do  res = b(res);  n = n−1  od
```

Thus, we get:

```
                {n ≥ 0}
if n = 0 then u = 0 else
if n = 1 then u = 1 else
begin p = n−1; ⟨u,v⟩ = ⟨1,0⟩;  while p > 0 do ⟨u,v⟩ = ⟨u+v,u⟩;  p = p−1 od end
            {u = fib(n)}
```

This program has a linear time complexity, in the sense that it computes the result by a linear number of additions. In order to get a program which requires $O(\log \mathtt{n})$ arithmetic operations when computing $\mathtt{fib(n)}$, we should invent the *multiplication* operation, which is not present in Equation (†1). From that equation by unfolding we have:

$$\begin{aligned}\mathtt{fib(n+2)} &= \mathtt{fib(n+1) + fib(n)} = \{\text{by unfolding } \mathtt{fib(n+1)}\} = \\ &= \mathtt{2 \cdot fib(n) + fib(n-1)} = \{\text{by unfolding } \mathtt{fib(n)}\} = \\ &= \mathtt{3 \cdot fib(n-1) + 2 \cdot fib(n-2)}\end{aligned} \qquad (\dagger 2)$$

The unfolding process may continue for some more steps, but we stop here. We will not discuss here the important issue of how many unfolding steps should be performed when deriving programs by transformation. Let us simply note that more unfoldings may exhibit more patterns of function calls from which more efficient functions can be derived.

In our case the invention of the multiplication operation is reduced to three generalization steps [55]. First, we generalize the initial values `0` and `1` of the function `fib` to two variables $\mathtt{a_0}$ and $\mathtt{a_1}$, respectively. (This kind of generalization step is usually done when mechanically proving theorems about functions [7].) By promoting those new variables to arguments, we get the following new function `G`:

$$\mathtt{G(a_0,a_1,0) = a_0} \qquad \mathtt{G(a_0,a_1,1) = a_1}$$
$$\mathtt{G(a_0,a_1,n+2) = G(a_0,a_1,n+1) + G(a_0,a_1,n)} \qquad \text{for } \mathtt{n \geq 0} \qquad (\dagger 3)$$

This function `G` satisfies the following equation which is derived from Equation (†3), as Equation (†2) has been derived from (†1):

$$\mathtt{G(a_0,a_1,n+2) = 3 \cdot G(a_0,a_1,n-1) + 2 \cdot G(a_0,a_1,n-2)} \qquad (\dagger 4)$$

The second generalization consists in generalizing the coefficients 2 and 3 to two functions p(n) and q(n), respectively (and thus, multiplication is introduced). By this generalization we establish a correspondence between the value of the coefficients and the number of unfoldings performed. We can then derive the explicit definitions of the functions p(n) and q(n) as shown in [55], and we get that $\mathtt{p(n)} = \mathtt{G(1,0,n)}$ and $\mathtt{q(n)} = \mathtt{G(0,1,n)}$.

The third, final generalization consists in generalizing the argument n+2 on the left hand side of Equation (†4) to n+k and promoting the new variable k to an argument of a new function defined as follows: $\mathtt{F}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{n},\mathtt{k}) =_{\mathtt{def}} \mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{n}+\mathtt{k})$.

From the equations defining $\mathtt{F}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{n},\mathtt{k})$ we get (the details are in [55, pages 184–185]):

$$\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{n}{+}\mathtt{k}) = \mathtt{G}(0,1,\mathtt{k})\cdot\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{n}{+}1) + \mathtt{G}(1,0,\mathtt{k})\cdot\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{n})$$

Then, by taking n = k and n = k+1, we also get:

$$\begin{array}{ll} \mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,2\mathtt{k}) = \mathtt{G}(0,1,\mathtt{k})\cdot\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{k}{+}1) + \mathtt{G}(1,0,\mathtt{k})\cdot\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{k}) & \text{for } \mathtt{k}>0 \\ \mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,2\mathtt{k}{+}1) = \mathtt{G}(0,1,\mathtt{k})\cdot\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{k}{+}2) + \mathtt{G}(1,0,\mathtt{k})\cdot\mathtt{G}(\mathtt{a}_0,\mathtt{a}_1,\mathtt{k}{+}1) & \text{for } \mathtt{k}\geq 0 \end{array}$$

Eventually, by tupling together the function calls which share the same subcalls, we get the following program which computes fib(n) by performing an $O(\log \mathtt{n})$ number of arithmetic operations only, as desired. For all $\mathtt{k} \geq 0$, the function r(k) is the pair ⟨G(1,0,k), G(0,1,k)⟩.

```
fib(0) = 0              fib(1) = 1
fib(n+2) = u+v  where ⟨u,v⟩ = r(n+1)   for n ≥ 0
r(0) = ⟨1,0⟩
r(2k) = ⟨u²+v², 2uv+v²⟩            where ⟨u,v⟩ = r(k)   for k > 0
r(2k+1) = ⟨2uv+v², (u+v)²+v²⟩    where ⟨u,v⟩ = r(k)   for k ≥ 0
```

We leave to the reader to derive the iterative program that can be obtained by a simple schema equivalence from this program. One can say that the program we have derived is even better than the program based on 2×2 matrix multiplications [41], because it tuples together two values only, not four, as required by the use of the 2×2 matrices. Note that our derivation of the program does not rely on any knowledge of matrix theory.

In the paper with Rod Burstall [55] there is a generalization of the derivation of the logarithmic time program for fib to the case of any linear recurrence relation over any semiring structure. What remains to be done? One may want to derive a *constant time* program for evaluating any linear recurrence relation over a semiring. This would require the introduction of the *exponentiation* operation. Recall that $\mathtt{fib(n)} = (\mathtt{A}^{\mathtt{n}} - \mathtt{B}^{\mathtt{n}})/\mathtt{sqrt(5)}$, where $\mathtt{A} = (1{+}\mathtt{sqrt(5)})/2$ and $\mathtt{B} = (1{-}\mathtt{sqrt(5)})/2$.

From September 1977 to June 1978, I visited the School of Computer and Information Science at Syracuse University, N.Y., USA. I attended courses taught by Professor Alan Robinson, John Reynolds, Lockwood Morris, and Robert Kowalski (at that time a visiting professor from Imperial College, London, UK). It was a splendid occasion for deepening my knowledge about many aspects of Computer Science from such illustrious teachers.

In Syracuse I had the opportunity of reading more carefully some parts of the book *Automata Theory, Languages, and Computation* by Hopcroft and Ullman [30] and the book *Introduction to Mathematical Logic* by Mendelson [40]. I was exposed by Professor Kowalski for the first time to various topics of Artificial Intelligence and I read the preliminary draft of his beautiful book *Logic for Problem Solving* [35]. I remember the stress put by Kowalski on Keith Clark's *negation as failure* semantics for logic programs [12]. This Computational Logic area was going to become my main research area in the years to come, through my cooperation with Maurizio Proietti in Logic Program Transformation.

4 The Tupling Strategy and the List Introduction Strategy

The results of the use of tupling and generalization during program transformation were presented in a paper of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, USA [52]. While giving a seminar on those results at the University of Warsaw (Poland) Professor Helena Rasiowa[2] who was in the audience, at the end kindly said to me: "Your paper is a collection of examples!". I was not surprised by that remark, but I was happy to have, among the examples, a simple derivation of an iterative program for computing the moves of the Towers of Hanoi problem. That task was considered to be very challenging by some authors (see, for instance, [26, page 285]), and the derivation I proposed is also easily mechanizable.

The following Hanoi function $\mathtt{h(n,A,B,C)}$ computes the shortest sequence of moves in the free monoid $\{\mathtt{AB,BC,CA,BA,CB,AC}\}^*$ to move $\mathtt{n}(\geq 0)$ disks from peg $\mathtt{A}$ to peg $\mathtt{B}$ using peg $\mathtt{C}$ as an extra peg. A move of a disk from peg $\mathtt{X}$ to peg $\mathtt{Y}$ is denoted by $\mathtt{XY}$, for any distinct $\mathtt{X}$, $\mathtt{Y}$ in $\{\mathtt{A,B,C}\}$. Every disk is of a different size and over any disk only smaller disks can be placed. ε denotes the empty sequence of moves, and :: denotes the concatenation of sequences of moves.

$$\mathtt{h(0,A,B,C)} = \varepsilon$$
$$\mathtt{h(n{+}1,A,B,C)} = \mathtt{h(n,A,C,B)} :: \mathtt{AB} :: \mathtt{h(n,C,B,A)} \quad \text{for } \mathtt{n} \geq 0 \qquad (\dagger 5)$$

In order to get an iterative program for computing $\mathtt{h(n,A,B,C)}$, we first unfold $\mathtt{h(n,A,C,B)}$ and $\mathtt{h(n,C,B,A)}$ in (†5) and then we tuple together in the new function $\mathtt{t(n{-}1)}$ the calls of $\mathtt{h(n{-}1,A,B,C)}$, $\mathtt{h(n{-}1,B,C,A)}$, and $\mathtt{h(n{-}1,C,A,B)}$ which share common subcalls (see Figure 1). The order of the components in the tuple is insignificant. Details are in [53].

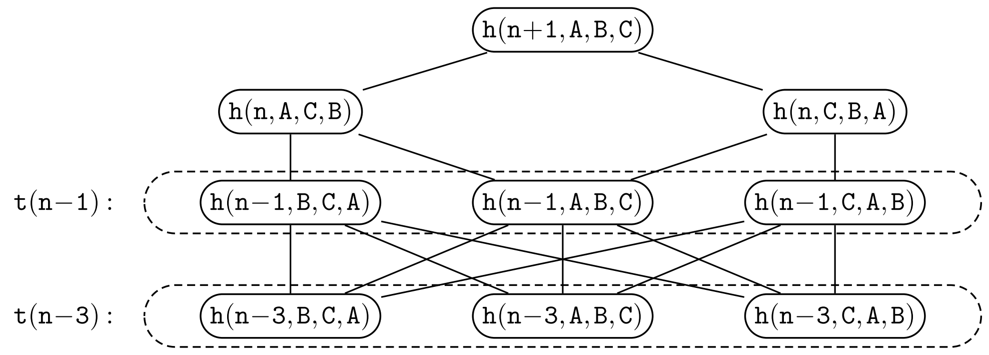


Figure 1: An upper portion of the call graph *m-dag* of the Hanoi function $\mathtt{h(n{+}1,A,B,C)}$. An edge from an upper node to a lower node denotes that the upper call requires the lower call. Dashed lines denote tuples.

We get:

$$\mathtt{h(0,A,B,C)} = \varepsilon \qquad \mathtt{h(1,A,B,C)} = \mathtt{AB}$$
$$\mathtt{h(n{+}2,A,B,C)} = \mathtt{u} :: \mathtt{AC} :: \mathtt{v} :: \mathtt{AB} :: \mathtt{w} :: \mathtt{CB} :: \mathtt{u} \quad \mathtt{where}\ \langle \mathtt{u,v,w} \rangle = \mathtt{t(n)} \quad \text{for } \mathtt{n} \geq 0$$
$$\mathtt{t(0)} = \langle \varepsilon,\varepsilon,\varepsilon \rangle \qquad \mathtt{t(1)} = \langle \mathtt{AB,BC,CA} \rangle$$
$$\mathtt{t(n{+}2)} = \langle \mathtt{u} :: \mathtt{AC} :: \mathtt{v} :: \mathtt{AB} :: \mathtt{w} :: \mathtt{CB} :: \mathtt{u},\quad \mathtt{v} :: \mathtt{BA} :: \mathtt{w} :: \mathtt{BC} :: \mathtt{u} :: \mathtt{AC} :: \mathtt{v},$$
$$\mathtt{w} :: \mathtt{CB} :: \mathtt{u} :: \mathtt{CA} :: \mathtt{v} :: \mathtt{BA} :: \mathtt{w} \rangle \qquad \mathtt{where}\ \langle \mathtt{u,v,w} \rangle = \mathtt{t(n)} \quad \text{for } \mathtt{n} \geq 0$$

Then, we can apply the schema equivalence stating that $\mathtt{g(n)}$ defined by the equations:

[2]Helena Rasiowa and Roman Sikorski gave in 1950 a first algebraic proof of Gödel Completeness Theorem for first-order predicate calculus.

$g(0)=a \qquad g(1)=b \qquad g(n+2)=c(g(n)) \quad \text{for } n \geq 0$

is equal to the value of `res` returned by the program:

```
if even(n) then res = a else res = b;
while n > 1 do res = c(res); n = n - 2 od
```

We get the following program, where for $k=1,2,3$, `Tk` denotes the `k`-th component of the triple `T`:

```
          {n ≥ 0}
if n = 0 then Hanoi = ε else
if n = 1 then Hanoi = AB else
begin n = n - 2; if even(n) then T = ⟨ε,ε,ε⟩ else T = ⟨AB,BC,CA⟩;
  while n > 1 do T = ⟨T1::AC::T2::AB::T3::CB::T1, T2::BA::T3::BC::T1::AC::T2,
                      T3::CB::T1::CA::T2::BA::T3⟩;  n = n - 2 od;
  Hanoi = T1::AC::T2::AB::T3::CB::T1
end
       {Hanoi = h(n,A,B,C)}
```

The technique we have presented is based only on the tupling strategy and a simple schema equivalence. That technique is successful also for the many variants of the Towers of Hanoi problem that can be found in the literature (see, among others, [23]). A different derivation for computing the Hanoi function can be done by introducing, besides the tuple $t(n)$, also the tuple $t'(n-2) =_{def} \langle h(n-2,A,C,B),\ h(n-2,C,B,A),\ h(n-2,B,A,C)\rangle$ corresponding to the calls of `h` at level $n-2$ (not depicted in Figure 1). We leave this derivation to the reader.

In a later paper I addressed the problem of finding the `m`-th move of algorithms which compute sequences of moves without computing any other move [54]. This problem arose as a generalization of the problem relative to the Towers of Hanoi. If the moves are computed by a function defined by a recurrence relation, then under suitable hypotheses, it is indeed possible to compute the `m`-th move without computing any other move. For the case of the Hanoi function $h(n,A,B,C)$ we have that the length $Lh(n)$ of the sequence of moves for `n` disks, satisfies the following equations: $Lh(0) = 0 \qquad Lh(n+1) = 2 \cdot Lh(n)+1 \quad \text{for } n \geq 0$

One can show [54] that the `m`-th move of $h(n,A,B,C)$, for $1 \leq m \leq 2^n-1$ and $n \geq 0$, can be computed using the deterministic finite automaton of Figure 2. We assume that $M[\ell..0]$ is the binary expansion of `m`, the most significant bit being at the leftmost position ℓ. Thus, $m = \sum_{i=0}^{\ell} M[i] \cdot 2^i$ and `m` is not a power of 2 iff $M[\ell..0] \notin 10^*$. Let $trans(X,p)$ denotes the state `Y` such that in the finite automaton of Figure 2 there is an arc from state `X` to state `Y` with label `p`.

```
i = ℓ; state = AB;
while M[i..0] ∉ 10* do begin state = trans(state,M[i]); i = i - 1 end od
```

The `m`-th move is the name of the final state, with `B` and `C` interchanged if an odd number of state transitions is made.

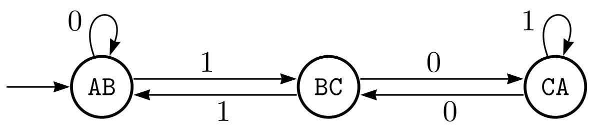

$$\begin{tikzcd} \arrow[r] & AB \arrow[loop above, "0"] \arrow[r, shift left, "1"] & BC \arrow[l, shift left, "1"] \arrow[r, shift left, "0"] & CA \arrow[l, shift left, "0"] \arrow[loop above, "1"] \end{tikzcd}$$

Figure 2: The finite automaton for computing the `m`-th move in the sequence $h(n,A,B,C)$ of moves for the Towers of Hanoi problem with `n` disks and pegs `A`, `B`, and `C`.

Suppose that we want to compute the `44`-th move of `h(6,A,B,C)`. The binary expansion of `44` is `101100`. Starting from the left, we take the prefix `101` up to (and excluding) the suffix in `10`* (in our case `100`). We perform the transitions on the automaton of Figure 2 starting from state `AB` according to that prefix (from left to right) and we get to state `CA`. Since the length of the prefix is odd (it is indeed `3`), the move to be computed is `BA`, that is, `CA` with `B` and `C` interchanged.

In a subsequent paper with Maurizio Proietti [57] we want to explore the idea of introducing *lists*, rather than *arrays* (indeed, tuples being of fixed size can be seen as arrays). Originally, this idea was suggested to me by Rod Burstall. Since every recursive function can be computed by using stacks (actually, two stacks are sufficient for computing any partial recursive function on natural numbers [30]), this technique seems to me, at first, not very relevant in the practice of improving the time complexity of a program or avoiding inefficient recursions. We explored the use of this technique and, indeed, we managed to achieve good results. In particular, the *list introduction strategy* can be used when the recursive calls do not generate a sequence of *cuts* of *constant* size in the m-dag of the function calls, and thus it does not allow the use of the tupling strategy. A cut in an m-dag is set C of nodes such that every path from the root to a leaf intersects C. In the case of the Hanoi function (see Figure 1) we have depicted the cuts associated with `t(n−1)` and `t(n−3)`. Both of them are of size `3` and thus, the tupling strategy (with three function calls) is successful. More details on cuts and their use for program transformation also in relation with pebble games [44] can be found in my Ph.D. thesis [51].

We used the list introduction strategy for deriving a program for computing the binomial coefficients: $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$. In this case the sequence of cuts from the root to the leaves is of increasing size. Indeed, $\binom{n+1}{k+1}$ requires the computations of $\binom{n}{k}$ and $\binom{n}{k+1}$, which in turns, require the computations of $\binom{n-1}{k-1}$, $\binom{n-1}{k}$, $\binom{n-1}{k+1}$, and so on. (Indeed, in the Pascal Triangle the basis has an increasing size when the height of the triangle increases). Therefore, the tupling strategy cannot be used.

Now, in order to show the power of the list introduction strategy, let us consider the n-queens problem. Details are in [57]. An $n \times n$ board configuration Qs is represented by a list of pairs of the form: $[\langle R_1, C_1\rangle, \ldots, \langle R_n, C_n\rangle]$, where for $i = 1, \ldots, n$, $\langle R_i, C_i\rangle$ denotes a queen placed in row R_i and column C_i. For $i = 1, \ldots, n$, the values of R_i and C_i belong to the list $[1, \ldots, n]$.

We start from the following initial program *Queens*:

1. $\mathit{queens}(\mathit{Ns}, \mathit{Qs}) \leftarrow \mathit{placequeens}(\mathit{Ns}, \mathit{Qs}),\ \mathit{safeboard}(\mathit{Qs})$
2. $\mathit{placequeens}([\,], [\,]) \leftarrow$
3. $\mathit{placequeens}(\mathit{Ns}, [Q|\mathit{Qs}]) \leftarrow \mathit{select}(Q, \mathit{Ns}, \mathit{Ns}1),\ \mathit{placequeens}(\mathit{Ns}1, \mathit{Qs})$
4. $\mathit{safeboard}([\,]) \leftarrow$
5. $\mathit{safeboard}([Q|\mathit{Qs}]) \leftarrow \mathit{safequeen}(Q, \mathit{Qs}),\ \mathit{safeboard}(\mathit{Qs})$
6. $\mathit{safequeen}(Q, [\,]) \leftarrow$
7. $\mathit{safequeen}(Q1, [Q2|\mathit{Qs}]) \leftarrow \mathit{notattack}(Q1, Q2),\ \mathit{safequeen}(Q1, \mathit{Qs})$

In order to place n queens we solve the goal $\mathit{queens}([1, \ldots, n], \mathit{Qs})$. By clause 1 we have that $\mathit{placequeens}([1, \ldots, n], \mathit{Qs})$ generates a board configuration Qs and $\mathit{safeboard}(\mathit{Qs})$ checks that in Qs no two queens lie on the same diagonal (either ‘up diagonal’ or ‘down diagonal’ in Dijkstra’s terminology [21]). We assume that $\mathit{notattack}(Q1, Q2)$ holds iff queen position (or queen, for short) $Q1$, that is, $\langle R1, C1\rangle$, is not on the same diagonal of the queen $Q2$. The tests that the queens are neither on the same row nor on the same column can be avoided by assuming that $\mathit{select}(Q, \mathit{Ns}, \mathit{Ns}1)$ holds iff Ns is a list of distinct numbers in $[1, \ldots, n]$, Q is queen $\langle R, C\rangle$ such that row R is the length of Ns and column C is a member of Ns, and $Ns1$ is the new list obtained from Ns by deleting the occurrence of C. The length of the list Ns decreases by one unit after each call of *placequeens*. In particular, we have that board configurations having k queens (with $1 \leq k \leq n$) are of the form: $[\langle n, c_1\rangle, \langle n-1, c_2\rangle, \ldots, \langle n-k+1, c_k\rangle]$, where $c_1, c_2, \ldots, c_k$ are distinct numbers in $[1, \ldots, n]$.

Program *Queens* solves the problem using the *generate-and-test* approach and it is not efficient. A more efficient program using an *accumulator* that stores the diagonals which are not safe, has been proposed in [67, page 255]. Efficiency is increased because backtracking is reduced.

By applying the list introduction strategy (which includes also some generalization steps) one can derive the following program *TransfQueens* whose behaviour is similar to that of the accumulator version. The various transformation steps are described in [57]. The higher efficiency of the final program is due to the fact that the test for a safe board configuration is 'promoted' into the process of generating new configurations, and the number of generated unsafe board configurations is decreased (see the *filter promotion* technique [5, 15]).

8. $queens([\,],[\,]) \leftarrow$
9. $queens(Ns,[Q|Qs]) \leftarrow select(Q,Ns,Ns1),\ genlist1(Ns1,Qs,[Q])$
10. $genlist1([\,],[\,],Ps) \leftarrow$
11. $genlist1(Ns,[Q1|Qs],[\,]) \leftarrow select(Q1,Ns,Ns1),\ genlist1(Ns1,Qs,[Q1])$
12. $genlist1(Ns,[Q1|Qs],[P1|Ps]) \leftarrow select(Q1,Ns,Ns1),\ notattack(P1,Q1),$
    $genlist2(Ns1,Qs,[P1],Ps,Q1)$
13. $genlist2(Ns1,Qs,Ps1,[\,],Q1) \leftarrow genlist1(Ns1,Qs,[Q1|Ps1])$
14. $genlist2(Ns1,Qs,Ps1,[P2|Ps2],Q1) \leftarrow notattack(P2,Q1),$
    $genlist2(Ns1,Qs,[P2|Ps1],Ps2,Q1)$

This program performs much less backtracking than the *Queens* program[3]. By clause 9, the first queen position Q is selected and $genlist1$ is called with its last argument storing the current board configuration, which is the list $[Q]$. When a new queen is placed at position $Q1$ and it is not attacked by the last queen placed at position $P1$ (see clause 12), $genlist2$ checks whether or not $Q1$ is attacked by the queens already present in the current configuration and whose positions are stored in Ps (see clauses 13 and 14). If $Q1$ is not attacked, the configuration is updated (see the last argument $[Q1|Ps1]$ of $genlist1$ in clause 13). Otherwise, if $Q1$ is attacked, by backtracking (see the atom *select* in clause 12), a different queen position is selected. If all positions for the new queen are under attack, then by backtracking (see the atoms *select* in clauses 9 and 11), the position of a previously placed queen, if there is one, is selected in a different way.

The explanation which we have just given about the derived program (clauses 8–14), may appear unclear to the non-expert reader, but one should note that it was not needed at all. Indeed, correctness of the derived program is guaranteed by the correctness of the transformation rules, and the efficiency improvement is due to filter promotion.

5 The Lambda Abstraction Strategy

While studying the tupling strategy and analyzing its power, a sentence by John Darlington, with whom I shared the office in Edinburgh, came often to my mind: "After unfolding, having

[3]In some experiments we have done, for 10 queens *TransfQueens* runs about 70 times faster than *Queens*.

done some local improvements (such as the ones obtained by the `where` abstraction as shown in Section 3 for the `fusc` function), you need to fold." This *need for folding* [16] is an important requirement. Folding steps make the local improvements to be become global, so that they can be replicated at each level of recursion and thus become significant.

However, folding steps need matchings between expressions and these matchings may be sometimes impossible. Generalization of constants to variables may allow matchings in some cases, but not always. In particular, when an expression should match one of its subexpressions, generalization of constant to variables does not help. In those cases we have suggested to construct functions from expressions [62]. This is done by replacing the expression $\mathtt{E}[\mathtt{e}]$ where the subexpression $\mathtt{e}$ occurs, by the application $(\lambda \mathtt{x}.\mathtt{E}[\mathtt{x}])\,\mathtt{e}$. We call this technique *lambda abstraction strategy* (or, as in other papers, *higher-order abstraction*).

Let us see how lambda abstraction works in the following two examples taken from [62]. The first example refers to the following program *Reverse* for reversing a list, where [], :, and @ denote the empty list, *cons*, and *append* on lists, respectively.

1. $\mathtt{rev}([\,]) = [\,]$
2. $\mathtt{rev}(\mathtt{a}\!:\!\ell) = \mathtt{rev}(\ell)\ @\ [\mathtt{a}]$
3. $[\,]\ @\ \mathtt{y} = \mathtt{y}$
4. $(\mathtt{a}\!:\!\ell)\ @\ \mathtt{y} = \mathtt{a} : (\ell\ @\ \mathtt{y})$

We want to derive a tail recursive definition of `rev`. We need `rev` to be the top operator of the right hand side of Eq. 2, that is, $\mathtt{rev}(\ell)@[\mathtt{a}]$, and by induction we need that right hand side to be $\mathtt{rev}(\ell)$. There is a subexpression mismatch between $\mathtt{rev}(\ell)@[\mathtt{a}]$ and $\mathtt{rev}(\ell)$. Then we proceed as follows: (i) instead of $\mathtt{rev}(\ell)$, we consider $\mathtt{rev}(\ell)@[\,]$, (ii) we generalize the constant [] to the variable `x`, thereby deriving $\mathtt{rev}(\ell)@\mathtt{x}$, and (iii) we abstract $\mathtt{rev}(\ell)@\mathtt{x}$ with respect to `x`, thereby deriving the function $\lambda \mathtt{x}.\ \mathtt{rev}(\ell)@\mathtt{x}$.

The definition of the new function $\mathtt{f}(\ell) =_{\mathtt{def}} \lambda \mathtt{x}.\ \mathtt{rev}(\ell)@\mathtt{x}$ is as follows.

5. $\mathtt{f}([\,]) = \lambda \mathtt{x}.\ \mathtt{rev}([\,])@\mathtt{x} = \{\text{by Eq. } 1\} = \lambda \mathtt{x}.\ [\,]@\mathtt{x} = \{\text{by Eq. } 3\} = \lambda \mathtt{x}.\mathtt{x}$
6. $\mathtt{f}(\mathtt{a}\!:\!\ell) = \lambda \mathtt{x}.\mathtt{rev}(\mathtt{a}\!:\!\ell)@\mathtt{x} = \lambda \mathtt{x}.(\mathtt{rev}(\ell)@[\mathtt{a}])@\mathtt{x} = \{\text{by associativity of } @\} =$
   $= \lambda \mathtt{x}.\mathtt{rev}(\ell)@([\mathtt{a}]@\mathtt{x}) = \lambda \mathtt{x}.\mathtt{rev}(\ell)@(\mathtt{a}\!:\!\mathtt{x}) = \{\text{by folding}\} = \lambda \mathtt{x}.(\mathtt{f}(\ell)\,(\mathtt{a}\!:\!\mathtt{x}))$

We also have:

7. $\mathtt{rev}(\ell) = \mathtt{f}(\ell)\,[\,]$

The derived program (Eqs. 5–7) is more efficient than program (Eqs. 1–4) because the expensive operation *append* has been replaced by the cheaper operation *cons*. Eqs. 5–7 are basically equivalent to the program proposed in [31] where a new representation for list has to be invented.

Note that the mechanization of the transformation we have now presented requires the use of associativity property for the append function. Thus, in general, it is important to have knowledge of the algebraic properties of the operations in use.

A second example refers to a problem proposed by Richard Bird [6]. Given a binary tree `t` we want to construct an isomorphic binary tree $\widetilde{\mathtt{t}}$ such that: (i) `t` and $\widetilde{\mathtt{t}}$ have the same multiset of leaves, and (ii) the leaves of $\widetilde{\mathtt{t}}$, when read from left to right, are in ascending order. One should derive a program which construct $\widetilde{\mathtt{t}}$ by making one traversal only of the tree `t`.

In order to solve this program Richard Bird uses the so called *locally recursive programs* whose semantics is quite complex and it is based on the *call-by-need* mode of evaluation. By using the tupling and lambda abstraction strategies we will get the desired program with the following advantages over Bird's solution: (i) the use of *call-by-value* semantics, (ii) the absence of local recursion, (iii) the leaves are sorted *on the fly*, and (iv) the computation of components of tuples is done only when they are required for later computations.

By $\mathtt{tip(n)}$ we denote a binary tree whose single leaf is the integer $\mathtt{n}$, and by $\mathtt{t1}\wedge\mathtt{t2}$ we denote a binary tree with children $\mathtt{t1}$ and $\mathtt{t2}$. By $\mathtt{hd}$ and $\mathtt{tl}$ we denote, as usual, the *head* and *tail* functions on lists. Our initial program is as follows.

1. $\mathtt{TreeSort(t)} = \mathtt{replace(t,sort(leaves(t)))}$ where:

(i) $\mathtt{leaves(t)}$ returns the list of the leaves of the tree $\mathtt{t}$, (ii) $\mathtt{sort}(\ell)$ rearranges the list ℓ in ascending order from left to right, and (iii) $\mathtt{replace}(\mathtt{t},\ell)$ uses in the left-to-right order the elements of the list ℓ to replace from left-to-right the leaves of the tree $\mathtt{t}$.

We assume that the length of ℓ is at least the number of leaves in $\mathtt{t}$. For instance, we have: $\mathtt{TreeSort((tip(1)\wedge tip(2))\wedge tip(1))} = \mathtt{(tip(1)\wedge tip(1))\wedge tip(2)}$. Here is the definition of the various functions required:

2. $\mathtt{leaves(tip(n))} = [\mathtt{n}]$
3. $\mathtt{leaves(t1\wedge t2)} = \mathtt{leaves(t1)}\ @\ \mathtt{leaves(t2)}$
4. $\mathtt{replace}(\mathtt{tip(n)},\ell) = \mathtt{tip}(\mathtt{hd}(\ell))$
5. $\mathtt{replace}(\mathtt{t1}\wedge\mathtt{t2},\ell) = \mathtt{replace}(\mathtt{t1},\mathtt{take}(\mathtt{k},\ell))\wedge\mathtt{replace}(\mathtt{t2},\mathtt{drop}(\mathtt{k},\ell))$ $\mathtt{where\ k = size(t1)}$
6. $\mathtt{take}(\mathtt{n},\ell) = \mathtt{if\ n = 0\ then}\ []\ \mathtt{else\ take}(\mathtt{n}-1,\ell)\ @\ [\mathtt{hd}(\mathtt{drop}(\mathtt{n}-1,\ell))]$
7. $\mathtt{drop}(\mathtt{n},\ell) = \mathtt{if\ n = 0\ then}\ \ell\ \mathtt{else\ tl}(\mathtt{drop}(\mathtt{n}-1,\ell))$

For instance, $\mathtt{take}(2,[\mathtt{a,b,c,d,e}]) = [\mathtt{hd}([\mathtt{a,b,c,d,e}]),\mathtt{hd}([\mathtt{b,c,d,e}])] = [\mathtt{a,b}]$ and
$\mathtt{drop}(2,[\mathtt{a,b,c,d,e}]) = \mathtt{tl}(\mathtt{tl}([\mathtt{a,b,c,d,e}])) = [\mathtt{c,d,e}]$.

As usual, given a list ℓ, we denote by $\mathtt{length}(\ell)$ the number of elements in ℓ. We assume that $0 \leq \mathtt{k} \leq \mathtt{length}(\ell)$ holds when evaluating $\mathtt{take}(\mathtt{k},\ell)$ and $\mathtt{drop}(\mathtt{k},\ell)$. For all list ℓ, for all $0 \leq \mathtt{n} \leq \mathtt{length}(\ell)$, we have $\ell = \mathtt{take}(\mathtt{n},\ell) @ \mathtt{drop}(\mathtt{n},\ell)$. The function $\mathtt{size(t)}$ returns the number of leaves in the tree $\mathtt{t}$. We have:

8. $\mathtt{size(tip(n))} = 1$
9. $\mathtt{size(t1\wedge t2)} = \mathtt{size(t1)} + \mathtt{size(t2)}$.

Here is the definition of $\mathtt{sort}$ using $\mathtt{merge}$ of two ordered lists:

10. $\mathtt{sort}(\ell) = \mathtt{if}\ \ell = []\ \mathtt{then}\ []\ \mathtt{else\ merge}([\mathtt{hd}(\ell)],\mathtt{sort}(\mathtt{tl}(\ell)))$.
11. $\mathtt{merge}([],\ell) = \ell$
12. $\mathtt{merge}(\ell,[]) = \ell$
13. $\mathtt{merge}(\mathtt{a}{:}\ell1,\ \mathtt{b}{:}\ell2) = \mathtt{if\ a \leq b\ then\ a : merge}(\ell1,\ \mathtt{b}{:}\ell2)\ \mathtt{else\ b : merge}(\mathtt{a}{:}\ell1,\ \ell2)$

Unfortunately, $\mathtt{TreeSort(t)}$ traverses the tree $\mathtt{t}$ twice: a first visit is for collecting the leaves, and a second visit is for replacing them in ascending order.

Now, let us start off the derivation of the one traversal algorithm by getting the inductive definition of $\mathtt{TreeSort(t)}$. From Eq. 1 we get:

14. $\mathtt{replace(tip(n),sort(leaves(tip(n))))} = \mathtt{replace(tip(n),sort([n]))} = \mathtt{tip(n)}$
15. $\mathtt{replace(t1\wedge t2,sort(leaves(t1\wedge t2)))} = \mathtt{replace}(\mathtt{t1},\mathtt{take}(\mathtt{size(t1)},\ell))\ \wedge$
    $\wedge\ \mathtt{replace}(\mathtt{t2},\mathtt{drop}(\mathtt{size(t1)},\ell))$ $\mathtt{where}\ \ell = \mathtt{sort(leaves(t1\wedge t2))}$

Now no folding step can be performed, because in $\mathtt{replace}(\mathtt{t1},\mathtt{take}(\mathtt{size(t1)},\ell))$ the subexpression $\mathtt{take}(\mathtt{size(t1)},\ell)$ does not match $\mathtt{sort(leaves(t1))}$. Similarly, for the subtree $\mathtt{t2}$, instead of $\mathtt{t1}$. By the lambda abstraction we generalize the mismatching subexpression to the list variable $\mathtt{z}$, and we introduce the function $\lambda\mathtt{z}.\ \mathtt{replace(t,z)}$ whose definition is as follows (the details are in [62]):

16. λz. replace(tip(n),z) = λz. tip(hd(z))
17. λz. replace(t1∧t2,z) = λz. ((λy. replace(t1,y) take(k,z)) ∧
    ∧ ((λy. replace(t2,y)) drop(k,z)))   where k = size(t1)

The functions λz. replace(t,z) and sort(leaves(t)) visit the same tree t. We apply the tupling strategy and we define the function:

T(t) $=_{\texttt{def}}$ ⟨λz. replace(t,z), sort(leaves(t))⟩

whose explicit definition is:

18. T(tip(n)) = ⟨λz. tip(hd(z)), [n]⟩
19. T(t1∧t2) = ⟨λz. ((a1 take(size(t1),z)) ∧ (a2 drop(size(t1),z))), merge(b1,b2)⟩
    where ⟨a1,b1⟩ = T(t1) and ⟨a2,b2⟩ = T(t2)

Now T(t1), take(size(t1),z), and drop(size(t1),z) visit the same tree t1. We apply the tupling strategy and we introduce the new function:

U(t,y) $=_{\texttt{def}}$ ⟨λz. replace(t,z), sort(leaves(t)), take(size(t),y), drop(size(t),y)⟩

We get the following explicit definition for U(tip(n),y):

U(tip(n),y) = ⟨λz. tip(hd(z)), [n], [hd(y)], tl(y)⟩

However, when looking for the explicit definition of U(t1∧t2,y) we get again a subexpression mismatch (see [62]) and we use again lambda abstraction for the last two components of the 4-tuple U(t,y). Thus, we introduce the following function:

V(t) $=_{\texttt{def}}$ ⟨λz.replace(t,z), sort(leaves(t)), λz.take(size(t),z), λz.drop(size(t),z)⟩

whose explicit definition is:

20. V(tip(n)) = ⟨λz. tip(hd(z)), [n], λz. [hd(z)], λz. tl(z)⟩
21. V(t1∧t2) = ⟨λz.((a1 (c1 z)) ∧ (a2 (d1 z))), merge(b1,b2), λz.((c1 z) @ (c2 (d1 z))), λz.(d2 (d1 z))⟩
    where ⟨a1,b1,c1,d1⟩ = V(t1) and ⟨a2,b2,c2,d2⟩ = V(t2)

We get the following program such that for all trees t, NewTreeSort(t) = TreeSort(t) (see Eq. 1):

22. NewTreeSort(t) = (a2 b2)   where ⟨a2,b2⟩ = T(t)
18. T(tip(n)) = ⟨λz. tip(hd(z)), [n]⟩
23. T(t1∧t2) = ⟨λz. ((a1 c1,z)) ∧ (a2 d1,z))), merge(b1,b2)⟩
    where ⟨a1,b1,c1,d1⟩ = V(t1) and ⟨a2,b2⟩ = T(t2)

together with Eqs. 20 and 21 for the function V(t).

A further improvement of this program can be made by avoiding the append function @ occurring in Eq. 21. One can use the same technique of lambda abstraction shown in the *Reverse* example at the beginning of this section. We consider a variant of the function V(t) whose 3rd component is the abstraction λz x. take(size(t),z) @ x, instead of λz. take(size(t),z). The function T*(t) is like T(t), but uses V*(t), instead of V(t). We get the following final program such that for all trees t, NewTreeSort*(t) = TreeSort(t):

22*. NewTreeSort*(t) = (a2 b2)   where ⟨a2,b2⟩ = T*(t)
18*. T*(tip(n)) = ⟨λz. tip(hd(z)), [n]⟩
23*. T*(t1∧t2) = ⟨λz. ((a1 (c1(z,[]))) ∧ (a2 (d1 z))), merge(b1,b2)⟩
    where ⟨a1,b1,c1,d1⟩ = V*(t1) and ⟨a2,b2⟩ = T*(t2)
20*. V*(tip(n)) = ⟨λz. tip(hd(z)), [n], λz x. hd(z):x, λz. tl(z)⟩

21*. $\texttt{V}^*(\texttt{t1}\wedge\texttt{t2}) = \langle\lambda\texttt{z}.((\texttt{a1 (c1(z,[]))})\wedge(\texttt{a2 (d1 z)})),\ \texttt{merge(b1,b2)},$
$\lambda\texttt{z x}.(\texttt{c1(z,c2((d1 z),x))}),\ \lambda\texttt{z}.(\texttt{d2 (d1 z)})\rangle$
$\texttt{where}\ \langle\texttt{a1,b1,c1,d1}\rangle = \texttt{V}^*(\texttt{t1})\ \texttt{and}\ \langle\texttt{a2,b2,c2,d2}\rangle = \texttt{V}^*(\texttt{t2})$

Computer experiments performed at the time of writing the paper [62] from which we take this example, show that the computation of the final function `NewTreeSort*(t)` is faster than the one of the initial function `TreeSort(t)` for trees whose size is greater than about 30. For trees of smaller size the overhead of dealing with functions is not compensated by the fact that the input tree is visited once only.

Note also that since lambda expressions do not have free variables, we can operate on them by using pairs of bound variables and function bodies, instead of the more expensive closures. Thus, for instance, $\lambda\texttt{z.expr}$ can be represented by the pair $\langle\texttt{z},\texttt{expr}\rangle$.

Some years later Maurizio Proietti and I have studied the application of the lambda abstraction strategy in the area of logic programming. As in functional programs where we have lambda expressions denoting functions, in logic programming we should have terms denoting goals, and thus goals should be allowed to occur as arguments of predicates. To allow goals as arguments, we have proposed a novel logic language, we have defined its semantics, and we have provided for it a set of unfold/fold transformations rules, together with some goal replacement rules, such as the one stating the equivalence of the goal $g \wedge \mathit{true}$ with the goal g [56, 63]. Those rules have been proved correct.

Here is an example of efficiency improvement obtained by program transformation in this novel language. This transformation has not been mechanized, but we believe that it is not hard to do it. Details can be found in [63, Section 7.1]. Let us consider a program which given a binary tree (either $l(N)$ or $t(L,N,R)$), (i) flips all its left and right subtrees, and (ii) checks in a subsequent traversal of the tree, whether or not all labels are natural numbers.

1. $\mathit{flipcheck}(X,Y) \leftarrow \mathit{flip}(X,Y),\ \mathit{check}(Y)$
2. $\mathit{flip}(l(N),l(N)) \leftarrow$
3. $\mathit{flip}(t(L,N,R),t(FR,N,FL)) \leftarrow \mathit{flip}(L,FL),\ \mathit{flip}(R,FR)$
4. $\mathit{check}(l(N)) \leftarrow \mathit{nat}(N)$
5. $\mathit{check}(t(L,N,R)) \leftarrow \mathit{nat}(N),\ \mathit{check}(L),\ \mathit{check}(R)$
6. $\mathit{nat}(0) \leftarrow$
7. $\mathit{nat}(s(N)) \leftarrow \mathit{nat}(N)$

We derived the following program which traverses the input tree only once and uses the continuation passing style:

8 $\mathit{flipcheck}(X,Y) \leftarrow \mathit{newp}(X,Y,G,\mathit{true},G)$
9 $\mathit{newp}(l(N),l(N),G,C,D) \leftarrow \mathit{eq\_c}(G,\mathit{nat\_c}(N,C),D)$
10 $\mathit{newp}(t(L,N,R),t(FR,N,FL),G,C,D) \leftarrow$
$\mathit{newp}(L,FL,U,C,\ \mathit{newp}(R,FR,V,U,\ \mathit{eq\_c}(G,\mathit{nat\_c}(N,V),D)))$
11 $\mathit{nat\_c}(0,C) \leftarrow C$
12 $\mathit{nat\_c}(s(N),C) \leftarrow \mathit{nat\_c}(N,C)$

For the predicate *eq_c* we assume that: $\vdash \forall(\mathit{eq\_c}(X,Y,C) \leftrightarrow ((X=Y)\wedge C))$.

6 Communications and Parallelism

While at Edinburgh I had the privilege of attending a course on the Calculus of Communicating Systems (CCS) by Professor Robin Milner (1934-2010) [42]. I remember the day when Robin Milner and Gordon Plotkin decided the name to be given to this new calculus. As I was told, they first decided that the name should have been of three letters only! I appreciated the beauty of the calculus which resembles a development of lambda calculus. The application of *a function* $\lambda\mathtt{x}.\mathtt{e}[\mathtt{x}]$ to *an argument* `a` can, indeed, be understood as a communication which takes place between: (i) the 'function agent' and (ii) the 'argument agent' through the 'port' named λ. After their communication, which is called a *handshaking*, the agents continue their respective activities, namely, (i) the function agent does the evaluation of $\mathtt{e}[\mathtt{a}]$, that is, the body $\mathtt{e}[\mathtt{x}]$ of the function where the variable `x` have been bound to the value `a`, and (ii) the argument agent does nothing, that is, it become the *null-agent* (indeed, for the rest of the computation, the argument has nothing left to do).

At about the same time, Professor Tony Hoare in Oxford was developing his calculus of Concurrent Sequential Programs (CSP) [29]. I remember a visit that Tony Hoare made to Robin Milner at Edinburgh and the stimulating seminar Hoare gave on CSP on that occasion.

In subsequent years, I thought of exploring the power of communications and parallelism in functional programming, also because the various components of the tuples introduced by the tupling strategy can be computed in parallel. These components can be considered as independent agents which may synchronize at the end of their computations. During those years, the notion of communicating agents was emerging quite significantly in various programming paradigms.

Andrzej Skowron and I did some work in this area and we proposed (some variants of) a functional language with communications [60, 61]. Each function call is assumed to be an *agent*, that is, a triple of the form $\langle\mathtt{x},\mathtt{m}\rangle::\mathtt{expr}$, where `x` is its name, `m` is its *message*, that is, its local information, and `expr` is its expression, that is, the task it has to perform. The operational semantics of the language is based on the conditional rewriting of sets (or multisets) of agents, similarly to what is done in coordination languages (see, for instance, [25]).

As an example of a functional program with communications which we proposed, let us consider the following program for computing the familiar Fibonacci function.

The variable `x` ranges over agent names which are strings constructed from `x` as the following grammar indicates: $\mathtt{x} ::= \varepsilon \mid \mathtt{x.0} \mid \mathtt{x.1}$. The left and right son-calls of the agent whose name is `x` have names `x.0` and `x.1`, respectively. By default, the name of the agent of the initial function call is the empty string ε.

In our example, the variables `ms` and `ms1` range over the three message constants: `R` (for *ready*), `R1` (for *ready*1), and `W` (for *wait*). Agents with messages `R` and `R1` may make rewritings, while agents with message `W` cannot (see Rules `1`–`4` below). The variables `n` and `val` range over integers and the variable `exp` ranges over integer expressions.

1. $\{\langle\mathtt{x},\mathtt{ms}\rangle::\mathtt{fib(0)}\} \Rightarrow \{\langle\mathtt{x},\mathtt{ms}\rangle::\mathtt{0}\}$ if $\mathtt{ms}=\mathtt{R}$ or $\mathtt{ms}=\mathtt{R1}$
2. $\{\langle\mathtt{x},\mathtt{ms}\rangle::\mathtt{fib(1)}\} \Rightarrow \{\langle\mathtt{x},\mathtt{ms}\rangle::\mathtt{1}\}$ if $\mathtt{ms}=\mathtt{R}$ or $\mathtt{ms}=\mathtt{R1}$
3. $\{\langle\mathtt{x},\mathtt{R}\rangle::\mathtt{fib(n+2)}\} \Rightarrow \{\langle\mathtt{x},\mathtt{R}\rangle::+(\mathtt{x.0},\mathtt{x.1}),\ \langle\mathtt{x.0},\mathtt{R}\rangle::\mathtt{fib(n+1)},\ \langle\mathtt{x.1},\mathtt{R1}\rangle::\mathtt{fib(n)}\}$ if $\mathtt{n}\geq 0$
4. $\{\langle\mathtt{x},\mathtt{R1}\rangle::\mathtt{fib(n+2)}\} \Rightarrow \{\langle\mathtt{x},\mathtt{R1}\rangle::+(\mathtt{x.0},\mathtt{x.1}),\ \langle\mathtt{x.0},\mathtt{W}\rangle::\mathtt{fib(n+1)},\ \langle\mathtt{x.1},\mathtt{R}\rangle::\mathtt{fib(n)}\}$ if $\mathtt{n}\geq 0$
5. $\{\langle\mathtt{x.0},\mathtt{ms}\rangle::\mathtt{val},\ \langle\mathtt{x},\mathtt{ms1}\rangle::+(\mathtt{x.0},\mathtt{exp})\} \Rightarrow \{\langle\mathtt{x},\mathtt{ms1}\rangle::+(\mathtt{val},\mathtt{exp})\}$

6. $\{\langle$x.1,ms$\rangle$ :: val, $\langle$x,ms1$\rangle$ :: +(exp,x.1)$\}$ $\Rightarrow$ $\{\langle$x.1,ms$\rangle$ :: val, $\langle$x,ms1$\rangle$ :: +(exp,val)$\}$
7. $\{\langle$x.0.1,R1$\rangle$ :: val, $\langle$x.1.0,W$\rangle$ :: exp$\}$ $\Rightarrow$ $\{\langle$x.0.1,R1$\rangle$ :: val, $\langle$x.1.0,R$\rangle$ :: val$\}$

Rules 1 and 2 are the expected ones for computing fib(0) and fib(1). The recursive call of fib(n+2) has two variants (see Rules 3 and 4) so to be able to evaluate the call of agent x.0.1 in a different way than that of agent x.1.0. The expression +(x.0,x.1) has the effect that, once the values of the son-calls are evaluated and sent to the father-call, according to Rules 5 and 6, then the father-call silently performs the sum of the values it has received. Rule 7 sends the value computed by agent x.0.1 to agent x.1.0. This communication is correct and improves efficiency. Indeed, by our program the value of fib(n−1) which is needed for computing fib(n+1) and fib(n), is computed once only. Note, in fact, that one of the two agents which have to compute fib(n−1), has the message W and cannot make further rewritings.

We have considered the problem of how to modify the rules of the programs when acquiring knowledge of new facts about the functions to be evaluated for improving program efficiency. In the case of the Fibonacci function, one such fact may be the equality of the expressions to be computed by the agents x.0.0 and x.1.

Note that the above Rules 1–7 do not perform the on-the-fly garbage collection of the agents because right-sons are not erased. To overcome this problem one may use more complex messages [61] so that every agent knows the agents which are waiting for receiving the value it computes. If there are none, the agent may be erased once it has sent its value to the father-call.

Note also that, if instead of Rule 6, we use the simpler equation:

$\widetilde{6}$. $\{\langle$x.1,ms$\rangle$ :: val, $\langle$x,ms1$\rangle$ :: +(exp,x.1)$\}$ $\Rightarrow$ $\{\langle$x,ms1$\rangle$ :: +(exp,val)$\}$

deadlock may be generated. We have also proposed a modal logic for proving correctness of our functional programs with agents and communications [59] and, in particular, the absence of deadlock. Unfortunately, no implementation of our language proposal and its modal logic has been done.

Concerning a more theoretical study of parallelism and communications, Anna Labella and I considered categorical models for calculus with handshaking communications both in the case of CCS [42] and CSP [29]. We were inspired by the definition of the cartesian closed categories for providing models of the lambda calculus.

We followed an approach different from Winskel's one [73]. We did not give an *a priori* definition of a categorical structure, where the embedding of the algebraic models of CCS or CSP might not be completely satisfactory. We started, instead, from the algebraic models, based on labelled trees of various kinds, and we defined suitable categories of labeled trees where one can interpret all the basic operations of CCS and CSP. In a sense, we followed the approach presented many years earlier by Rod Burstall for the description of flowchart programs [8]. The details of our categorical constructions can be found in [33, 36].

In some models of ours we used *enriched categories* [34]. An enriched category is a category where the sets of morphisms associated with the pairs of objects, are replaced by objects from a fixed monoidal category. For lack of space we will not enter into the details here.

7 Transformation and Verification in Logic Programming

While studying at Edinburgh, I thought of applying the transformation methodology to CCS agents. I remember talking to Robin Milner about this idea. He did not show much interest maybe because for him it was more important to first acquire a good understanding the equivalences between terms in the CCS calculus, before applying them to the transformations of agents which, of course, should be equivalence preserving.

Then, I thought of applying program transformation to the area of logic programming which I first studied during my Ph.D. research at Edinburgh. At that time William Clocksin and Chris Mellish were writing their popular book on Prolog [13]. I remember reading some parts of a draft of the book. Also I had the chance of looking at David Warren's report on how to compile logic programs [70]. I also read his paper comparing the Prolog implementation with Lisp [72] and the later report on the Warren Abstract Machine [71]. From those days I still remember David's kindness, his cooperation with Fernando Pereira, and his love for plants and flowers.

A few years later, when back in Italy, I was introduced by Anna Labella to her former student Maurizio Proietti who, not long before, had graduated in Mathematics at Rome University 'La Sapienza', defending a thesis on Category Theory. I spoke to Maurizio and I introduced him to logic programming [38]. I also encouraged him to work in the field of logic program transformation. He kindly accepted. The basis of his work was a paper by Hisao Tamaki and Taisuke Sato [68] that soon afterwards became the standard reference for logic program transformation.

That was the beginning of Maurizio's cooperation with me. He was first funded by a research grant from the private company Enidata (Rome) and soon later, he became a researcher of the Italian National Research Council in Rome. We first considered some techniques for finding the *eureka predicates*, that is, the predicate definitions to be introduced during program transformation [64].

Besides the definition introduction, unfolding, and folding rules, we have used for our transformations a rule called *Generalization + Equality Introduction* (see also [7] for a similar rule when proving theorems in functional programs). By this rule, a clause of the form $H \leftarrow A_1, \ldots, A_n$ is generalized to the clause $H \leftarrow GenA_1, \ldots, GenA_n,\ X_1 = t_1,\ \ldots, X_n = t_r$, where $(GenA_1, \ldots, GenA_n)\,\vartheta = (A_1, \ldots, A_n)$ being ϑ the substitution $\{X_1/t_1, \ldots,\ X_r/t_r\}$.

We have also introduced: (i) the class of *non-ascending programs*, where, among other properties, each variable should occur in an atom at most once, (ii) the *synchronized descent rule* (SDR) for driving the unfolding steps by selecting the atoms to be unfolded, and (iii) the *loop absorption strategy* for the synthesis of the eureka predicates. We have also characterized classes of programs in which that strategy is guaranteed to be successful.

Let us see a simple example of application of the loop absorption strategy. Here is a program, called *Comsub*, for computing common subsequences of lists.

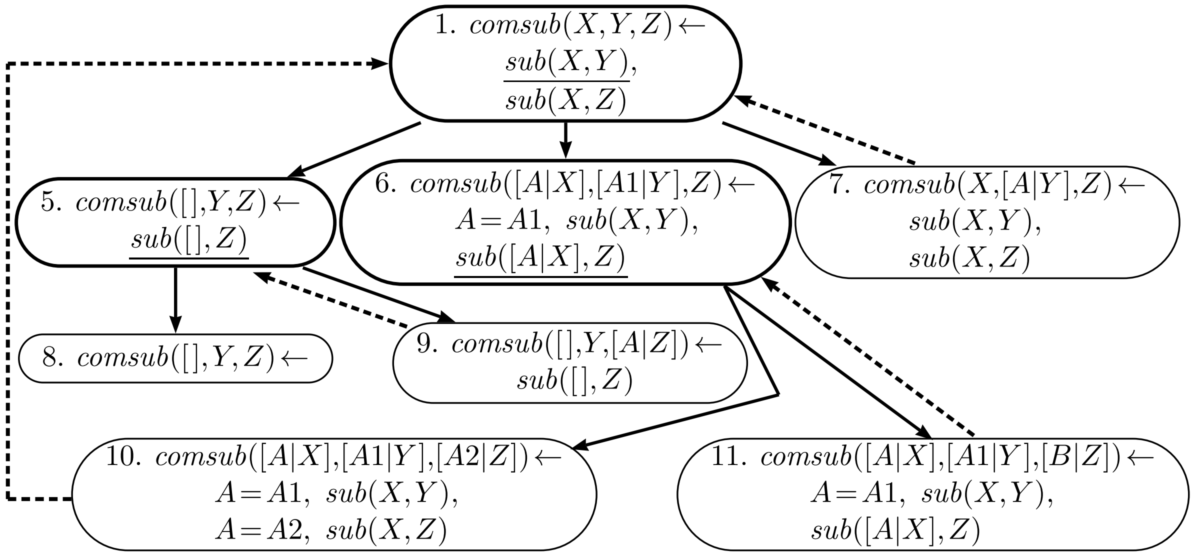
1. $comsub(X,Y,Z) \leftarrow sub(X,Y),\ sub(X,Z)$
2. $sub([\,],X) \leftarrow$
3. $sub([A|X],[A|Y]) \leftarrow sub(X,Y)$
4. $sub(X,[A|Y]) \leftarrow sub(X,Y)$

where $sub(X,Y)$ holds iff X is a sublist of Y. The order of the elements should be preserved, but the elements in X need not to be consecutive in Y. For instance, [1,2] is a sublist of [1,3,2,3], while [2,1] is not. We want to derive a program where the double visit of the list X in clause 1 is avoided.

First, we make the given program to be non-ascending by replacing clause 3 by the following clause:

3.1 $sub([A|X],[A1|Y]) \leftarrow A = A1, sub(X,Y)$

Let *Comsub*1 be the set $\{1,2,3.1,4\}$ of clauses. In Figure 3 we have depicted an upper portion of the unfolding tree for *Comsub*1. In that figure we have underlined the atoms which



1. comsub(X,Y,Z) ← sub(X,Y), sub(X,Z)
5. comsub([],Y,Z) ← sub([],Z)
6. comsub([A|X],[A1|Y],Z) ← A=A1, sub(X,Y), sub([A|X],Z)
7. comsub(X,[A|Y],Z) ← sub(X,Y), sub(X,Z)
8. comsub([],Y,Z) ←
9. comsub([],Y,[A|Z]) ← sub([],Z)
10. comsub([A|X],[A1|Y],[A2|Z]) ← A=A1, sub(X,Y), A=A2, sub(X,Z)
11. comsub([A|X],[A1|Y],[B|Z]) ← A=A1, sub(X,Y), sub([A|X],Z)

Figure 3: An upper portion of the unfolding tree for *Comsub*1.

are unfolded. Solid down-arrows denote unfolding, and dashed up-arrows denote loops which suggest the definition clauses needed for folding, as we will explain. In clause 6 we unfold the atom $sub([A|X],Z)$ which is selected by the SDR rule. Indeed, by the synchronized descent rule, in clause 6 we have to unfold that atom, because in its ancestor-clause 1 we have unfolded the other atom $sub(X,Y)$ occurring in the body of that ancestor. Unfolding is stopped when the recursive defined atoms in the body of a leaf-clause, say L, are subsumed by the body of an ancestor-clause, say A. In this case we say that a loop of the form $\langle A,L\rangle$ has been detected. Details can be found in [64].

According to the loop absorption strategy, for each detected loop $\langle A,L\rangle$ we introduce a new definition clause D so that the bodies of both clauses A and L can be folded using D. The loops $\langle 1,10\rangle$ and $\langle 1,7\rangle$ need not a new definition because we have clause 1 defining *comsub*. The loops $\langle 5,9\rangle$ and $\langle 6,11\rangle$ require the following two new predicate definitions

$newsub(Z) \leftarrow sub([],Z)$ for loop $\langle 5,9\rangle$

$newcomsub(A,X,Y,Z) \leftarrow sub(X,Y),\ sub([A|X],Z)$ for loop $\langle 6,11\rangle$

By performing the unfolding and folding steps which correspond to the subtrees rooted in clauses 1, 5, and 6 of Figure 3, we get the explicit definitions of the predicates *newsub* and *newcomsub*.

Eventually, by simplifying the equalities, we get the following program:

5. $comsub([],Y,Z) \leftarrow$ $(*)$
6. $comsub([],Y,[A|Z]) \leftarrow newsub(Z)$
7. $comsub([A|X],[A|Y],Z) \leftarrow newcomsub(A,X,Y,Z)$ $(*)$
8. $comsub(X,[A|Y],Z) \leftarrow comsub(X,Y,Z)$ $(*)$

9. $newcomsub(A, X, Y, [A|Z]) \leftarrow comsub(X, Y, Z)$ $(*)$
10. $newcomsub(A, X, Y, [B|Z]) \leftarrow newcomsub(A, X, Y, Z)$ $(*)$
11. $newsub(Z)$
12. $newsub([A|Z]) \leftarrow newsub(Z)$

Now clause 6 is subsumed by clause 5 and can be erased. Then, also clauses 11 and 12 can be erased and the final program is made out of the marked clauses 5, 7–10 only. This final program is equal to the one derived by Tamaki-Sato [68]. Note that our derivation does not rely on human intuition and can easily be mechanized. The computation of all solutions of the goal $comsub(X, Y, Z)$, where X is a free variable and Y and Z are ground lists of 10 elements, is about 6 times faster when using the final program, instead of the initial one [64]. A development of the technique we have now illustrated can be found in [65].

The following example, taken from a paper of ours [58] written some years later in honor of Professor Robert Kowalski, shows an application of the program transformation methodology also to the case when clauses may have negated atoms in their body. For that kind of logic programs, called *locally stratified logic programs*, we have also provided the transformation rules that can be applied and we have shown that they are correct, in the sense that they preserve the perfect model semantics. The details on the rules and the definition of the perfect model semantics can be found in [58].

Let us consider the following program *CFParser* for deriving a word generated by a given context-free grammar over the alphabet $\{a, b\}$:

1. $derive([\,], [\,]) \leftarrow$
2. $derive([A|S], [A|W]) \leftarrow terminal(A),\ derive(S, W)$
3. $derive([A|S], W) \leftarrow nonterminal(A),\ production(A, B),\ append(B, S, T),\ derive(T, W)$

4. $nonterminal(s) \leftarrow$	5. $nonterminal(x) \leftarrow$
6. $terminal(a) \leftarrow$	7. $terminal(b) \leftarrow$
8. $production(s, [a, x, b]) \leftarrow$	9. $production(x, [\,]) \leftarrow$
10. $production(x, [a, x]) \leftarrow$	11. $production(x, [a, b, x]) \leftarrow$
12. $word([\,]) \leftarrow$	13. $word([A|W]) \leftarrow terminal(A),\ word(W)$
14. $append([\,], Ys, Ys) \leftarrow$	15. $append([A|Xs], Ys, [A|Zs]) \leftarrow append(Xs, Ys, Zs)$

The relation $derive([s], W)$ holds iff the word W can be derived from the start symbol s using the following productions (see clauses 8–11):

$s \rightarrow a\,x\,b \qquad\qquad x \rightarrow \varepsilon \mid a\,x \mid a\,b\,x$

The nonterminal symbols are s and x (see clauses 4 and 5), the terminal symbols are a and b (see clauses 6 and 7), words in $\{a, b\}^*$ are represented as lists of a's and b's, and the empty word ε is represented as the empty list $[\,]$.

The relation $derive(L, W)$ holds iff L is a sequence of terminal or nonterminal symbols from which the word W can be derived by using the productions.

We would like to derive an efficient program for an initial goal G of the form:

$word(W),\ \neg derive([s], W)$

which holds in the perfect model of the program *CFParser* iff W is a word which is *not* derivable from s by using the given context-free grammar. We perform our two step program derivation presented in [58, Section 2.3]. In the first step, from goal G we derive the following two clauses:

16. $g(W) \leftarrow word(W), \neg new1(W)$
17. $new1(W) \leftarrow derive([s], W)$

In the second step, we apply the *unfold-definition-folding strategy* presented in [65]. We will not recall here the formal definition of this strategy. It will be enough to say that it is similar to the loop absorption strategy we have seen in action in the above derivation starting from the *Comsub*1 program.

For our *CFParser* program, at the end of the second step, we get:

$g([\,]) \leftarrow$	$g([a\|A]) \leftarrow new2(A)$	$g([b\|A]) \leftarrow new3(A)$
$new2([\,]) \leftarrow$	$new2([a\|A]) \leftarrow new4(A)$	$new2([b\|A]) \leftarrow new5(A)$
$new3([\,]) \leftarrow$	$new3([a\|A]) \leftarrow new3(A)$	$new3([b\|A]) \leftarrow new3(A)$
$new4([\,]) \leftarrow$	$new4([a\|A]) \leftarrow new4(A)$	$new4([b\|A]) \leftarrow new6(A)$
$new5([a\|A]) \leftarrow new3(A)$	$new5([b\|A]) \leftarrow new3(A)$	
$new6([a\|A]) \leftarrow new4(A)$	$new6([b\|A]) \leftarrow new5(A)$	

This program corresponds to the deterministic finite automaton of Figure 4. Each predicate of the derived program is a state, (ii) g is the initial state, (iii) a state p is final iff it has a clause of the form $p([\,]) \leftarrow$, (iv) a clause of the form $p([\ell|A]) \leftarrow q(A)$ denotes a transition with label ℓ from p to q. Note that the derivation of the final program that corresponds to a finite automaton has been possible because the context-free grammar indeed generates a regular language.

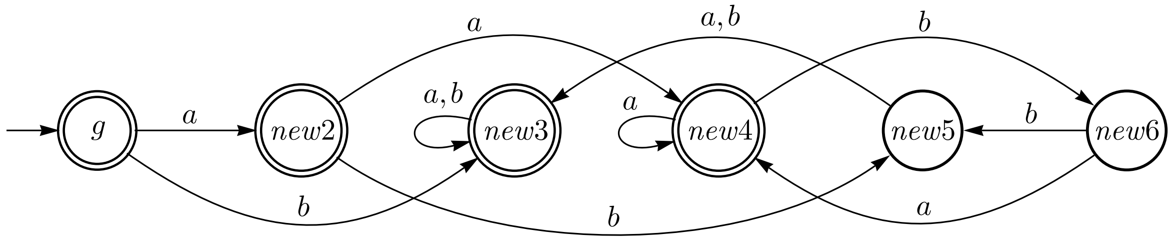


Figure 4: The finite automaton which accepts the words which are *not* generated from s by the productions: $s \rightarrow a\,x\,b$ and $x \rightarrow \varepsilon \mid a\,x \mid a\,b\,x$. State g is the initial state and the final states have double circles.

Finally we present an example on how to use the transformation methodology for the verification of program properties. This example is the so called *Yale Shooting Problem* which is often used in temporal reasoning. This problem can be described and formalized as follows.

We have a person and a gun. Three *events* are possible: (e1) a *load* event, when the gun is loaded, (e2) a *shoot* event, when the gun shoots, and (e3) a *wait* event, when nothing happens (see clauses 1–3 below). A *situation* is (the result of) a sequence of events. A sequence is represented as a list. We assume that, as time progresses, the list grows ‘to the left’, that is, given the current list S of events, when a new event E occurs, the new list of events is $[E|S]$. In any situation, at least one of the following three facts *holds*: (f1) the person is *alive*, (f2) the person is *dead*, and (f3) the gun is *loaded* (see clauses 4–6 below).

We also assume the following hypotheses (see clauses 7–11 and note the presence of a negated atom in clause 11). (s1) In the initial situation denoted by the empty list, the person is *alive.* (s2) After a *load* event the gun is *loaded.* (s3) If the gun is *loaded*, then after a *shoot* event the person is *dead.* (s4) If the gun is *loaded*, then it is *abnormal* that after a *shoot* event the person is *alive.* (s5) *Inertia Axiom*: If a fact F holds in a situation S and it is not abnormal that F holds after the event E following S, then F holds also after the event E.

The following locally stratified program *YSP* formalizes the above statements. A similar formalization is in a paper by Apt and Bezem [1].

1. $\mathit{event}(\mathit{load}) \leftarrow$ 2. $\mathit{event}(\mathit{shoot}) \leftarrow$ 3. $\mathit{event}(\mathit{wait}) \leftarrow$

4. $\mathit{fact}(\mathit{alive}) \leftarrow$ 5. $\mathit{fact}(\mathit{dead}) \leftarrow$ 6. $\mathit{fact}(\mathit{loaded}) \leftarrow$

7. $\mathit{holds}(\mathit{alive}, [\,]) \leftarrow$ 8. $\mathit{holds}(\mathit{loaded}, [\mathit{load}|S]) \leftarrow$

9. $\mathit{holds}(\mathit{dead}, [\mathit{shoot}|S]) \leftarrow \mathit{holds}(\mathit{loaded}, S)$

10. $\mathit{ab}(\mathit{alive}, \mathit{shoot}, S) \leftarrow \mathit{holds}(\mathit{loaded}, S)$

11. $\mathit{holds}(F, [E|S]) \leftarrow \mathit{fact}(F),\ \mathit{event}(E),\ \mathit{holds}(F, S),\ \neg\, \mathit{ab}(F, E, S)$

12. $\mathit{append}([\,], Ys, Ys) \leftarrow$ 13. $\mathit{append}([A|Xs], Ys, [A|Zs]) \leftarrow \mathit{append}(Xs, Ys, Zs)$

By applying SLDNF-resolution [38], Apt and Bezem showed that $\mathit{holds}(\mathit{dead}, [\mathit{shoot}, \mathit{wait}, \mathit{load}])$ is true in the perfect model of program *YSP*. Now we consider a property Γ which *cannot* be shown by SLDNF-resolution (see [58]):

$$\Gamma \;\equiv\; \forall S\,(\mathit{holds}(\mathit{dead}, S) \;\rightarrow\; \exists S0, S1, S2, S'\,(\mathit{append}(S2, [\mathit{shoot}|S1], S'),\, \mathit{append}(S', [\mathit{load}|S0], S)))$$

Property Γ means that the fact that the person is *dead* in the current situation S implies that in the past there was a *load* event followed, possibly not immediately, by a *shoot* event. Thus, since time progresses 'to the left', S is a list of events of the form: $[\ldots, \mathit{shoot}, \ldots, \mathit{load}, \ldots]$.

In the first step of our two step verification method (see [58, Section 2.3]), we apply the Lloyd-Topor transformation [38, page 113] starting from the statement: $g \leftarrow \Gamma$ (where g is a new predicate name) and we derive the following clauses:

14. $g \leftarrow \neg new1$

15. $new1 \leftarrow \mathit{holds}(\mathit{dead}, S),\ \neg new2(S)$

16. $new2(S) \leftarrow \mathit{append}(S2, [\mathit{shoot}|S1], S'),\ \mathit{append}(S', [\mathit{load}|S0], S)$

At the end of the second step, after a few iterations of the unfold-definition-folding strategy and after the deletion of all definitions of predicates which are not required by g, we are left with the single clause: $g \leftarrow$. Details can be found in [58].

Since g holds in the (perfect model of the) final program, we have that property Γ holds in the (perfect model of the) final program. Thus, Γ holds also in the initial program made out of clauses 1–13.

Much more recently we have explored some verification techniques based on the transformation of *constrained Horn clauses,* also in the case of imperative and functional programs [19] and in the case of business processes (see, for instance, [17]). This recent work has been done in cooperation with Emanuele De Angelis and Fabio Fioravanti. They also have been working and still work in the implementation and development of an automatic transformation and verification tool [18], which was originally set up by Ornella Aioni and Maurizio Proietti.

8 Future Developments

Reviewing my research activity when writing this paper, I realized that many topics and issues would need a more accurate analysis and study. It would be difficult to list them all, but I have been encouraged to mention at least some of them. I hope that these suggestions may be useful for researchers in the field and they may find these suggestions of some interest.

Concerning the theory of combinators and WCL presented in Section 1, one should note that the combinator $X \equiv B(B(BB)B)(BB)$ we have presented has parentheses and one could consider to construct a B-combinator, call it $\widetilde{B}$, which places those parentheses in a sequence of seven B's, so that $\widetilde{B}BBBBBBB >^* B(B(BB)B)(BB)$. A routine construction, following [3], shows that $\widetilde{B}$ is, in fact, $B(B(B(BB)B)B)(BB)$. The relation between combinators X and $\widetilde{B}$ could be for the reader a stimulus for studying the process of placing parentheses in a list of variables, that is, the process of constructing a binary tree from the list of its leaves.

One can start by considering, first, the use of regular combinators only. A combinator X is said to be *regular* if its reduction is of the form $Xx_1 \ldots x_n > x_1 t_2 \ldots t_m$, where $t_2, \ldots, t_m$ are terms made out of $x_2, \ldots, x_n$ only. A particular regular combinator for placing parentheses is, indeed, B. Similarly, one could study the permutative and duplicative properties of the regular combinators C (defined by $Cxyz > xzy$) and W (defined by $Wxy > xyy$) and other regular (or non-regular) combinators. This study will improve the results reported in the classical book by Curry and Feys [14, Chapter 5].

For Section 2 one could develop the techniques presented in [49]. Those developments can be useful in the area of Term Rewriting Systems for constructing terms with infinite behaviour.

For the issues considered in Section 3 on Program Transformation, it will be important to investigate how to invent the multiplication operation, having at our disposal in the initial program version only the addition operation. Generalizations of various kinds can be suggested as we have done in this paper, but an interesting technique would be the one based on the idea of deriving multiplication as the *iteration* of additions. Then, in an analogous way, exponentiation can be invented as the iteration of multiplications, thus allowing us to derive even more efficient programs. The idea of iteration can hopefully be generated by mechanically analyzing the m-dags constructed by unfolding and looking at repeated patterns.

For Sections 4 and 5, it could be important to mechanize the techniques we have presented there, and in particular those for finding the suitable tuples of functions and suitable lambda-abstractions via the analysis of: (i) cuts and pebble games in the m-dags, and (ii) subexpression mismatchings, respectively.

For Section 6 one can provide an implementation of the functional language with communications we have proposed so that one can execute programs written in that language. One may also: (i) automate the process of adding communications to functional programs for improving their efficiency by making use of the properties of the functions to be evaluated, and (ii) automate the reasoning on the modal theories presented in [59] in which one can prove correctness of those communications. Thus, one will have a machine-checked proof of correctness of the communications which have been added.

For Section 7 a possible project is to construct a transformation system of logic programs with goals as arguments in which: (i) one can run the programs according to the operational semantics we have defined in our paper [56], and (ii) one can apply the various transformation rules (definition introduction, unfolding, folding, goal replacement) we have listed in that paper.

9 Acknowledgements

My gratitude goes to the various people who taught me Computer Science and among them, Paolo Ercoli, Rod Burstall, Robin Milner, Gordon Plotkin, John Reynolds, Alan Robinson, and Leslie Valiant. I am also grateful to my colleagues and students. In particular, I would like to thank Anna Labella, Andrzej Skowron, Maurizio Proietti, Valerio Senni, Sophie Renault, Fabio Fioravanti, and Emanuele De Angelis. A very special thank goes to Maurizio, who for many years has been 'so devoted to me, so patient, so zealous', as John Henry Newman said of his friend Ambrose St. John [43, page 190]. Maurizio has been for me an unvaluable source of inspiration and strength.

Many thanks to Andrei Nemytykh for inviting me to write this paper and for his comments, and Maurizio Proietti and Laurent Fribourg for their friendly help and encouragement.

References

[1] K. R. Apt & M. Bezem (1991): *Acyclic Programs. New Generation Computing* 9, pp. 335–363, doi:10.1007/BF03037168.

[2] H. P. Barendregt (1984): *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam, doi:10.2307/2274112.

[3] C. Batini & A. Pettorossi (1975): *On Subrecursiveness in Weak Combinatory Logic.* In: *Proceedings of the Symposium λ-Calculus and Computer Science Theory*, Lecture Notes in Computer Science 37, Springer-Verlag, pp. 297–311, doi:10.1007/BFb0029533.

[4] F. L. Bauer & H. Wössner (1982): *Algorithmic Language and Program Development.* Springer-Verlag, doi:10.1007/978-3-642-61807-9.

[5] R. S. Bird (1984): *The Promotion and Accumulation Strategies in Transformational Programming. ACM Toplas* 6(4), pp. 487–504, doi:10.1145/1780.1781.

[6] R. S. Bird (1984): *Using Circular Programs to Eliminate Multiple Traversal of Data. Acta Informatica* 21, pp. 239–250, doi:10.1007/BF00264249.

[7] R. S. Boyer & J. S. Moore (1975): *Proving Theorems About Lisp Functions. Journal of the ACM* 22(1), pp. 129–144, doi:10.1145/321864.321875.

[8] R. Burstall (1972): *An Algebraic Description of Programs with Assertions, Verification and Simulation.* In: *Proc. ACM Conference on Proving Assertions about Programs*, ACM, New York, NY, USA, pp. 7–14, doi:10.1145/800235.807068.

[9] R. M. Burstall (1977): *Design Considerations for a Functional Programming Language.* In: *Proc. Infotech State of the Art Conference "The Software Revolution", Copenhagen, Denmark*, pp. 47–57.

[10] R. M. Burstall & J. Darlington (1977): *A Transformation System for Developing Recursive Programs. Journal of the ACM* 24(1), pp. 44–67, doi:10.1145/321992.321996.

[11] R. M. Burstall, D.B. MacQueen & G. H. Sannella (1980): *Hope: An Experimental Applicative Language.* In: *Conference Record of the 1980 LISP Conference, Stanford University, Stanford, Ca, USA*, pp. 136–143, doi:10.1145/800087.802799.

[12] K. L. Clark (1978): *Negation as Failure.* In H. Gallaire & J. Minker, editors: *Logic and Data Bases*, Plenum Press, New York, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11.

[13] W. F. Clocksin & C. S. Mellish (1984): *Programming in Prolog*, Second edition. Springer-Verlag, New York, doi:10.1007/978-3-642-96873-0.

[14] H. B. Curry & R. Feys (1974): *Combinatory Logic.* North-Holland.

[15] J. Darlington (1978): *A Synthesis of Several Sorting Algorithms.* *Acta Informatica* 11, pp. 1–30, doi:10.1007/BF00264597.

[16] J. Darlington (1981): *An Experimental Program Transformation System.* *Artificial Intelligence* 16, pp. 1–46, doi:10.1016/0004-3702(81)90014-X.

[17] E. De Angelis, F. Fioravanti, M. C. Meo, A. Pettorossi & M. Proietti (2017): *Verification of Time-Aware Business Processes using Constrained Horn Clauses.* In: *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation* (*LOPSTR 2016*), Lecture Notes in Computer Science 10184, Springer, pp. 38–55, doi:10.1007/978-3-319-63139-4_3.

[18] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations.* In: *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Computer Science 8413, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47. Available at: http://www.map.uniroma2.it/VeriMAP.

[19] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2017): *Semantics-based generation of verification conditions via program specialization.* *Science of Computer Programming* 147, pp. 78–108, doi:10.1016/j.scico.2016.11.002. Selected and Extended papers from the Int. Symp. on Principles and Practice of Declarative Programming 2015.

[20] N. Dershowitz (1987): *Termination of rewriting.* *Journal of Symbolic Computation* 3(1-2), pp. 69–116, doi:10.1016/S0747-7171(87)80022-6.

[21] E. W. Dijkstra (1971): *A Short Introduction to the Art of Programming.* Technical Report, EWD 316.

[22] E. W. Dijkstra (1982): *Selected Writing on Computing: A Personal Perspective.* Springer-Verlag, New York, Heidelberg, Berlin, doi:10.1007/978-1-4612-5695-3.

[23] M. C. Er (1983): *An iterative solution to the generalized Towers of Hanoi problems.* *BIT* 23, pp. 295–302, doi:10.1007/BF01934458.

[24] P. Flajolet & J.-M. Steyaert (1974): *On Sets Having Only Hard Subsets.* In J. Loeckx, editor: *2nd ICALP, Automata, Languages and Programming*, Lecture Notes in Computer Science 14, Springer, pp. 446–457, doi:10.1007/978-3-662-21545-6_34.

[25] D. Gelernter & N. Carriero (1992): *Coordination Languages and their Significance.* *Communications of the ACM* 35(2), pp. 97–107, doi:10.1145/129630.129635.

[26] P. J. Hayes (1977): *A note on the Towers of Hanoi problem.* *The Computer Journal* 20(3), pp. 282–285, doi:10.1093/comjnl/20.3.282.

[27] J. R. Hindley, B. Lercher & J. P. Seldin (1975): *Introduzione alla Logica Combinatoria.* Serie di Logica Matematica, Boringhieri. (In Italian).

[28] J. R. Hindley & J. P. Seldin (1986): *Introduction to Combinators and λ-Calculus.* London Mathematical Society, Cambridge University Press, doi:10.1007/BF00047236.

[29] C.A.R. Hoare (1978): *Communicating Sequential Processes.* *CACM* 21(8), pp. 666–677, doi:10.1145/359576.359585.

[30] J. E. Hopcroft & J. D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, doi:10.1145/568438.568455.

[31] R. J. M. Hughes (1986): *A novel representation of lists and its application to the function "reverse".* *Info. Proc. Lett.* 22, pp. 141–144, doi:10.1016/0020-0190(86)90059-1.

[32] V. E. Itkin & Z. Zwienogrodsky (1971): *On equivalence of program schemata.* *Journ. Comp. Syst. Sci.* 5.

[33] S. Kasangian, A. Labella & A. Pettorossi (1990): *Observers, experiments, and agents: A comprehensive approach to parallelism.* In I. Guessarian, editor: *Semantics of Systems of Concurrent Processes.*

LITP Spring School, Lecture Notes in Computer Science 469, Springer-Verlag, pp. 375–406, doi:10.1007/3-540-53479-2_16.

[34] G. M. Kelly (1982): *Basic Concepts of Enriched Category Theory.* Cambridge University Press, Cambridge.

[35] R. A. Kowalski (1979): *Logic for Problem Solving.* North Holland. Available at: www.doc.ic.ac.uk/~rak/papers/LogicForProblemSolving.pdf.

[36] A. Labella & A. Pettorossi (1985): *Categorical Models for Handshaking Communications. Fundamenta Informaticae. Series IV.* VIII(3-4), pp. 322–357.

[37] S. S. Lavrov (1961): *Economy of memory in closed operator schemes. U.S.S.R. Computat. Math. and Math. Physics*, pp. 810–828.

[38] J. W. Lloyd (1987): *Foundations of Logic Programming.* Springer-Verlag, Berlin, doi:10.1007/978-3-642-83189-8. Second Edition.

[39] V. V. Martynuk (1965): *On the analysis of control-flow graphs for a program scheme. Journ. Comp. Math. and Math. Phys.* 5,2.

[40] E. Mendelson (1987): *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, USA, doi:10.2307/2274877. Third Edition.

[41] J. Miller & S. Brown (1966): *An algorithm for evaluation of remote terms in a linear recurrence sequence. The Computer Journal* 9, pp. 188–190, doi:10.1093/comjnl/9.2.188.

[42] R. Milner (1989): *Communication and Concurrency.* Prentice Hall, doi:10.5555/534666.

[43] J. H. Newman (2001): *Apologia Pro Vita Sua.* Maisie Ward (ed.), Sheed and Ward, London.

[44] M. S. Paterson & C. E. Hewitt (1970): *Comparative Schematology.* In: *Conference on Concurrent Systems and Parallel Computation Project MAC, Woods Hole, Mass., USA*, pp. 119–127. Available at: https://dl.acm.org/doi/pdf/10.1145/1344551.1344563.

[45] A. Pettorossi (1971): *Ottimizzazione di un Collegamento per Trasmissione di Dati Mediante Simulazione Numerica.* Laurea Thesis (in Italian). University of Rome, Italy. Available on request to the author.

[46] A. Pettorossi (1972): *Automatic Derivation of Control Flow Graphs of Fortran Programs.* Master Thesis (in Italian). Original title: "Generazione Automatica del Grafo di Flusso del Controllo per un Programma di Calcolo Scritto in Fortran". University of Rome, Italy. Available on request to the author.

[47] A. Pettorossi (1978): *Improving memory utilization in transforming programs.* In: *Proc. Mathematical Foundations of Computer Science 1978, Zakopane (Poland)*, Lecture Notes in Computer Science 64, Springer-Verlag, pp. 416–425, doi:10.1007/3-540-08921-7_89.

[48] A. Pettorossi (1979): *On the Definition of Hierarchies of Infinite Sequential Computations.* In Lothar Budach, editor: *Fundamentals of Computation Theory, FCT'79*, Akademic-Verlag, Berlin, pp. 335–341. Available on request to the author.

[49] A. Pettorossi (1980): *Synthesis of Subtree Rewriting Systems Behaviour by Solving Equations.* In: *Proc. 5ème Colloque de Lille (France) on "Les Arbres en Algèbre et en Programmation"*, U.E.R. I.E.E.A. BP 36, Université de Lille I, 59655 Villeneuve d'Ascq Cedex, France, pp. 63–74. Available on request to the author.

[50] A. Pettorossi (1981): *Comparing and Putting Together Recursive Path Orderings, Simplification Orderings, and Non-Ascending Property for Termination Proofs of Term Rewriting Systems.* In: *Proc. ICALP 1981, Haifa (Israel)*, Lecture Notes in Computer Science 115, Springer-Verlag, pp. 432–447, doi:10.1007/3-540-10843-2_35.

[51] A. Pettorossi (1984): *Methodologies for Transformations and Memoing in Applicative Languages.* Ph.D. thesis, Edinburgh University, Edinburgh, Scotland. Available at: https://era.ed.ac.uk/handle/1842/15643.

[52] A. Pettorossi (1984): *A Powerful Strategy for Deriving Efficient Programs by Transformation.* In: *ACM Symposium on Lisp and Functional Programming*, ACM Press, pp. 273–281, doi:`10.1145/800055.802044`.

[53] A. Pettorossi (1985): *Towers of Hanoi Problems: Deriving Iterative Solutions by Program Transformation. BIT* 25, pp. 327–334, doi:`10.1007/BF01934378`.

[54] A. Pettorossi (1987): *Derivation of Efficient Programs For Computing Sequences of Actions. Theoretical Computer Science* 53, pp. 151–167, doi:`10.1016/0304-3975(87)90030-2`.

[55] A. Pettorossi & R. M. Burstall (1982): *Deriving Very Efficient Algorithms for Evaluating Linear Recurrence Relations Using the Program Transformation Technique. Acta Informatica* 18, pp. 181–206, doi:`10.1007/BF00264438`.

[56] A. Pettorossi & M. Proietti (2000): *Transformation Rules for Logic Programs with Goals as Arguments.* In A. Bossi, editor: *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation* (*LOPSTR '99*)*, Venezia, Italy*, Lecture Notes in Computer Science 1817, Springer-Verlag, Berlin, pp. 177–196, doi:`10.1007/10720327_11`.

[57] A. Pettorossi & M. Proietti (2002): *The List Introduction Strategy for the Derivation of Logic Programs. Formal Aspects of Computing* 13(3-5), pp. 233–251, doi:`10.1007/s001650200011`.

[58] A. Pettorossi & M. Proietti (2002): *Program Derivation = Rules + Strategies.* In A. Kakas & F. Sadri, editors: *Computational Logic: Logic Programming and Beyond* (*Essays in honour of Bob Kowalski, Part I*), Lecture Notes in Computer Science 2407, Springer-Verlag, pp. 273–309, doi:`10.1007/3-540-45628-7_12`.

[59] A. Pettorossi & A. Skowron (1983): *Complete Modal Theories for Verifying Communicating Agents Behaviour in Recursive Equations Programs.* Internal Report CSR-128-83, University of Edinburgh, Edinburgh, Scotland. Available on request to the authors.

[60] A. Pettorossi & A. Skowron (1985): *A methodology for improving parallel programs by adding communications.* In: *Computation Theory, SCT 1984*, Lecture Notes in Computer Science 280, Springer-Verlag, Berlin, pp. 228–250, doi:`10.1007/3-540-16066-3_20`.

[61] A. Pettorossi & A. Skowron (1985): *A System for Developing Distributed Communicating Programs.* In M. Feilmeier, G. Joubert & U. Schendel, editors: *International Conference 'Parallel Computing 85'*, North-Holland, pp. 241–246. Available on request to the authors.

[62] A. Pettorossi & A. Skowron (1989): *The Lambda Abstraction Strategy for Program Derivation. Fundamenta Informaticae* XII(4), pp. 541–561. Available on request to the authors.

[63] Alberto Pettorossi & Maurizio Proietti (2004): *Transformations of Logic Programs with Goals as Arguments. Theory Pract. Log. Program.* 4(4), pp. 495–537, doi:`10.1017/S147106840400198X`.

[64] M. Proietti & A. Pettorossi (1990): *Synthesis of Eureka Predicates for Developing Logic Programs.* In N. D. Jones, editor: *Third European Symposium on Programming, ESOP '90*, Lecture Notes in Computer Science 432, Springer-Verlag, pp. 306–325, doi:`10.1007/3-540-52592-0_71`.

[65] M. Proietti & A. Pettorossi (1991): *Unfolding-Definition-Folding, in this Order, for Avoiding Unnecessary Variables in Logic Programs.* In J. Małuszyński & M. Wirsing, editors: *Third International Symposium on Programming Language Implementation and Logic Programming, PLILP '91*, Lecture Notes in Computer Science 528, Springer-Verlag, pp. 347–358, doi:`10.1007/3-540-54444-5_111`.

[66] H. Rogers (1967): *Theory of Recursive Functions and Effective Computability.* McGraw-Hill.

[67] L. S. Sterling & E. Shapiro (1994): *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts. Second Edition.

[68] H. Tamaki & T. Sato (1984): *Unfold/Fold Transformation of Logic Programs.* In S.-Å. Tärnlund, editor: *Proceedings of the Second International Conference on Logic Programming, ICLP '84*, Uppsala University, Uppsala, Sweden, pp. 127–138.

[69] S. A. Walker & H. R. Strong (1973): *Characterization of Flowchartable Recursions*. *Journal of Computer and System Sciences* 7(4), pp. 404–447, doi:10.1016/S0022-0000(73)80032-7.

[70] D. H. D. Warren (1977): *Implementing Prolog – Compiling Predicate Logic Programs*. Research Report 39 & 40, Department of Artificial Intelligence, University of Edinburgh, Scotland.

[71] D. H. D. Warren (1983): *An Abstract Prolog Instruction Set*. Technical Report 309, SRI International.

[72] D. H. D. Warren, L. M. Pereira & F. Pereira (1977): *Prolog - the language and its implementation compared with Lisp*. *SIGART Newsl.* 64, pp. 109–115, doi:10.1145/872736.806939.

[73] G. Winskel (1984): *Synchronization Trees*. *Theoretical Computer Science* 34(1-2), pp. 33–82, doi:10.1016/0304-3975(84)90112-9.

[74] Y. I. Yanov (1960): *The Logical Schemes of Algorithms*. *Problems of Cybernetics* 1, pp. 82–140. English translation.