FACTUAL KNOWLEDGE FOR DEVELOPING CONCURRENT PROGRAMS

Alberto Pettorossi
IASI-CNR
Viale Manzoni 30
00185 Roma (Italy)

Andrzej Skowron
Institute of Mathematics
Warsaw University
PKiN IX p.907
00-901 Warsaw (Poland)
University of North Carolina at Charlotte
Computer Science Department
Charlotte, NC 28223 (USA)

ABSTRACT

We propose a system for the derivation of algorithms which allows us to use "factual knowledge" for the development of concurrent programs. From preliminary program versions the system can derive new versions which have higher performances and can be evaluated by communicating agents in a parallel architecture. The knowledge about the facts or properties of the programs is also used for the improvement of the system itself.

I THE STRUCTURE OF THE SYSTEM

We present some preliminary ideas for designing an interactive system which can be used for algorithm derivation. The components of the system are best understood by relating them to the Burstall-Darlington methodology [2]. In that approach the programmer is first asked to produce a correct version of the program, and then he has to care about efficiency issues. He then improves that preliminary version by performing "eureka steps" and applying correctness preserving transformation rules [2] (maybe with the help of a machine for rule application). We generalize those concepts and we suggest the structure of a system (depicted in figure 1) where: i) the mathematical descriptions of the problems generalize the first correct program versions, ii) the factual knowledge [1] generalizes the eureka steps, and iii) the Logical System generalizes the machine for the application of the transformation rules.

For point i) we assume that the descriptions of the problems are constructive, that is, they correspond to executable functional programs. We also assume that we may have some constraints on their executions as, for instance, on the number of computing agents and their topological connections, on the space and time resources, etc.
For point ii) we consider that during the development process the programmer acquires (maybe in an incremental way) the knowledge of some facts about the functions to be computed or the behaviour of the computing agents. Those new facts may or may not be logical consequences of the knowledge already available from the descriptions of the problems themselves.
The Logical System of point iii) is more powerful than the traditional matching procedure, which applies the transformation rules and verifies the related validating conditions [3]. It is basically made out of three modules: - a Knowledge Base in which new facts are incrementally added by the programmer or the system itself, - an Analyzer-Synthesizer which checks the correctness of the acquired facts and draws the logical consequences from the currently available Knowledge Base, and - a Translation Algorithm which uses the checked facts for the (semi)automatic derivation of new and more efficient versions of the programs.

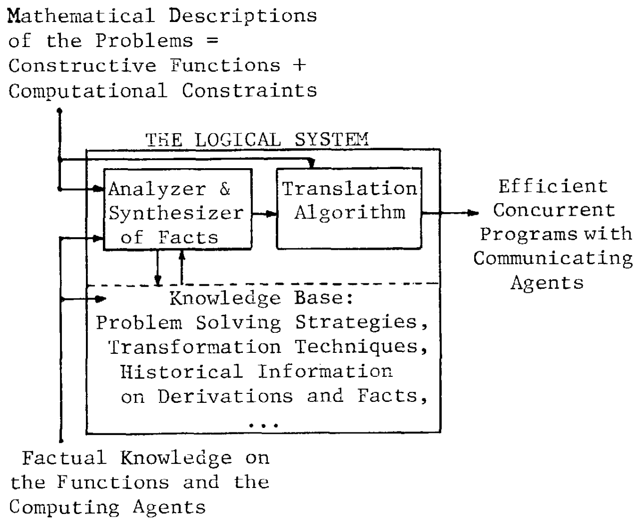

Figure 1. The structure of the general system.

The Analyzer-Synthesizer module also provides an input to the Knowledge Base. It activates a "learning process" by updating the historical information about the derivations of the algorithms already performed or the effectivity of the strategies which have been used. That information may be very valuable for the future developments of similar algorithms with constraints. Related ideas on the structure of a program development system were suggested in [7].

The general system we have presented is also capable of generating approximation algorithms for solving problems which may require exponential resources for an exact solution. In that case, in fact, the knowledge of the constraints may force the translation procedure to derive only program versions which use polynomial time or space. We will not discuss this point here.

As a first step towards the realization of the general system we consider a specific instance of it, which is suited for dealing with a class of simple problems of the kind studied in [2]. We assume that the solutions of those problems can be expressed as

a set of recursive equations. In that case, in fact, some strategies for developing programs have been already analyzed in the literature (see, for instance, the divide-and-conquer strategy), and the programmer can easily provide factual knowledge from his past experience or through simple considerations.
Figure 2 shows the structure of the particular instance of the system we consider in what follows.

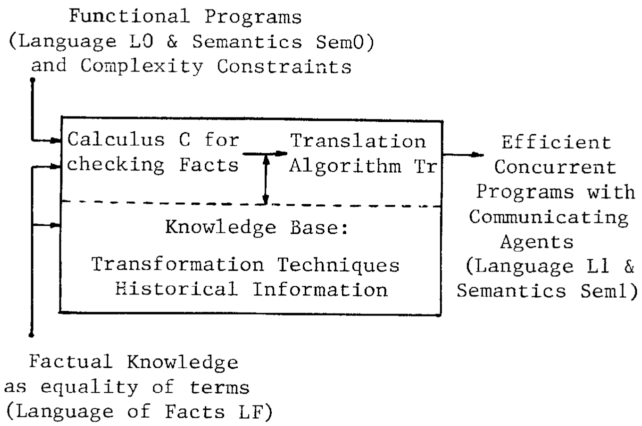

Figure 2. A Calculus + Translation System.

The Logical System is essentially made out of three parts:
- a Calculus based on a Theorem Prover which uses symbolic evaluation and induction for checking the validity of the facts about the programs,
- a Translation Algorithm, which translates checked facts into suitable communications among computing agents, so that the derived program versions may achieve the desired performance,
- a Knowledge Base, which has a dictionary of transformation rules and maintains the historical information about the program derivations already performed.

Our system extends the Burstall-Darlington approach in the following respects:
- it allows for the development of distributed and communicating algorithms from program specifications;
- the specification language (or the one of the initial program versions) may be different from the language of the derived programs, and therefore the development of the algorithms is made easier;
- the application of the transformation rules makes the new versions of the programs provably more efficient than the old ones;
- the requirements for the desired complexity bounds are explicitly considered, and the system tries to meet them by applying transformation techniques which turned out to be successful in previous derivations.

We assume that the factual knowledge about the programs to be developed is expressed as equality of terms. This notion will be formally defined later. The example we will give in the following Section will clarify the ideas. We will not deal here with the question on how the Knowldge Base is updated and how new transformation techniques can be derived from old ones.

II PROGRAM DERIVATION USING FACTUAL KNOWLEDGE

Let us present the basic ideas of the approach we suggest, through an example. We consider the N Chinese Rings Problem. It is a generalization of a puzzle described in [4, p.63], and it is often analyzed in Artificial Intelligence papers.
N rings, numbered from 1 to N, are placed on a stick. We are asked to remove all of them from the stick by a sequence of moves. We have to comply to the following rule, where k (or $\underline{k}$) denotes the move which takes away from (or puts back to) the stick the ring k:
for k=2,...,N moves k or $\underline{k}$ can be performed iff rings 1,...,k-2 are not on the stick and ring k-1 is on the stick.
clear(k) computes the sequence of moves which removes rings 1,...,k from the stick, if initially they are all on the stick. Conversely, put(k) computes the moves for putting back rings 1,...,k on the stick, if initially they are not on the stick.
clear(N), recursively defined by the following program P written in a language called L0 (defined below), solves the puzzle.

$$P: \begin{cases} \text{clear}(1)=1, \quad \text{clear}(2)=2:1, & \\ \text{clear}(k+2)=\text{clear}(k):k+2:\text{put}(k):\text{clear}(k+1) & k>0 \\ \text{put}(1)=\underline{1}, \quad \text{put}(2)=\underline{1}:\underline{2}, & \\ \text{put}(k+2)=\text{put}(k+1):\overline{\text{clear}(k)}:\underline{k+2}:\text{put}(k) & k>0 \end{cases}$$

Suppose that we are also required to obtain a linear time algorithm, i.e., an algorithm which evokes a linear number of recursive calls. Now factual knowledge can be used for developing the above program with the given complexity constraints.
Let us denote by $\underline{s}$ the sequence of moves $\underline{mp}...\underline{m1}$ for any sequence s=m1...mp. We can supply to our system the following fact F1: put(k)=$\underline{\text{clear(k)}}$.
The calculus C (later defined) can check it using induction. The cases for k=1 and 2 are obvious, and for the recursive case we have: put(k+2)=$\underline{\text{clear(k+1)}:\text{put(k)}:\underline{\text{k+2}}:\text{clear(k)}}$=$\underline{\text{clear(k+2)}}$ because $\underline{\underline{s}}$=s.

Once the fact F1 has been accepted, the translation algorithm Tr produces from it the following program version:

$$P1: \begin{cases} \text{clear}(1)=1, \quad \text{clear}(2)=2:1, \\ \text{clear}(k+2)=s:k+2:\underline{s}:\text{clear}(k+1) \ \underline{\text{where}} \ s=\text{clear}(k) \end{cases}$$

This program is more efficient than program P because a smaller number of recursive calls is generated. However, we have not derived yet the required linear algorithm. Notice, in fact, that each call of clear(k+2) requires the value of the left son call clear(k) and the right son call clear(k+1) (The order of the calls we used, is the left-to-right one, after substituting clear(k) for s in the expression of clear(k+2)).

Now a new fact about the program P1 (or P) can be discovered by symbolic evaluation:
F2: clear(k+2)]0=clear(k+2)]11 for k>0.
Later on we will give a formal definition of the language LF of facts. For the time being it is enough to remark that by clear(k+2)]0 we denote the left son call of clear(k+2) and by clear(k+2)]11 we denote the right son call of the right son call of clear(k+2).
Fact F2 is obvious because both sides are equal to

clear(k) (as it will be checked by our calculus using symbolic evaluation). From fact F2 the translation algorithm Tr will derive the following program:

```
    [clear(1)=1,      clear(2)=2:1,
P2: |clear(k+2)=(s:k+2:s:clear(k+1)(1 comm ℓ)
    [           where s=clear(k)(ε comm ℓ)) decl ℓ
```

The informal explanation of the communication annotations added by Tr is as follows.
We assume that recursively defined functions are evaluated by a set of computing agents, i.e., triples of the form <agentname,message>::expression. Messages are the local memories of the agents and expressions are their tasks, that is, what they have to evaluate. Agents dynamically create new agents while the computation progresses. In particular in our program P2 the agent <x,m>::clear(k+2) generates the two agents <x0,m0>::clear(k) and <x1,m1>::clear(k+1).

The naming convention for the agents is the following: the father agent with name x generates the sons with names x0,...,xk,..., each of which is associated (in the left to right order) to a recursive call occurring in the corresponding program equation.

By e comm ℓ we mean that a memory location ℓ is kept during the evaluation of e. Let f(...)ε denote the call f(...) itself, and let f(...)js recursively denote the s-son call of the j-son call of f(...) for $0 \leqslant j \leqslant k$, s ε {0,...,k}*. By f(...)(s comm ℓ) we mean that the s-son of the agent evaluating f(...) may look at the value in the location ℓ to know the result of its own computation. That s-son agent will write its result in the location ℓ, if it did not find any value there. It can easily be seen that by writing and reading the location ℓ the computation time may be shortened. To make sure that unneeded agents are not generated, in the language L1 we have also the annotations of the form: s read ℓ and s write ℓ. The first one forces the s-son to wait for the value of its expression to be written in the location ℓ by another agent.
Conversely s write ℓ forces the s-son to write its final result in the location ℓ and it will never try to read ℓ.
The following figure 3 shows the use of the location ℓ for program P2.

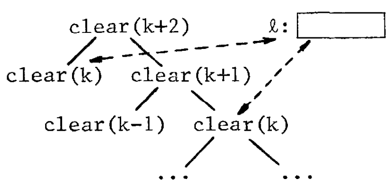

Figure 3. Using the location ℓ for the fact F2.

The following program P3 generates a linear number of agents only, and it meets the desired efficiency requirements for a linear algorithm.

```
    [clear(1)=1,      clear(2)=2:1,
P3: |clear(k+2)=(s:k+2:s:clear(k+1)(1 write ℓ)
    [           where s=clear(k)(ε read ℓ)) decl ℓ
```

One more fact can be discovered about the program P0:
F3: clear(k+2)]001=clear(k+2)]010.

Its incorporation into program P2 using two memory locations produces:

```
     [clear(1)=1,      clear(2)=2:1,
P21: |clear(k+2)=(s:k+2:s:clear(k+1)(1 comm ℓ1)
     |           where s=clear(k)(ε comm ℓ1)(01 comm ℓ2)
     [                       (10 comm ℓ2)) decl ℓ1,ℓ2
```

Fact F3 speeds up the computation of P2 because during the evaluation of clear(k+5) the repeated evaluation of clear(k) may be avoided. However, there is no point in incorporating Fact F3 into P3 because the agents with names x001 and x010 will not be generated.

III FACTS AND SEMANTICS OF CONCURRENT PROGRAMS

In this Section we will give the definition of the language L1 in which programs with communication annotations are written, and its semantics Sem1.
We will also define the language LF of facts and the Calculus C, while the details of the Translation Algorithm Tr will be left to the reader. The definition of the language L0 and its semantics Sem0 can be derived from the language L1 (and Sem1) if one does not take into consideration the communication annotations. Those definitions allow us to formally analyze some properties of our system and to state some basic results.

Let start off by introducing the following preliminary notions.
An expression e ε Exp in L1 is defined by:
e::=n|x|g(e,...)|f(e,...)|e(s ann ℓ)|e[z] where z=e'
where n ε Constants, x ε Variables, g ε Basic-Functions, f ε Recursive-Functions, s ε {0,...,k}*, ℓ ε Locations, and ann ε {comm, read, write}.
A program P in L1 is a set of recursive equations each of which is of the form: f(e,...)=n (base case) or f(e,...)=e1 (recursive case) where f occurs in e1 and e1 is of the form: e or e decl ℓ.
For simplicity we assume that no nested recursive calls of f occur in e1 and there is one recursive case only. It is possible, however, to extend our results releasing those hypotheses.

In order to define the semantics Sem1 of L1 we need first to introduce the notion of agents.
A (computing) agent is a triple of the form:
<agn,msg>::e where agn ε Agn, msg ε Msg, and e ε CExp.
Agn is a set of agentnames agn defined by:
agn::=ε|agn 0|...|agn k.
Msg is a set of messages such that: i) E (the empty message) ε Msg, and ii) R←em←W ε Msg where: em is the empty elementary message φ or it is a constant elementary message n ε Constants, and R and W are the sets of the names of the agents which read and write (respectively) the message em.
CExp is a set of closed expressions defined as in L1 with the additional case: .agn (.agn stands for the value of the expression of the agent agn).

The semantics Sem1 is defined in an operational way by assigning to each program in L1 a set of conditional rewriting rules for agents. Those rules tell us how to produce new sets of agents from old ones. They are of the form:
set of agents <= set of agents if condition
and they can be applied in a parallel way by rewriting non-conflicting subsets of agents [5].

Let a configuration be a (finite) set of agents and CON be the set of all configurations.
By r(x1,...,xk) we denote a rule-schema r:

lh <= rh if cond

in which x1,...,xk are the only (meta)variable occurrences. Given the constants a1,...,ak,

r(a1,...,ak) or r or lh <= rh if cond

denotes a concrete instance of r which can be derived by substituting a1,...,ak for x1,...,xk.

Let c,c' $\in$ CON, and r be the rule-schema of the form given above. Let us define the (one-step) transition relation of r as follows:

c $\xrightarrow{r}$ c' holds iff cond is true and lh $\subseteq$ c and c'=(c-lh) $\cup$ rh .
The transition relation corresponding to a sequence s=r1...rk of instances of rule-schemas is defined as the composition of the transition relations $\xrightarrow{r1}$,...,$\xrightarrow{rk}$ and it is denoted by $\xrightarrow{s}$.
Let Sem1(P) denote the rule-schemas associated to P for any program P in L1 (They will be introduced bellow). The (one-step) transition relation of a program P in L1 (written as $\xrightarrow{P}$) defines the semantics of P, and it is specified as follows:

c $\xrightarrow{P}$ c' holds iff there exists a non-empty finite sequence s of instances (derived by the same substitution) of rule-schemas in Sem1(P), s.t. for an arbitrary permutation s' of s we have:
c $\xrightarrow{s'}$ c'.
The condition on the permutations of s is the one which is usually considered for expressing that the atomic transitions r1,...,rk refer to non-conflicting subsets of agents, and therefore they can be performed in parallel. More details are given in a companion paper [6] where we studied the behaviour of sets of communicating agents which concurrently evaluate functional programs.

Therefore for computing, for instance, the value of f(...) where f is defined by a program P in L1, we consider an initial computing agent <ε,E>::f(...), and by applying the rewriting rules we derive new agents from old ones. When we eventually obtain the agent <ε,m>::n where n $\in$ Constants, we say that the value of f(...) is n.

Given a program in L1, Sem1 produces the rewriting rule-schemas for agents as follows:

1. Generation of sons with communications

```
f(e0,...ep)=g(...,f(e,...)(s comm ℓ),...,f(e',..),..,
                  0                      i
              f(e1,..)(s1 write ℓ),...,
              j
              f(e2,..)(s2 read ℓ),...) decl ℓ
              k
```

produces the rule-schema:

```
{<x,E>::f(e0,..,ep)} <= {<x,xR̄←φ←xW̄>::g(..,.x0,..,
                                  .xi,..,.xj,..,.xk,...),
     <x0,E>::f(e,...),...,<xi,E>::f(e',..),...,
     <xj,E>::f(e1,..),...,<xk,E>::f(e2,...),...}
                        if ∀z ε R̄  ∀y   x ≠ yz
```

where W̄={js | s write ℓ or s comm ℓ occurs in the j-th call},
and R̄={ks | s read ℓ or s comm ℓ occurs in the k-th call}.
xA denotes the set {xa | a $\in$ A}.
The condition of the rule makes it impossible for an agent which has to make a reading communication, to generate new agents (That agent has to wait for the value of its expression to be computed by another agent). As usual, we identify by the numbers 0,...,k,... the son calls in the left-to-right order.

2. Base Cases

```
f(e0,...,ek)=n  produces:
{<x,E>::f(e0,...,ek)} <= {<x,E>::n}
```

3. Values to Fathers

```
{<x,m>::g(...,.xj,...), <xj,m'>::n}  <=
              {<x,m>::g(...,n,...), <xj,m'>::n}
```

4. Writing Communications

```
{<x,R←φ←W>::e,  <xs,m>::n}    <=
         {<x,R←φ←(W-xs)>::e, <xs,m>::n} if xs ε W
```

5. Reading Communications

```
{<x,R←n←W>::e,  <xs,m>::e1}   <=
         {<x,(R-xs)←n←W>::e,  <xs,m>::n} if xs ε R
```

6. Basic Functions Evaluation

```
{<x,m>::g(n1,...)} <= {<x,m>::v}   if v=g(n1,...)
```

The g in the condition is the mathematical function.

7. Initial Agent

For evaluating the expression f(n1,...) the initial configuration is: {<ε,E>::f(n1,...)}.

The where-expressions are not considered by Sem1 because one may get rid of them by substituting the corresponding expressions. However, when applying the generation-of-sons rule, we assume that Sem1 creates the same agent for all substituted occurrences of the same where-expression.

Now, as an example of the definition of Sem1 let us present the evaluation of clear(5). We write {...}---->{==, ag1,...,agk} for denoting that the agents to the left are the ones to the right, except for ag1,...,agk (see also figure 4).
Sem1(P3) contains (besides others) the following rule-schemas:

```
{<x,E>::clear(k+2)}    <=
      {<x,{x0}←φ←{x11}>::.x0:k+2:.x0:.x1,            (r1)
       <x0,E>::clear(k),
       <x1,E>::clear(k+1)} if x≠y11 for any y;
{<x,E>::clear(1)} <= {<x,E>::1};                      (r2)
{<x,E>::clear(2)} <= {<x,E>::2:1};                    (r3)
{<x,{x0}←φ←{x11}>::e,  <x11,m>::n}     <=
        {<x,{x0}←n←{}>::e,  <x11,m>::n};              (r4)
{<x,{x0}←n←{}>::e,  <x0,m>::e1}    <=
        {<x,{}←n←{}>::e,  <x0,m>::n}.                 (r5)
```

The rule-schema r1 comes from the Generation-of-Sons schema, the rule-schemas r2 and r3 from the Base-Cases schema, and r4 and r5 from the Writing and Reading Communications schemas.
The initial agent is <ε,E>::clear(5).

```
        {<ε,E>::clear(5)}
---->   {<ε,{0} ← φ ← {11}>::.0:5:.0:.1,
          <0,E>::clear(3),  <1,E>::clear(4)}
---->   {==, <1,{10} ← φ ← {111}>::.10:4:.10:.11,
             <10,E>::clear(2), <11,E>::clear(3)}
---->   {==, <11,{110} ← φ ← {1111}>::.110:3:.110:.111,
             <110,E>::clear(1),
             <111,E>::clear(2)}
---->...{==, <110,E>::1, <111,E>::2:1}
---->   {==, <1,{10} ← 2:1 ← {}>::.10:4:.10:.11}
---->   {==, <1,{} ← 2:1 ← {}>::.10:4:.10:.11,
             <10,E>::2:1}
```

```
---->...{==, <1,{} ← 2:1 ← {}>::2:1:4·1:2:.11}
---->  {==, <11,{110} ← φ ← {1111}>::1:3:1:.111}
---->  {==, <11,{110} ← φ ← {1111}>::1:3:1:2:1}
---->  {==, <1,{} ← 2:1 ← {}>::2:1:4:1:2:1:3:1:2:1}
---->  {==, <ε,{0} ← 1:3:1:2:1 ← {}>::.0:5:.0:.1}
---->  {==, <ε,{} ← 1:3:1:2:1 ← {}>::.0:5:.0:.1,
            <0,E>::1:3:1:2:1}
---->  {==, <ε,{} ← 1:3:1:2:1 ← {}>::1:3:1:2:1: 5 :
                                      1:2:1:3:1:.1}
---->  {==, <ε,...>::1:3:1:2:1:5:1:2:1:3:1:
                 2:1:4:1:2:1:3:1:2:1}.
```

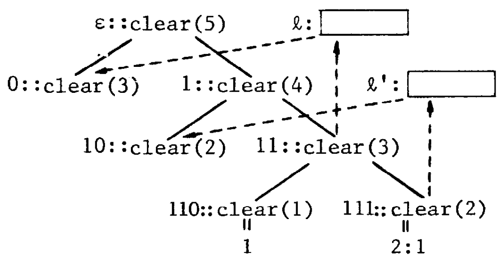

Figure 4. Flow of messages when computing clear(5) using P3.

Notice that the sons agents after sending their values to the fathers, remain in the configurations because they may perform a writing communication. An improved operational semantics may garbage-collect the agents which are no longer needed for computing the final result.

The syntax of the language LF of facts is defined as follows:

e::=...(as in L1 without communication annotations)
    | e]s with s ε {0,1,...,k}*
fact::= f(e,...)]s1 = f(e',...)]s2
    |g1(...,f(...),...)=g2(...,f(...),...)

The Calculus C for checking facts about a given program P will be presented assuming that P has only one recursive case for the defined function f and the facts are of the first form.

A fact e1]s1=e2]s2 is accepted by the Calculus C iff both expressions turn out to be identical (and different from error) after applying the rules of the Basic-Functions algebra and the following rewriting rules:

i) e]ε +---> e
ii) n]s +---> error if s≠ε
iii) x]s +---> error if s≠ε
iv) g(e0,...,ek)]js +---> if 0⩽j⩽k then ej]s else error
v) f(e0,...,ek)]s +---> if f(e0,...,ek)=e is an instance of the recursive case of P then e]s else error

For simplicity in the facts presented in the previous Section we used the s-selectors with reference to the recursive calls only, so that for instance, $g(\ldots,\underset{j}{f(\ldots)},\ldots,\underset{k}{f(\ldots)},\ldots)]js$ +---> $\underset{j}{f(\ldots)}]s$.

The fact F2 is accepted by the Calculus C because: clear(k+2)]0 +---> clear(k) and clear(k+2)]11 +---> clear(k+1)]1 +---> clear(k).

Fact F1 of Section II is an example of the second form of facts.

IV SOME RESULTS AND CONCLUSIONS

The following results can be shown about our system for developing concurrent programs [5].

Correctness Theorem for Communications.
If for every program P in L0 and s ann ℓ and s'ann ℓ occurring in Tr(P) in the recursive call at position j and j'(respectively) f(...)]js=f(...)]j's' holds, and Tr(P) is deadlock-free then Tr is correct, that is, for every P in L0 the programs P and Tr(P) compute the same function. □

The proof of the above Theorem would require the formalization of the Translation Algorithm Tr, which we did not present here. We have seen Tr in action when developing program P in Section II.

Proposition. Given a program P in L0, if a reading communication takes place during the evaluation of Tr(P) with a non-linear recursion then an exponential number of calls can be saved, and in some cases one may obtain a linear time algorithm (see for instance, program P3). □

That Proposition is important because it guarantees the performance improvements of the derived programs, and often it allows to satisfy the given complexity requirements.

We have seen that by adding suitable communications to the functional programs we can derive more efficient executions.
A general question arises: Is there an optimal set of facts from which one can obtain the most efficient communications to be added to a given program? The answer is positive in the case of programs with one recursive case only. It can be shown that given a fact of the form f(...)]s1=f(...)]s2, the corresponding optimal communication is produced by erasing the longest initial equal subsequence of s1 and s2. For instance, from the fact F3 of Section II we can get fact F4: clear(k+2)]01=clear(k+2)]10. It can easily be seen that the communications derived from F4 save more computations steps than those derived from F3.

We have presented some basic idea for the construction of a knowledge base system for developing concurrent functional programs. The system uses a calculus for checking the correctness of supplied "factual knowledge" (or facts) about the functions to be computed. It then translates those facts into suitable communications among concurrent agents so that the derived computations may satisfy given complexity constraints.

REFERENCES

[1]Barstow,D. "An Experiment in Knowledge-Based Automatic Programming" Artif. Intel. 12:2 (1979) 73-119.
[2]Burstall,R.M.,J.Darlington "A Transformation System for Developing Recursive Programs" JACM 24:1 (77)
[3]Bauer,F.L. & al. "Notes on the Project CIP" TUM-INFO-7729 Infomatik. Technische Univ. München (1977)
[4]Iverson,K.E. "A Programming Language" Wiley,N.Y. (62)
[5]Pettorossi,A.,A.Skowron "A Methodology for Improving Parallel Programs by Adding Communications" LNCS n.208, Springer Verlag, 1985, pp.228-250.
[6]Pettorossi,A.,A.Skowron "Using Facts for Improving the Parallel Execution of Functional Programs" In Proc.1986 Int.Conf.Parall.Processing, Illin.(86)
[7]Scherlis,W.L.,D.Scott "First Steps Towards Inferential Programming" In Proc. IFIP 83 North Holland (1983).