FACTUAL KNOWLEDGE FOR DEVELOPING CONCURRENT PROGRAMS

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

Andrzej Skowron
Institute of Mathematics
Warsaw University
Institute of Computer Science
University of North Carolina at Charlotte
00-901 Warsaw (Poland)
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 Juristall-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 these 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 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, 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.

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, i.e., 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.

**Functional Programs**

*Function 1: Checking Facts*

---

**Calculus C for Translation**

Efficient Concurrent Programs with Communicating Agents

**Knowledge Base:** Historical Information

**Transformation Techniques**

- Equality of terms
  - Language of Facts LF

**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 Knowledge Base is updated and how new transformation techniques can be derived from old ones.

**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 $k'$) denotes the move which takes away from (or puts back to) the stick the ring $k$.

For $k=2, \ldots, N$ moves $k$ or $k'$ can be performed iff rings $1, \ldots, k$ are not on the stick and ring $k-1$ is on the stick.

A move $k$ computes the sequence of moves which removes rings $1, \ldots, k$ from the stick, if initially they are all on the stick. Conversely, put(k) computes the moves for putting back rings $1, \ldots, k$ on the stick, if initially they are not on the stick.

Let us denote by $s$ the sequence of moves $s_1 \ldots s_p$. We can supply to our system the following fact $F_1$: $\text{put}(k) = \text{clear}(k)$.

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: $\text{put}(k+2) = \text{clear}(k+1)$. The left son call of the right son call of $\text{clear}(k+2)$ requires the value of the left son call $\text{clear}(k)$ and the right son call $\text{clear}(k+1)$. The order of the calls we used is the left-to-right one, after substituting $\text{clear}(k)$ for $s$ in the expression of $\text{clear}(k+2)$.)

Once the fact $F_1$ has been accepted, the translation algorithm $Tr$ produces from it the following program version:

- $\text{clear}(1)\rightarrow 1$, $\text{clear}(2)\rightarrow 2$, $\text{clear}(k+2)\rightarrow s:k+2\rightarrow s:clear(k+1)$ where $s=clear(k)$

This program is more efficient than program $P$ because a smaller number of recursive calls is generated: 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 $P_1$ (or $P$) can be discovered by symbolic evaluation:

- $\text{clear}(k+2)\rightarrow 0$ for $k>0$.

Later on we will give a formal definition of the language $L_0$ 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

\[ p. clear(l) = l, \quad p. clear(2) = 2 : 1, \quad p. clear(k+2) = s : k+2 : s : clear(k+1), \]
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:

\[
\begin{align*}
P2: & \text{clear(k+2) = (s:k+2 :: \text{clear}(k+1)(1 \text{ comm } s) where } s = \text{clear}(k)(c \text{ comm } k)) \text{ decl } k \\
\end{align*}
\]

The informal explanation of the communication annotations added by Tr is as follows.

So assume that recursively defined functions are evaluated by a set of computing agents, i.e., triples of the form \(<\text{name}, \text{message}> :: \text{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> :: \text{clear}(k+2)> generates the two agents \(<x0,m0> :: \text{clear}(k)> and \(<x1,m1> :: \text{clear}(k+1)>.

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Using the location \(k\) for the fact F2.}
\end{figure}

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

\[
\begin{align*}
P3: & \text{clear}(k+2) = (s:k+2 :: \text{clear}(k+1)(1 \text{ write } s) where } s = \text{clear}(k)(c \text{ comm } k)) \text{ decl } k \\
\end{align*}
\]

One more fact can be discovered about the program P0:

\[
P3: \text{clear}(k+2) 001 = \text{clear}(k+2) 010.
\]
Let a configuration be a (finite) set of agents and CON be the set of all configurations. By $r(x_1, \ldots, x_k)$ we denote a rule-schema $r$: 

$$r(a_1, \ldots, a_k) \text{ or } r$$

in which $x_1, \ldots, x_k$ are the only (meta)variable occurrences. Given the constants $a_1, \ldots, a_k$, $r(a_1, \ldots, a_k)$ denotes a concrete instance of $r$ which can be derived by substituting $a_1, \ldots, a_k$ for $x_1, \ldots, x_k$.

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

$$c \rightarrow r \rightarrow c'$$

where $c'$ holds iff $c$ is true and $lh \rightarrow rh$ it cond.

The transition relation corresponding to a sequence $r_1, \ldots, r_k$ of instances of rule-schemas is defined as the composition of the transition relations $r_1 \rightarrow \ldots \rightarrow r_k$ and it is denoted by $\triangleright$. Let Semi($P$) denote the rule-schemas associated to $P$ for any program $P$ in $L_1$. They will be introduced below. The (one-step) transition relation of a program $P$ in $L_1$ (written as $P \triangleright$) defines the semantics of $P$, and it is specified as follows:

$$c \triangleright c'$$

where $c'$ holds iff there exists a non-empty finite sequence $s$ of instances (derived by the same substitution) of rule-schemas in Semi($P$), s.t. for an arbitrary permutation $s'$ of $s$ we have:

$$c \triangleright s$$

The condition on the permutations of $s$ is the one which is usually considered for expressing that the atomic transitions $r_1, \ldots, r_k$ refer to non-conflicting subsets of agents, and then they can be performed in parallel. More details are given in a companion paper [6] where we studied the behavior of agents communicating agents which concurrently evaluate functional programs.

Therefore for computing, for instance, the value of $f(\ldots)$ where $f$ is defined by a program $P$ in $L_1$, we consider an initial computing agent $<c,E>::f(nl,\ldots)$ where $n$ is a constant, and say that the value of $f(\ldots)$ is $n$.

Given a program in $L_1$, Semi produces the rewriting rule-schemas for agents as follows.

1. Generation of sons with communications

$$f(e_0, \ldots, e_p)\equiv f(e_1, \ldots, f(e_r, \ldots))$$

produces the rule-schema:

$$<x, E, \triangleright f(e_0, \ldots, e_p)> \equiv \{<x, E, \triangleright f(e_1, \ldots, f(e_r, \ldots)>\}$$

The condition on the permutations of $s$ is the one which is usually considered for expressing that the atomic transitions $r_1, \ldots, r_k$ refer to non-conflicting subsets of agents, and then they can be performed in parallel. More details are given in a companion paper [6] where we studied the behavior of agents communicating agents which concurrently evaluate functional programs.

2. Base Cases

$$f(e_0, \ldots, e_k)\equiv f(e_0, \ldots, e_k)$$

produces:

$$\{<x, E, \triangleright f(e_0, \ldots, e_k)>\}$$

3. Values to Fathers

$$f(e_0, \ldots, e_k)\equiv g(e_0, \ldots, e_k)$$

4. Writing Communications

$$f(e_0, \ldots, e_k)\equiv f(e_0, \ldots, e_k)$$

5. Reading Communications

$$f(e_0, \ldots, e_k)\equiv f(e_0, \ldots, e_k)$$

6. Basic Functions Evaluation

$$f(e_0, \ldots, e_k)\equiv f(e_0, \ldots, e_k)$$

7. Initial Agent

$$f(e_0, \ldots, e_k)\equiv f(e_0, \ldots, e_k)$$

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

Now, as an example of the definition of Semi let us present the evaluation of clear(5). We write

$$\{<x, E, \triangleright f(\ldots)>\}$$

for denoting that the agents to the left are the ones to the right, except for $a_1, \ldots, a_k$ (see also figure 4).

Semi($P$) contains (besides others) the following rule-schemas:

$$<x, E, \triangleright f(k+2)> \equiv \{<x, E, \triangleright f(k+2)>\} \quad (r_1)$$

$$<x, E, \triangleright f(1)> \equiv \{<x, E, \triangleright f(1)>\} \quad (r_2)$$

$$<x, E, \triangleright f(2)> \equiv \{<x, E, \triangleright f(2)>\} \quad (r_3)$$

$$<x, E, \triangleright f(3)> \equiv \{<x, E, \triangleright f(3)>\} \quad (r_4)$$

The rule-schema $r_1$ comes from the Generation-of-Sons schema, the rule-schemas $r_2$ and $r_3$ from the Base-Cases schema, and $r_4$ and $r_5$ from the Writing and Reading Communications schemas.

The initial agent is $<x, E, \triangleright f(5)>$.

We have:

$$\{<x, E, \triangleright f(5)>\}$$

The where-expressions are not considered by Semi because one may get rid of them by substituting the corresponding expressions. However, when applying the generation-of-sons rule, we assume that Semi creates the same agent for all substituted occurrences of the same where-expression.
An improved operational semantics may garbage-collection at the end of program execution.

A general question arises: Is there an optimal set of communications to be added to a given program? The answer is positive in the case of programs with non-linear recursion. An exponential time algorithm can be derived from F4 save more computations steps than those derived from F3.

We have presented some basic ideas 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