PARTIAL EVALUATION WITH INFERENCE RULES

EVALUATION PARTIELLE DE REGLES D'INFERENCE

Laurent Hascoët
INRIA Sophia-Antipolis
January 12, 1988

Abstract

We suppose we are given a program P in some language L, and a set of inference rules that define the dynamic semantics of this language L. We propose a tactic for (partially) evaluating a given predicate in a set of inference rules. This tactic applied to P and to the dynamic semantics of L gives back a new set of specialized inference rules that define the dynamic semantics of P. Thus, this process, given an interpreter for the language L, yields a compiled version of any program written in L, the compiled version being a set of inference rules.

Keywords: Inference Rules, Partial Evaluation, Prolog, Tactic

Résumé

Nous considérons un programme P écrit dans le langage L, et un ensemble de règles d'inférence définissant la sémantique dynamique de L. Par évaluation partielle de ces règles d'inférence par rapport à P, nous obtenons un ensemble de règles d'inférence définissant la sémantique de P, et que l'on peut considérer comme une version compilée de P. Nous proposons ici une tactique pour effectuer l'évaluation partielle d'un ensemble de règles d'inférence. Cette tactique est elle même implémentée sous forme de règles d'inférence, et est appliquée à quelques exemples.

Mots Clés: Règles d'Inférence, Evaluation Partielle, Prolog, Tactiques

PARTIAL EVALUATION WITH INFERENCE RULES

L. Hascoët — INRIA Sophia-Antipolis
January 12, 1988

Abstract

We suppose we are given a program P in some language L, and a set of inference rules that define the dynamic semantics of this language L. We propose a tactic for (partially) evaluating a given predicate in a set of inference rules. This tactic applied to P and to the dynamic semantics of L gives back a new set of specialized inference rules that define the dynamic semantics of P. Thus, this process, given an interpreter for the language L, yields a compiled version of any program written in L, the compiled version being a set of inference rules.

1. INTRODUCTION

This paper presents a method to achieve partial evaluation of programs written with inference rules. Most of the previous works about partial evaluation was done on functional languages, such as ML [Futamura 87][Jones 87a][Schmidt 87]. It is a recent trend to study partial evaluation in logic programming. An interesting point is that some problems of partial evaluation well-known for functionnal languages also arise with logic programming. The most famous application of partial evaluation is the automatic generation of compilers from interpreters. The idea originates from Ershov [Ershov 77a]and Futamura [Futamura 71]. It is to perform a partial evaluation of the interpreter, where the only known input is the program to interpret. The result is a compiled version of the program. This paper will focus on the partial evaluation of interpreters. This is natural since the logic programming language that we use is mainly dedicated to defining semantic operations on languages, such as interpreting or compiling.

The partial evaluation method that we present may be interesting because of its choices about call unfolding. There is also an obvious relation with the works about abstract interpretation [Bruynooghe 87][Gallagher 87]. Since this method is now (partially) implemented, we have already some nice results. Some of these are shown at the end. They give examples of what could be expected from partial evaluation used for compilation.

Let's present the context of this work. Our team is involved in the development of the CENTAUR system [Kahn G. 88], which is originaly a syntax-directed editor. Like many other syntax-directed editors, CENTAUR provides tools to define operations on syntactic trees, mainly for semantic operations. Our choice is to define these tools in terms of inference rules. Thus we have a special language of inference rules, called TYPOL [Despeyroux 84][Despeyroux 83]. We call our

method "natural semantics" [Clément 85]. TYPOL programs are just a bunch of inference rules, with some extra information around, about the language to which they apply, and used for type-checking TYPOL programs themselves. TYPOL programs are compiled into C-prolog [Bowen 84]to be executed. Using TYPOL, we have already defined many semantic tools, such as:

- type-checkers for ASPLE, mini-ML, ML, ESTEREL,TYPOL.
- interpreters and/or compilers for ASPLE, SML, mini-ML, ML, CAM, TYPOL.
- translators from ESTEREL to kernel-ESTEREL, or between dialects of PROLOG.

We also programmed some more exotic operations. We will focus here on the interpreters.

The motivations of this work are the following. For exterior reasons, we felt the need for some kind of tool to build proof trees from TYPOL rules. These reasons are:

- Some people in our team are concerned in proving properties of TYPOL programs, such as the equivalence between an ASPLE interpreter on one side, and on the other side a compiler from ASPLE to SML, along with the SML interpreter. All this proving means building a huge number of proof trees, whose constituents are TYPOL rules. This is tedious and can lead to mistakes in unifications. We must provide automatic assistance.
- With such a tool, one can write an interpreter for TYPOL. This has didactic as well as practical interest, since the Prolog programs generated by the compiler are not easy to read and debug.
- As well, while building a proof tree, one can use any available rule to expand any unsolved predicate. So, one can simulate a real nondeterministic evaluation of TYPOL programs. This is useful because the theoretical semantics of TYPOL is the one of inference rules, *i.e.* the order in which the rules or the predicates appear is meaningless. In fact, since TYPOL is compiled into C-prolog before execution, the order of the predicates and rules has actually the same meaning as in PROLOG. While TYPOL is nondeterministic in theory, it is deterministic in practice.

We have now programmed a first, experimental version of this tool to manipulate proof trees [Hascoët 87]. This tool is itself written in TYPOL, because it is a good language for developing prototypes. This tool allows the user to define tactics about how to build a proof tree made with TYPOL rules. When N. D. Jones came to visit us in Sophia-Antipolis in the spring of '87, he explained to us his works on partial evaluation [Jones 87a][Jones 87b]. We felt we had to define a partial evaluator for TYPOL programs. We already had a running version of our proof-tree builder. Therefore adapting the proof-tree builder to partial evaluation was only a question of writing a new tactic! It was a challenging problem to design this tactic, then implement it and watch the result. Like most of the partial evaluators, the crucial question our system has to answer is when to unfold calls.

In the following section, we explain how one can represent the execution of a TYPOL program by a proof tree. The third section describes our method of partial evaluation. In the fourth section, we give a proof of the correctness of our method. The last section presents the actual implementation of our tactic. We also show examples of compilation using partial evaluation.

2. DYNAMIC SEMANTICS IN TYPOL

2.1. TYPOL programs

Throughout this paper, we are going to stick to the same TYPOL program. This program defines an interpreter for a small PASCAL-like language, called ASPLE. We are going to illustrate the use of TYPOL by interpreting a very simple ASPLE program. Here are some typical TYPOL rules from the ASPLE interpreter:

$$(1)\qquad \frac{\emptyset \overset{\text{allocate}}{\vdash} \text{DECLS} : \sigma \qquad \sigma, i \vdash \text{STMS} : \sigma', o}{\vdash \textbf{begin}\ \text{DECLS}\ \textbf{;}\ \text{STMS}\ \textbf{end} : i, o}$$

(2) $$\frac{\sigma, i_1 \vdash \text{STM} : \sigma', o_1 \qquad \sigma', i_2 \vdash \text{STMS} : \sigma'', o_2 \qquad i = i_1 \oplus i_2 \qquad o = o_1 \oplus o_2}{\sigma, i \vdash \text{STM}; \text{STMS} : \sigma'', o}$$

(3) $$\frac{\sigma \vdash \text{EXP} : \text{“}true\text{”} \quad \sigma, i \vdash \text{STM1} : \sigma', o}{\sigma, i \vdash \textbf{if } \text{EXP } \textbf{then } \text{STM1 } \textbf{else } \text{STM2 } \textbf{fi} : \sigma', o}$$

(4) $$\frac{\sigma \vdash \text{EXP} : \text{“}false\text{”} \quad \sigma, i \vdash \text{STM2} : \sigma', o}{\sigma, i \vdash \textbf{if } \text{EXP } \textbf{then } \text{STM1 } \textbf{else } \text{STM2 } \textbf{fi} : \sigma', o}$$

(5) $$\frac{\sigma \vdash \text{EXP} : \text{“}true\text{”} \quad \sigma, i_1 \vdash \text{STM} : \sigma', o \quad \sigma', i_2 \vdash \textbf{while } \text{EXP } \textbf{do } \text{STM } \textbf{end} : \sigma'', o_2 \quad i = i_1 \oplus i_2 \quad o = o_1 \oplus o_2}{\sigma, i \vdash \textbf{while } \text{EXP } \textbf{do } \text{STM } \textbf{end} : \sigma'', o}$$

(6) $$\frac{\sigma \vdash \text{EXP} : \text{“}false\text{”}}{\sigma, \emptyset \vdash \textbf{while } \text{EXP } \textbf{do } \text{STM } \textbf{end} : \sigma, \emptyset}$$

(7) $$\frac{\sigma \vdash \text{VAR} : x \qquad \textbf{read}\langle v, \text{TYPE}\rangle \qquad \sigma \overset{\text{update}}{\vdash} x, v : \sigma'}{\sigma, \text{in}[v] \vdash \textbf{input } \text{VAR TYPE} : \sigma', \emptyset}$$

The intuitive meaning of these rules is straightforward. The variables named i_n and o_n are the lists of input and output values consumed and produced by the execution of the program. The operation named $\oplus$ represents the concatenation of such lists. The predicates come in two main forms. One form applies to user-defined TYPOL predicates, and uses a turnstile symbol $\vdash$. The turnstile sometimes comes with a superscript that means it is a different predicate. The other form of predicates represents direct calls to external predicates (here Prolog predicates). Examples are the **read** predicate, and the $\oplus$.

- Rule (1) means that the declarations part of the program is used to build a store, with which the statements will be interpreted.
- Rule (2) means that a sequence of statements is interpreted by interpreting the first statement and the rest of the statements. Notice that the “rest” part of the list is interpreted with a new store that is output by the first statement. This gives no problem because the standard tactic to execute TYPOL rules is to evaluate predicates in the order they are written in.
- Rules (3) and (4) define the classical behavior of the **“if”** statement. Just notice that in the case where the test should return “false”, the rule (3) is going to fail after some computation. Then by usual Prolog backtracking, rule (4) is called instead.
- Rules (5) and (6) are similar, but deal with the **“while”** statement.
- The behavior of rule (7) must be given in a very operational way. It causes a call to the PROLOG **“read”** predicate, and then modifies the store. It also modifies the list of input values.

2.2. Building a proof tree

Let’s suppose that this TYPOL program is called, with some (type-checked) ASPLE program provided, and with some values input and output when necessary. This means that we are going to interpret the ASPLE program. What really happens is that every TYPOL rule is translated to PROLOG in an obvious way, and that PROLOG is run in the classical depth-first left-to-right order. We can therefore simulate exactly this mechanism by manipulating the TYPOL rules in depth-first, left-to-right order. We insist this is not the official theory of TYPOL semantics, but

only the current implementation. Whenever the TYPOL implementation changes, using a new execution tactic, we shall be able to change our simulation to match the new tactic again.

Thus, from now on, what we call executing a TYPOL program on a given unsolved TYPOL predicate is to build the proof tree

- taking this predicate as a starting goal,
- growing with TYPOL rules from the TYPOL program, or sometimes with direct executions in PROLOG (input-output, arithmetic expressions),
- built in strict depth-first left-to-right order, just like the compiled interpreter would have done in PROLOG.

For instance, let's suppose our type-checked ASPLE program is:

```
begin
    int x;
    input x int;
    if (deref x = 9)
      then x:=0;
      else  x:=deref x +1;
    fi;
    output deref x int;
end
```

The **deref** operator in ASPLE means that it is the value of the variable that is asked, and not only its address in the store. Let's suppose that we shall type "3" as the input value, then we may build the evaluation (the proof tree) of the program. The starting predicate is of course:

$$(a) \qquad \vdash \textbf{begin int x;}...\textbf{;}...\textbf{;}...\textbf{; end} : i, o$$

From now on, we are going to talk about "proof trees", both for partial proof trees and completely solved proof trees, regardless of whether some predicates remain unsolved or not. After some obvious steps, the proof tree becomes (b) ($[\mathbf{x} \mapsto]$ means that $\mathbf{x}$ is undefined):

$$\dfrac{\dfrac{}{\vdots}\quad \dfrac{\dfrac{}{\vdots}}{[\mathbf{x}\mapsto] \vdash \mathbf{x} : s\_id\mathbf{x}} \quad \mathbf{read}\!<\!v, \mathrm{int}\!> \quad \overset{\text{update}}{[\mathbf{x}\mapsto]\vdash s\_id\mathbf{x}, v : \sigma'}}{[\mathbf{x}\mapsto], \mathrm{in}[v] \vdash \textbf{input x int} : \sigma', \emptyset \qquad \sigma', i_2 \vdash \textbf{if}...\textbf{;}...\textbf{;} : \sigma'', o_2 \quad i = \mathrm{in}[v] \oplus i_2 \quad o = \emptyset \oplus o_2}}{\dfrac{\overset{\text{allocate}}{\emptyset \vdash \textbf{int x} : [\mathbf{x}\mapsto]} \qquad [\mathbf{x}\mapsto], i \vdash \textbf{input x int;}...\textbf{;}...\textbf{;} : \sigma'', o}{\vdash \textbf{begin int x;}...\textbf{;}...\textbf{;}...\textbf{; end} : i, o}}$$

In this proof tree, the small horizontal rules above three dots mean that the corresponding branch of the proof tree is too big to print here, but is completely solved, *i.e.* contains no unsolved predicate. Conversely, when a predicate appears without any fraction bar above it, it means this predicate is still to be solved. At this stage where the proof tree is (b), the next predicate to solve in the PROLOG order is the **"read"**, which will finally ask us for an integer, and we shall return "3" as we said. Then, after some more steps of evaluation, the current proof tree is:

$$(c) \qquad \dfrac{\dfrac{}{\vdots}\quad \dfrac{[\mathbf{x}\mapsto 3], i_{21} \vdash \textbf{if}... : \sigma', o_{21} \quad \sigma', i_{22} \vdash \textbf{output}...\textbf{;} : \sigma'', o_{22} \quad i_2 = i_{21} \oplus i_{22} \quad o_2 = o_{21} \oplus o_{22}}{[\mathbf{x}\mapsto 3], i_2 \vdash \textbf{if}...\textbf{;}...\textbf{;} : \sigma'', o_2 \qquad i = \mathrm{in}[3] \oplus i_2 \quad o = \emptyset \oplus o_2}}{\dfrac{\dfrac{}{\vdots}\quad \dfrac{}{\vdots}}{[\mathbf{x}\mapsto], i \vdash \textbf{input x int;}...\textbf{;}...\textbf{;} : \sigma'', o}}{\vdash \textbf{begin int x;}...\textbf{;}...\textbf{;}...\textbf{; end} : i, o}$$

and the next unsolved goal to solve is:

$$[\mathbf{x} \mapsto 3], i_{21} \vdash \mathbf{if...then...else...fi} : \sigma', o_{21}$$

The first rule for **"if"**, (3), will be chosen, and at some stage the resolution will fail, because $3 = 9$ returns *"false"*. Then, backtracking will undo the previous steps of the building of the proof tree, until it comes back to the (c) stage. At that moment, since it notices there was another possible inference rule to apply, it chooses the second rule for **"if"**, (4). It may be remarked that this is not the way people use to build proof trees, since they know where they are going. They would instead build a proof tree to show that the condition returns *"false"*, then use at once the correct rule (4) without trying the other one. But this tactic is human reasoning, and is different from the actual TYPOL automatic proving.

Now that rule (4) has been applied, the process goes on, and solves the **"else"** part of the **"if"**. At some time, the current predicate to solve will be $\mathbf{add}(3, 1, \mathrm{X})$ which is a PROLOG predicate, like **read** was. This means that this predicate will be solved by calling PROLOG, and will instantiate X by 4. From the TYPOL point of view, everything looks as if $\mathbf{add}(3, 1, 4)$ was a given axiom. When the execution terminates, we shall obtain the final proof tree (d):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdots}}{[\mathbf{x} \mapsto 3] \vdash \mathbf{x{=}9} : \text{“}false\text{”}} \quad \dfrac{\dfrac{\overline{\mathbf{add}(3,1,4)}}{\vdots}}{[\mathbf{x} \mapsto 3], \emptyset \vdash \mathbf{x{:=}x{+}1} : [\mathbf{x} \mapsto 4], \emptyset}}{[\mathbf{x} \mapsto 3], \emptyset \vdash \mathbf{if}... : [\mathbf{x} \mapsto 4], \emptyset} \quad \dfrac{\dfrac{\overline{\mathbf{write}(4)}}{\vdots}}{[\mathbf{x} \mapsto 4], \emptyset \vdash \mathbf{output}...; : [\mathbf{x} \mapsto 4], \mathrm{out}[4]} \quad \dfrac{}{\vdots}}{\dfrac{}{\vdots} \quad [\mathbf{x} \mapsto 3], \emptyset \vdash \mathbf{if}...;...; : [\mathbf{x} \mapsto 4], \mathrm{out}[4] \quad \dfrac{}{\vdots}}}{\dfrac{}{\vdots} \quad [\mathbf{x} \mapsto], \mathrm{in}[3] \vdash \mathbf{input\ x\ int};...;...; : [\mathbf{x} \mapsto 4], \mathrm{out}[4]}}{\vdash \mathbf{begin\ int\ x};...;...;...; \mathbf{\ end} : \mathrm{in}[3], \mathrm{out}[4]}$$

3. BUILDING A PARTIAL EVALUATION

3.1. What is a partial evaluation for inference rules

What we want now is to partially evaluate a TYPOL predicate, with a given TYPOL program. The main idea of partial evaluation, in any language, is to try and evaluate a program with some of its inputs unknown. This yields a new program that, given the remaining inputs, is equivalent to the starting program. This definition is more adapted to imperative or functionnal languages; for logic programs, we have to paraphrase it. In this case, we consider the program is the bunch of inference rules, along with a "free variable" V representing the goal to solve. The input to a logic program is or is equivalent to an instanciation of the variables in the goal we want solved (here the variable V). A partially known input is also an instanciation of the goal. Whether an input is partial or not is not for the system to decide, because Prolog cannot guess whether a variable is an expected result or a missing data. The remaining (unknown) input is again a substitution of the variables in the goal. The result of partial evaluation is another logic program, and the remaining input is given by another goal.

When we have given a method for partial evaluation, we shall have to check its correctness in the following sense:

If we call (P) the initial set of inference rules, i_1 the partial input, i_2 any remaining input, (P') the result of partial evaluation of (P) with repect to i_1

$$\begin{array}{ccc} (\mathrm{P}) & \xrightarrow{i_1} & (\mathrm{P}') \\ {\scriptstyle i_1 \cup i_2}\downarrow & & \downarrow{\scriptstyle i_2(\cup i_1)} \\ \mathrm{T} & \simeq & \mathrm{T}' \end{array}$$

The proof tree T' obtained by executing (P') with input i_2 (and somehow remembering i_1) must be equivalent to the proof tree T obtained by executing (P) with all its input known at once.

Two degenerate cases are already clear. First a normal, complete execution, such as the one yielding the (d) tree above, is a partial evaluation. Just see that (P') is

$$\vdash \textbf{begin int x};...;...;...; \textbf{end} : \mathrm{in}[3], \mathrm{out}[4]$$

and i_2 is empty. Here the result of partial evaluation is just an axiom. The other case is that no evaluation at all is also a partial evaluation, though unefficient, since in that case (P') equals (P).

3.2. Idea of the method

3.2.1. *Definitions about proof trees*

Skeletons:
A proof tree T_1 is a "skeleton" of another one T_2 if T_2 can be obtained from T_1 by instantiating free variables of T_1, and/or by developing some predicate not yet solved in T_1. For instance (a) is a skeleton of (b), which is a skeleton of (c), which is a skeleton of (d). The skeleton relation is transitive.

Border:
We call "border" of a proof tree the list of all the predicates that appear in the proof tree and that have not been expanded, *i.e.* for which no inference rule has been applied. The order of these predicates within the border is the same as in the preorder search of the proof tree, since we are bound by the PROLOG order of execution. For instance, the border of (a) is (a), while the border of (d) is empty. The border of a proof tree is not empty *iff* this proof tree is a partial proof tree.

Equivalence:
Two proof trees are equivalent *iff*:

- They have exactly the same root predicate (bottom line) and
- They have exactly the same border, in the same order.

Simplification:
What we call "simplifying" a proof tree is to build the equivalent proof tree made with the root predicate put under the border. All intermediate steps of the proof are removed. For instance, simplifying (b) gives us

$$(b') \qquad \frac{\textbf{read}{<}v, \mathrm{int}{>} \quad [\mathbf{x} \overset{\text{update}}{\mapsto}] \vdash s\_id\mathbf{x}, v : \sigma' \quad \sigma', i_2 \vdash \textbf{if}...;...;: \sigma'', o_2 \quad i = \mathrm{in}[v] \oplus i_2 \quad o = \emptyset \oplus o_2}{\vdash \textbf{begin int x};...;...;...; \textbf{end} : i, o}$$

3.2.2. *biggest common skeleton*

We want to partially evaluate the ASPLE evaluator when the known input is our small example program, represented by the (a) predicate, and the unknown part is the integer value the user will type. Intuitively, the (b) proof tree is a step towards partial evaluation, since it needs no knowledge of the missing data. The work needed to get (b) is independant from this data, and should be done once and for all at partial evaluation time. The way we store (b) is by adding its simplified form

(b') to (P), to get a first version of (P'). When (P') is run, the rule (b') is chosen first, and no inferences are needed to get the initial store $[\mathbf{x} \mapsto]$.

What we stress here is the fact that (b) is a common skeleton to all possible proof trees (execution trees). All these proof trees may be built more efficiently using (b) directly. If we consider the set S_0 of all possible proof trees starting from (a), and we call (s_0) the biggest common skeleton to all trees in S, then we keep (s_0) as a part of the future partially evaluated program.

For every tree in S, the result of removing the (s_0) part is the collection of the proof trees that were branched above (s_0). Now let's remove the common part to all trees in S_0. What we get is a collection of sets of proof trees. For each of these sets, there is no common skeleton. This comes from the fact that (s_0) is biggest. This is what happens for instance if we come to an "if" statement, and the proof trees start with either inference rule (3) or (4). Now in this case, let's split the corresponding set of trees into subsets, so that each subset *has* a common skeleton. We may then repeat the mechanism.

All the common skeletons that we kept, (s_0), (s_1), etc, are the bricks to build every tree in S. This means they are the partially evaluated program. Since a program is made of inference rules, we have to take their simplified versions (s'_0), (s'_1), etc. This is legal since simplification yields equivalent trees. Now we shall explain our tactic to obtain these biggest common skeletons.

3.3. Our partial evaluation tactic

Starting from the ideas above, and adding improvements from time to time, we are going to present the "operational" method to build biggest common skeletons, and therefore partial evaluations of a program (P). We always start with the initial, partially instanciated goal. We shall consider it as refinement zero of our partial evaluation (It is clear it is a common skeleton). Thus partial evaluation consists in building a proof tree. The first part of the tactic is when we can grow the common skeleton further. It is the same in Anders Bondorf's paper [Bondorf 87b], which deals with rewrite rules instead of inference rules.

Tactic (TC_1): When some unsolved goal in the border of the currently build proof tree can be expanded by only one rule in (P), then expand this goal to get a bigger common skeleton.

Next part of the tactic is about goals to be solved by Prolog. This is more delicate because these goals often deal with input output, or other side effects, or arithmetic operations. Since there is no general method, the user has to tell if the goal may be solved at partial evaluation time. Fortunately, there are few such goals.

Tactic (TC_2): When some unsolved goal in the border of the currently build proof tree may be solved by Prolog at partial evaluation time, and Prolog solving succeeds, then the common skeleton may be expanded in the corresponding way. On the other hand, if the goal is not solved, then the Prolog definition of the predicate must be kept for the partially evaluated program.

Examples of Prolog goals which generally cannot be solved at partial evaluation time are input output predicates, or arithmetic predicates whose operands are not instanciated yet. When the user allows the solving of the goal, if this solving fails, we prefer to do nothing here, instead of a delicate backtracking.

The next case is when many TYPOL rules apply to a goal. This corresponds to the case when the set of proof trees has to be split, because no common skeleton exists. We just split into as many subsets as there are applicable rules. This is related to the specialization of function calls for

the partial evaluation of functionnal languages.

Tactic ($\mathbf{TC}_3$): When some unsolved goal in the border of the currently build proof tree may be solved by two or more inference rules from (P), then do the following for each applicable rule: First take a copy of the unsolved goal. Make it a new initial common skeleton. Then apply the chosen rule. Then continue partial evaluation of this new common skeleton.
When all applicable rules have been treated, collect the bottom line predicates of all the corresponding skeletons. Compute their most precise common unifier U. Then go back to the initial proof tree and unify the unsolved goal with U. This gives a new bigger common skeleton.

The last part of this tactic, about the common unifier U, comes from a intuitive remark. Each case (using one applicable rule) yields back one possible solution to the unsolved goal. In logic programming, a solution to a goal is a substitution of its variables. If there is an intersection between all these substitutions, this means we know something about the solution of the unsolved goal, whatever rule will be applied there. Thus this information may be used in the biggest common skeleton. Sometimes this intersection becomes a fixpoint equation. In that case, solving this equation is equivalent to proving properties of the called predicate by structural induction. Examples will be given in section 3.5. and 5.1.

The fourth part of the method deals with the simplifications of the proof trees into inference rules.

Tactic ($\mathbf{TC}_4$): When no more unsolved goals can be expanded, then simplify the proof tree and store the resulting inference rule.

When all common skeletons are found and simplified, there is a last improvement to the resulting collection of inference rules C. Unsolved goals that are calls to other inference rules from C are often heavy, because they keep track of values that are now fixed forever at partial evaluation time. In fact, only the free variables in these calls may be usefull, to collect results. The rest is only syntactic structure. What we call syntax elimination is to replace such a call by a call to a new predicate, with a new name (or something to distinguish it), and whose arguments are the variables in the original call. An example will be given in section 3.5. This can be related to the projection method in [Launchbury 87].

Tactic ($\mathbf{TC}_5$): Replace all calls to TYPOL predicates, performing syntax elimination. Perform also the corresponding modification in the definition of these TYPOL predicates.

This method may be related to abstract interpretation of Prolog programs, like in [Bruynooghe 87]. The paper [Gallagher 87]presents a method of partial evaluation of FCP programs that uses abstract interpretation, and which is close to the tactic presented here.

3.4. Classical problems

3.4.1. *Infinite unfolding*

The first classical problem that we meet is infinite unfolding at partial evaluation time. As shown by inference rule (5), solving a goal may lead to the solving of the same goal. In our tactic, this may lead either to an infinite building of the biggest common skeleton, through tactic $(\mathbf{TC}_1)$, or to the building of an infinite number of specialized rules, through tactic $(\mathbf{TC}_3)$. To avoid that, we need to implement the following control:
If a call to a goal G_1 leads, in the same common skeleton, to a call to goal G_2, and if G_1 is close to G_2, then the goal G_1 should not be expanded in the skeleton. Instead, for the goal G_1, the tactic

(TC$_1$) that was applied before should be replaced by tactic **(TC$_3$)**. This first step of the method allows to transform an infinite biggest common skeleton into an infinite number of skeletons. The meaning of "close to" is intuitive: G_1 is close to G_2 if there is a risk of infinite unfolding. There are many examples of such situations, such as (with C-Prolog syntax):

- pred(A, R) calling pred(B, R).
- pred(A, R) calling pred(s(A), R).
- oper(5, RES) calling oper(6, RES).

Suppose now that a call to G_1 leads, through tactic **(TC$_3$)**, to a call to G_2 (G_1 close to G_2). Then we have to apply tactic **(TC$_3$)** again, but instead of solving G_1, solving the most precise common unifier of G_1 and G_2. This step may apply repeatedly until G_2 is an instance of G_1. We are sure this loop terminates because there exists a most general TYPOL term, the free variable. Then when G_2 is an instance of G_1, we just don't expand it, because we are already building the partial evaluation for G_1, which is more general. This method, though heavy to implement, prevents the infinite unfolding problem. Of course, the technical difficulty is transferred into the "close to" relation. Other works such as [Bruynooghe 87][Turchin 87], propose related methods.

3.4.2. *Backwards unification*

Another classical problem, specific to logic programs, is side effects and backwards unification. If we keep using "clean" predicates in our programs, this problem doesn't arise. However this is not the case. This is a known problem analysed for instance in [Venken 87][Kursawe 87]. In our method, we just not bother about the problem. We only give here some possible ideas. We say that a predicate has a side effect when this predicate, when executed and then backtracked, doesn't give back the same "state" as before its execution. For instance, if we meet the sequence of unsolved PROLOG goals:

value(X) **write**("Side effect here.") **equal**(X, 2)

and that the **value** predicate will only be defined later, as **value**(3), then the normal evaluation when everything is known would print the message before failing, while the result of partial evaluation is, because of backwards unification,

value(2) **write**("Side effect here.")

that will not print the message at evaluation time. If we have a better treatment of these side effects, we might even deal with failure at partial evaluation time, as shown below. Let's suppose the user tells the system the names of the PROLOG predicates that have side effects. We know TYPOL predicates have no intrinsic side effects. For instance, the **plus** predicate has no side effect, while predicates such as **read** or **write** have. Intuitively, a "normal" predicate has the following property: If we cannot expand the predicate P, then we cannot expand any instance of P. But we are not sure that is still true for exotic PROLOG predicates, such as == or even **write**. So we shall also require the user to tell us which predicates are "normal" With these two informations, we can go further in partial evaluation, and more safely because we know where side effects are. The idea is that at each time, the partial evaluator knows where the side effects are. For instance, if the tactic has led us to expanding the proof tree (it's a silly thing, but one never knows!):

$$\frac{\textbf{plus}\langle x, 1, y\rangle \qquad \textbf{fail}}{\vdash_{\text{test}} x}$$

and the system knows that **fail** is "normal", and that **plus** has no side effects, the **fail** may be executed at compile time, and the proof tree is just removed from the partially evaluated program. On the other hand, if there is a side effect somewhere, the rule may not be removed. For instance the following proof tree

$$\frac{\textbf{write}\langle x\rangle \quad \textbf{plus}\langle x, 1, y\rangle \quad \textbf{fail}}{\vdash_{\text{test}} x} \qquad \text{only becomes} \qquad \frac{\textbf{write}\langle x\rangle \quad \textbf{fail}}{\vdash_{\text{test}} x}$$

In the nicest case, if in some proof tree, we meet the following border:

$$\textit{goals} \qquad [\mathbf{x} \mapsto v, \mathbf{b} \mapsto \text{“}true\text{”}], i \vdash \textbf{if b then x:=x+2 else x:=x+1 fi} : \sigma, o \qquad \textit{goals}$$

the tactic $(\mathbf{TC_3})$ should be applied. But, while partially evaluating the predicate through rule (4), we meet something like **equal**<“$true$”, “$true$”, “$false$”>. If the user told us that **equal** is a “normal” predicate, we are allowed to backtrack at compile time (no side effects). This way, the common skeleton disappears. But now, there is only one skeleton that may be applied: the one that uses (3). So the tactic to use is no longer $(\mathbf{TC_3})$, but $(\mathbf{TC_1})$. That means that the separated rule that was generated through (3)

$$\frac{\textbf{plus}\langle v, 2, w\rangle}{[\mathbf{x} \mapsto v, \mathbf{b} \mapsto \text{“}true\text{”}], \emptyset \vdash \textbf{if b then x:=x+2 else x:=x+1 fi} : [\mathbf{x} \mapsto w, \mathbf{b} \mapsto \text{“}true\text{”}], \emptyset}$$

will be applied directly at compile time, and that spares one prolog call at run time.

3.5. Application to our example

As we already said, the (b) proof tree is a common skeleton. But we can go further. After applying our tactics $(\mathbf{TC_1})$ and $(\mathbf{TC_2})$, we obtain the following common skeleton (e):

$$\cfrac{\cfrac{\vdots\ \textbf{read}\langle v, \text{int}\rangle}{[\mathbf{x}\mapsto], \text{in}[v] \vdash \ldots : [\mathbf{x}\mapsto v], \emptyset} \quad \cfrac{\cfrac{\cfrac{\sigma \overset{\text{get}}{\vdash} s_i d\mathbf{x} \to w}{\sigma \vdash \textbf{deref x} : w} \quad \textbf{write}\langle w\rangle}{\sigma, \emptyset \vdash \textbf{output}\ldots : \sigma', \text{out}[w]} \quad \cfrac{}{\sigma', \emptyset \vdash ; : \sigma', \emptyset}\ \vdots}{[\mathbf{x}\mapsto v], i \vdash \textbf{if}\ldots : \sigma, o_2} \quad \cfrac{\vdots\ o = o_2 \oplus \text{out}[w]}{\sigma, \emptyset \vdash \textbf{output}\ldots; : \sigma', \text{out}[w]}}{[\mathbf{x} \mapsto v], i \vdash \textbf{if}\ldots;\ldots; : \sigma', o}}{[\mathbf{x} \mapsto], \text{in}[v|i] \vdash \textbf{input x int;}\ldots;\ldots; : \sigma', o}}{\vdash \textbf{begin int x;}\ldots;\ldots;\ldots; \textbf{end} : \text{in}[v|i], o}$$

Now, in application of tactic $(\mathbf{TC_3})$, two more proof trees are built for the predicate about **“if”**, because two rules apply, (3) and (4). We shall only show the two simplified rules obtained after tactic $(\mathbf{TC_4})$:

$$(f') \qquad [\mathbf{x} \mapsto 9], \emptyset \vdash \textbf{if}\ldots\textbf{then}\ldots\textbf{else}\ldots\textbf{fi} : [\mathbf{x} \mapsto 0], \emptyset$$

$$(g') \qquad \frac{\textbf{equal}\langle v, 9, \text{“}false\text{”}\rangle \qquad \textbf{add}\langle v, 1, w\rangle}{[\mathbf{x} \mapsto v], \emptyset \vdash \textbf{if}\ldots\textbf{then}\ldots\textbf{else}\ldots\textbf{fi} : [\mathbf{x} \mapsto w], \emptyset}$$

Now we complete tactic $(\mathbf{TC_3})$. The answer to

$$\mathrm{X} \quad = \quad [\mathbf{x} \mapsto v], i \vdash \textbf{if}\ldots\textbf{then}\ldots\textbf{else}\ldots\textbf{fi} : \sigma, o_2$$

is in all cases an instance of

$$\begin{aligned} \mathrm{X}' &= \{[\mathbf{x} \mapsto 9], \emptyset \vdash \textbf{if}\ldots\textbf{then}\ldots\textbf{else}\ldots\textbf{fi} : [\mathbf{x} \mapsto 0], \emptyset\} \bigcap \{[\mathbf{x} \mapsto v], \emptyset \vdash \textbf{if}\ldots\textbf{then}\ldots\textbf{else}\ldots\textbf{fi} : [\mathbf{x} \mapsto w], \emptyset\} \\ &= [\mathbf{x} \mapsto v'], \emptyset \vdash \textbf{if}\ldots\textbf{then}\ldots\textbf{else}\ldots\textbf{fi} : [\mathbf{x} \mapsto w'], \emptyset \end{aligned}$$

so that X may be unified with X' in proof tree (e). This gives more information to go on partially executing (e) further, and we obtain, after simplification:

$$(h') \qquad \frac{\mathbf{read}{<}v, \mathrm{int}{>} \quad [\mathbf{x} \mapsto v], \emptyset \vdash \mathbf{if}...\mathbf{then}...\mathbf{else}...\mathbf{fi} : [\mathbf{x} \mapsto w], \emptyset \quad \mathbf{write}{<}w{>}}{\vdash \mathbf{begin\ int\ x;}...;...;...;\ \mathbf{end} : \mathrm{in}[v], \mathrm{out}[w]}$$

Then we apply tactic $\mathbf{(TC_5)}$, about syntax elimination. Here, this is equivalent to say that the source program is of no use in the compiled code. We obtain the following partially evaluated program (t_1 and t_2 are new identifiers, to distinguish their predicates):

$$(h'') \qquad \frac{\mathbf{read}{<}v, \mathrm{int}{>} \quad v \vdash \mathrm{t}_2 : w \quad \mathbf{write}{<}w{>}}{\vdash \mathrm{t}_1 : \mathrm{in}[v], \mathrm{out}[w]}$$

$$(f'') \qquad 9 \vdash \mathrm{t}_2 : 0$$

$$(g'') \qquad \frac{\mathbf{equal}{<}v, 9, \text{“}false\text{”}{>} \quad \mathbf{add}{<}v, 1, w{>}}{v \vdash \mathrm{t}_2 : w}$$

4. PROOF OF EQUIVALENCE (CORRECTNESS)

In the following, we only expect some basic knowledge of PROLOG semantics. What we want to prove is that our tactics give us a partial evaluation (p.e.), in the sense we defined earlier. So we suppose that we have partially evaluated the predicate Q, which has some information missing that will be given at run time only. This missing information will be represented by a PROLOG environment $\rho_{i(nitial)}$, *i.e.* a mapping from variables to prolog terms. When the information is known, the actual predicate to solve will be: $\mathrm{Q}_i = \rho_i \bullet \mathrm{Q}$. In PROLOG, the composition of such mappings is natural, and it is associative and commutative. Our method consists in proving that each step of our tactic preserves the correctness. We already saw that the Q $\cup$ P program is a correct p.e. (*cf* degenerate cases above). Suppose now that we obtained some set of skeletons S_n, such that $\mathrm{S}_n \cup$ P is a correct p.e. Suppose that we may apply some step of our tactic, yielding new skeletons S_{n+1}. All we need to prove is that $\mathrm{S}_{n+1} \cup$ P is a correct p.e. We always add the initial program (P) because, since the S_i skeletons are intermediate steps, they may call to goals whose definition is in (P). Three remarks before we start:

- Side effects are still a problem we won't examine. If sometime we talk about backtracking, we shall suppose *everything* is undone then. Also, exotic predicates such as **var**, that tests if its argument is a free variable or not, are not considered.
- In our example ASPLE program above, the missing information is not in the starting question (a), but in the **read** predicate. So this missing information is not an instantiation of variables in (a). But it is obvious that it is equivalent. Just consider that the answer to the **read** is a given extra argument of (a).
- We shall treat completely the tactic $\mathbf{(TC_1)}$. We shall skip the tactic $\mathbf{(TC_2)}$. We shall only underline the differences in proving tactic $\mathbf{(TC_3)}$. Tactics $\mathbf{(TC_4)}$ and $\mathbf{(TC_5)}$ obviously preserve correctness.

4.1. Definitions and lemmas

Definition 1: Two predicates P_1 and P_2 unify *iff* there exists a smaller environment ρ such that

$$\rho \bullet \mathrm{P}_1 = \rho \bullet \mathrm{P}_2$$

Sometimes, ρ is called the most general unifier (m.g.u.). We assume that we never have α-conversion problems, *i.e.* all variables receive different names if they are different.

Definition 2: PROLOG semantics says that an inference rule $\frac{\mathrm{LP}}{\mathrm{Y}}$ can expand a predicate X *iff* X unifies with Y, and this gives an environment ρ. The result of the expansion is then

$$\frac{\rho \bullet \mathrm{LP}}{\rho \bullet \mathrm{X}} \qquad (\ = \frac{\rho \bullet \mathrm{LP}}{\rho \bullet \mathrm{Y}})$$

Lemma 1: If a rule $\frac{\mathrm{LP}}{\mathrm{Y}}$ expands $\rho \bullet \mathrm{X}$, then it expands X too.
Proof: If the rule expands $\rho \bullet \mathrm{X}$, then there exists ρ' such that

$$\rho' \bullet \rho \bullet \mathrm{X} = \rho' \bullet \mathrm{Y}$$

Since these two PROLOG terms are equal, we may apply ρ to both. But $\rho \bullet \rho = \rho$. So $\rho' \bullet \rho \bullet \mathrm{X} = \rho' \bullet \rho \bullet \mathrm{Y}$. This shows that X and Y unify, at least with the environment $\rho' \bullet \rho$.

Definition 3: Two environments ρ_1 and ρ_2 are compatible *iff* there is no contradiction (such as $5 = 6$) in the transitive closure of their union, $\rho_1 \bullet \rho_2$.

Lemma 2: The predicate $\rho_1 \bullet \mathrm{X}$ unifies with $\rho_2 \bullet \mathrm{X}$ *iff* ρ_1 and ρ_2 are compatible.
Proof: Unifies with $\rho_1 \bullet \rho_2$.

Lemma 3: If Y and $\rho_1 \bullet \mathrm{X}$ unify, giving environment ρ, and if Y and $\rho_2 \bullet \rho_1 \bullet \mathrm{X}$ unify, giving ρ'. Then ρ, ρ_1 and ρ_2 are compatible, and $\rho' \subseteq \rho \bullet \rho_2$.
Proof: By hypothesis

$$\rho' \bullet \mathrm{Y} = \rho' \bullet \rho_2 \bullet \rho_1 \bullet \mathrm{X}$$

Then, we may apply again ρ_2

$$\rho' \bullet \rho_2 \bullet \mathrm{Y} = \rho' \bullet \rho_2 \bullet \rho_1 \bullet \mathrm{X}$$

And since ρ is the smallest environment that unifies Y and $\rho_1 \bullet \mathrm{X}$, we know that $\rho \subseteq \rho' \bullet \rho_2$. We are allowed to apply ρ_1 to both sides of this inclusion because it is compatible with both sides. We then get:

$$\rho \subseteq \rho_1 \bullet \rho \subseteq \rho_1 \bullet \rho_2 \bullet \rho'$$

which shows that ρ is compatible with $\rho_1 \bullet \rho_2$.
Corollary 3.1: If $\rho_2 \bullet \rho_1 \bullet \mathrm{X}$ is solved, giving additional environment ρ', then, there exists a way to solve $\rho_1 \bullet \mathrm{X}$, that gives ρ, such that ρ, ρ_1 and ρ_2 are compatible.
Proof: By applying (Lemma 3) to the whole proof tree of $\rho_2 \bullet \rho_1 \bullet \mathrm{X}$.

Lemma 4: Converse of (Lemma 3). If Y and $\rho_1 \bullet \mathrm{X}$ unify, giving ρ, and if ρ , ρ_1 and ρ_2 are compatible, then Y and $\rho_2 \bullet \rho_1 \bullet \mathrm{X}$ unify, giving ρ', and $\rho' \subseteq \rho \bullet \rho_2$.
Proof: Starting from $\rho \bullet \mathrm{Y} = \rho \bullet \rho_1 \bullet \mathrm{X}$ and multiplying both sides by ρ_2 (compatible), then again by ρ_2, we get

$$(\rho \bullet \rho_2) \bullet \mathrm{Y} = (\rho \bullet \rho_2) \bullet (\rho_2 \bullet \rho_1 \bullet \mathrm{X})$$

4.2. Proof of ($\mathbf{TC}_1$)

Let's suppose that, after n steps of tactic, the current p.e. skeleton is S_n, which is correct, *i.e.*: The normal evaluation tree of Q_i, T_i, is equivalent to the evaluation of $\rho_i \bullet \mathrm{Q}$ using S_n first, then using rules from (P). Suppose the border of S_n contains a predicate that can be expanded in application of tactic ($\mathbf{TC}_1$). This gives a new p.e. skeleton S_{n+1}. We have to prove that S_{n+1} is also correct.

The S_n p.e. skeleton was built from Q. The successive expansions of Q are summed in an environment, named ρ_n. Let's name X the predicate that will be expanded to obtain S_{n+1}. X is the original shape of this predicate, *i.e.* the way it is written in the rule from which it came. So

$$S_n = \dfrac{\underline{\rho_n \bullet A_1} \quad \underline{\rho_n \bullet X} \quad \underline{\rho_n \bullet A_2}}{\rho_n \bullet Q}$$

Where all the predicates from the border of S_n that are before X are grouped in A_1, and after in A_2. Now we suppose that the rule $\frac{LP}{Y}$ is the only rule that expands $\rho_n \bullet X$, giving the environment ρ_{n+1}. Then we know the value of S_{n+1} (Definition 2):

$$S_{n+1} = \dfrac{\underline{\rho_{n+1} \bullet \rho_n \bullet A_1} \quad \underline{\rho_{n+1} \bullet LP} \quad \underline{\rho_{n+1} \bullet \rho_n \bullet A_2}}{\rho_{n+1} \bullet \rho_n \bullet Q}$$

By hypothesis, the evaluation of Q_i (gives T_i), is equivalent to the evaluation of Q_i with S_n first, then with (P). We are going to prove case by case that this latter evaluation is equivalent to evaluating Q_i with S_{n+1} first, then with (P). If ρ_i and ρ_n are compatible, then S_n applies to Q_i (Lemma 2), and gives a proof tree whose border is

(B_1) $\qquad \rho_i \bullet \rho_n \bullet A_1 \qquad \rho_i \bullet \rho_n \bullet X \qquad \rho_i \bullet \rho_n \bullet A_2$

Also, if ρ_i and $\rho_n \bullet \rho_{n+1}$ are compatible, then S_{n+1} applies to Q_i and gives a proof tree whose border is

(B_2) $\qquad \rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1 \qquad \rho_i \bullet \rho_{n+1} \bullet LP \qquad \rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_2$

Case 1: If ρ_i and ρ_n are not compatible. Then the evaluation by S_n fails (Lemma 2). Then also, ρ_i and $\rho_n \bullet \rho_{n+1}$ are incompatible too (Definition 3), so that the evaluation by S_{n+1} fails too. In that case, the two evaluations are equivalent.

Case 2: If ρ_i and ρ_n are compatible, but not ρ_i and $\rho_n \bullet \rho_{n+1}$. Then evaluation by S_{n+1} fails. Let's see what happens evaluating by S_n. The sequence of predicates to solve is the border (B_1). Either $\rho_i \bullet \rho_n \bullet A_1$ fails, and S_n is equivalent to S_{n+1}, or it succeeds, giving an environment ρ_a that is propagated by PROLOG. The new border to solve is:

(B_3) $\qquad \rho_a \bullet \rho_i \bullet \rho_n \bullet X \qquad \rho_a \bullet \rho_i \bullet \rho_n \bullet A_2$

We know that $\frac{LP}{Y}$ is the only rule that expands $\rho_n \bullet X$. By (Lemma 1), no other rule may expand $\rho_a \bullet \rho_i \bullet \rho_n \bullet X$. But if it expands $\rho_a \bullet \rho_i \bullet \rho_n \bullet X$, since it expands also $\rho_n \bullet X$ with ρ_{n+1}, we get by (Lemma 3) that ρ_{n+1} is compatible with $\rho_a \bullet \rho_i \bullet \rho_n$. This is in contradiction with the hypothesis. Evaluation by S_n fails too.

Case 3: If ρ_i, ρ_n and ρ_{n+1} are compatible, we have to show that the behaviors of (B_1) and (B_2) are equivalent. If $\rho_i \bullet \rho_n \bullet A_1$ fails, then it is trivial. If it succeeds, let's call ρ_a^1 the result. If ρ_a^1 is compatible with $\rho_i \bullet \rho_n \bullet \rho_{n+1}$, then refer to (Case 5). If it doesn't, backtracking occurs on A_1, to give another environment ρ_a^k that is compatible. Refer then to (Case 5) too. If no backtracking gives a compatible ρ_a, then refer to (Case 4).

Case 4: In that case, the execution, if performed through S_n, goes on like this: No backtracking on A_1 makes X succeed, so the sequence of predicates (B_1) fails. But there was no other choice

than S_n to expand Q_i. So execution through S_n fails. We have thus to show that the execution through S_{n+1} fails too. Let's suppose that there is a way of solving $\rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1$, that gives environment ρ'. Then, by (Corollary 3.1), There exists a solution of predicate $\rho_i \bullet \rho_n \bullet A_1$, with the environment ρ_a that is compatible with $\rho_i \bullet \rho_{n+1}$. This is in contradiction with the fact that there is no way of solving $\rho_a^k \bullet \rho_i \bullet \rho_n \bullet X$

Case 5: The execution through S_n, finally finds the first compatible ρ_a^k. Then the rule $\frac{LP}{Y}$ expands $\rho_a^k \bullet \rho_i \bullet \rho_n \bullet X$, and is the only one, from (Lemma 1). The rest of the execution is then equivalent to

$$(B_4) \qquad \rho_a^k \bullet \rho_i \bullet \rho_{n+1} \bullet LP \qquad \rho_a^k \bullet \rho_i \bullet \rho_n \bullet \rho_{n+1} \bullet A_2$$

If some backtrack occurs in (B_4), let's call ρ_a^l, ρ_a^m, ..., the next compatible solutions of A_1. Now, when evaluation through S_{n+1}, let's first show that the successive solutions to $\rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1$ are exactly ρ_a^k, ρ_a^l, ρ_a^m, ... By (Lemma 1), we get that every sequence of rules application (proof tree) solving $\rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1$ is also a solution to $\rho_i \bullet \rho_n \bullet A_1$, and of course its generated environment is compatible with ρ_{n+1}. Conversely, if ρ_a^i is a solution of $\rho_i \bullet \rho_n \bullet A_1$ that is compatible, then, by (lemma 4), $\rho_a^i \bullet \rho_{n+1}$ is also a solution of $\rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1$. Now, for every ρ_a^i found, the rest of the execution is

$$(B_5) \qquad \rho_a^k \bullet \rho_i \bullet \rho_{n+1} \bullet LP \qquad \rho_a^k \bullet \rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_2$$

and $(B_4) \Longleftrightarrow (B_5)$.

4.3. Proof of $(\mathbf{TC_3})$

Let $\rho_n \bullet X$ be a predicate on the border of S_n, for which we are going to apply tactic $(\mathbf{TC_3})$. Let R_1, R_2, etc..., be the rules in (P) that expand $\rho_n \bullet X$. For each rule, this generates an environment, ρ_{n+1}^1, ρ_{n+1}^2, etc... So the newly generated rules are:

$$(R'_j) \qquad \frac{\rho_{n+1}^j \bullet LP^j}{\rho_{n+1}^j \bullet Y^j}$$

We have to prove that the execution of Q_i through S_n, then (P), is the same as through S_n, then through (P) plus the rules R'_j. Both executions are equivalent until we get to the border (B_3). Then, if we apply rules from (P) the applicable rules are, in order, the rules R_j such that ρ_{n+1}^j is compatible with $\rho_a \bullet \rho_i \bullet \rho_n$. after application, the corresponding border is

$$(B_6) \qquad \rho_a \bullet \rho_i \bullet \rho_{n+1}^j \bullet LP^j \qquad \rho_a \bullet \rho_i \bullet \rho_n \bullet \rho_{n+1}^j \bullet A_2$$

Now if we choose the other way, *i.e.* if we apply the R'_j rules first, the ones that will be applied are the ones whose ρ_{n+1}^j is compatible with $\rho_a \bullet \rho_i \bullet \rho_n$, *i.e.* exactly the same as before. The corresponding border is also the same.

After applying the tactic to rules R'_j (*cf* previous section), the bottom line of each of the skeletons is no longer $\rho_{n+1}^j \bullet Y^j = \rho_{n+1}^j \bullet \rho_n \bullet X$, but something more precise that we shall write $\rho^j \bullet X$. If we call:

$$\rho_S = \bigcap_j \rho^j$$

we can prove, with the same method as in the proof of $(\mathbf{TC_1})$, that the ρ_S environment may be propagated to the S_n tree itself. ρ_S is the most precise common unifier between all the possible shapes of the resulting predicates. Of course, doing this removes at compile time some computations that would always fail at run time, therefore losing eventual side effects.

5. Implementing our tactic

Our tactic is implemented as a set of TYPOL rules that are added to our proof tree builder. What we show here is only a partial implementation, leaving some work to the user. A more complete implementation is in progress. We only need a hint about the use of the proof-tree builder: It uses predicates such as "command" that calls for an atomic command on the current state (σ). and utility predicates such as "next_goal" that gives the next unsolved goal from the border, in left-to-right order. These TYPOL rules must be read in a very operational way. In fact, they should rather be written in PROLOG. We just found it convenient, though abusive, to use TYPOL. The first rule is to traverse the proof tree in depth-first left-to-right order to find goals to which a tactic applies. Then, there is a rule to implement each tactic:

$$\frac{\overset{\text{next\_goal}}{\sigma_1 \vdash \sigma_2} \qquad \overset{\text{next\_tactic}}{\sigma_2 \vdash \text{“Try expand”}}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Partial eval”}}}$$

($\mathbf{TC}_4$)
$$\frac{\overset{\text{command}}{\sigma_1 \vdash \text{“Simplify”} : \sigma_2} \qquad \text{store\_rule}\langle\sigma_2\rangle}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Partial eval”}}}$$

($\mathbf{TC}_2$)
$$\frac{\text{executable\_goal}\langle\sigma_1\rangle \qquad \overset{\text{command}}{\sigma_1 \vdash \text{“Execute”} : \sigma_2} \qquad \overset{\text{next\_tactic}}{\sigma_2 \vdash \text{“Partial eval”}}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Try expand”}}}$$

($\mathbf{TC}_1$)
$$\frac{\overset{\text{command}}{\sigma_1 \vdash \text{“Expand”} : \sigma_2} \qquad \overset{\text{number\_rules}}{\sigma_2 \vdash 1} \qquad \overset{\text{command}}{\sigma_2 \vdash \text{“Apply”} : \sigma_3} \qquad \overset{\text{next\_tactic}}{\sigma_3 \vdash \text{“Partial eval”}}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Try expand”}}}$$

(no tactic applies)
$$\frac{\overset{\text{command}}{\sigma_1 \vdash \text{“Expand”} : \sigma_2} \qquad \overset{\text{number\_rules}}{\sigma_2 \vdash 0} \qquad ! \qquad \textbf{fail}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Try expand”}}}$$

($\mathbf{TC}_3$)
$$\frac{\overset{\text{next\_goal}}{\sigma_3 \vdash \sigma_4} \qquad \overset{\text{command}}{\sigma_4 \vdash \text{“Apply”} : \sigma_5} \qquad \overset{\text{next\_tactic}}{\sigma_5 \vdash \text{“Partial eval”}} \qquad \textbf{fail}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Try expand”}}} \qquad \overset{\text{focus\_on\_goal}}{\sigma_1 \vdash \sigma_2} \qquad \overset{\text{command}}{\sigma_2 \vdash \text{“Expand”} : \sigma_3}$$

This system works, but very slowly. This is the drawback of such a precise control on TYPOL execution. But we hope we shall soon be able to speed it up, although it is not in the scope of this paper. Major improvements may also be found by using MU-prolog, [Naish 85], since the essence of the tactic is to delay the expansion of a predicate until the missing information is given, *i.e.* until some variables are instantiated. This is close to the notion of **"wait"** facility found in some PROLOG interpreters.

We have applied this partial evaluation to many examples. One is a bigger ASPLE program that defines the factorial function. Another is the same factorial function written in ML, partially

executed by the TYPOL ML interpreter. The last example we shall show is about improving the style of a TYPOL program by partial evaluation.

5.1. Application to the ASPLE factorial program

The ASPLE text for the factorial function is slightly more complicated than our previous small example. It is:

```
begin
    int x, y, z;
    input x int;
    y:=1;
    z:=1;
    if (deref x <> 0)
      then while (deref z <> deref x) do
                z:=deref z +1;
                y:=deref y × deref z;
           end;
    fi;
    output deref y int;
end
```

While partially evaluating the corresponding predicate, we meet the predicate:

$$(W_1) \qquad [\mathbf{x} \mapsto a, \mathbf{y} \mapsto 1, \mathbf{z} \mapsto 1], i_1 \vdash \textbf{while...do...end} : \sigma_1, o_1$$

to solve by tactic $\mathbf{(TC_3)}$. But while doing that, across rule (5), we again meet the predicate:

$$(W_2) \qquad [\mathbf{x} \mapsto a, \mathbf{y} \mapsto 2, \mathbf{z} \mapsto 2], i_2 \vdash \textbf{while...do...end} : \sigma_2, o_2$$

Since this could lead to infinite unfolding, our system chooses to solve instead:

$$(W_3) \qquad [\mathbf{x} \mapsto a, \mathbf{y} \mapsto b, \mathbf{z} \mapsto c], i_2 \vdash \textbf{while...do...end} : \sigma_2, o_2$$

Then, while solving (W_3), we find again the same predicate (W_3) to solve, as a premise of itself. As we saw before, the system chooses just to do nothing. When the end of a $\mathbf{(TC_3)}$ tactic is reached, since the (W_3) predicate is found as a condition to itself, the computation of the most precise common unifier X' becomes a fixpoint equation:

$$\begin{aligned} X' &= [\mathbf{x} \mapsto a, \mathbf{y} \mapsto b, \mathbf{z} \mapsto c], i \vdash \textbf{while...do...end} : \sigma, o \\ &= [\mathbf{x} \mapsto d, \mathbf{y} \mapsto e, \mathbf{z} \mapsto d], \emptyset \vdash \textbf{while...do...end} : [\mathbf{x} \mapsto d, \mathbf{y} \mapsto e, \mathbf{z} \mapsto d], \emptyset \\ &\quad \bigcap [\mathbf{x} \mapsto a, \mathbf{y} \mapsto f, \mathbf{z} \mapsto g], i \vdash \textbf{while...do...end} : \sigma, o \end{aligned}$$

giving: $$X' = [\mathbf{x} \mapsto a, \mathbf{y} \mapsto f, \mathbf{z} \mapsto g], \emptyset \vdash \textbf{while...do...end} : [\mathbf{x} \mapsto a, \mathbf{y} \mapsto b, \mathbf{z} \mapsto a], \emptyset$$

As we said before, this fixpoint equation is equivalent to proving some theorem on X′ using structural induction. Since only b is significant in the result, tactic $\mathbf{TC_5}$ finally gives:

$$(F_1) \qquad \frac{\textbf{read}\langle a, \text{int}\rangle \qquad a \vdash t_2 : b, c \qquad \textbf{write}\langle c\rangle}{\vdash t_1 : \text{in}[a], \text{out}[c]}$$

$$(F_2) \qquad \frac{\textbf{different}\langle a, 0, \text{“}true\text{”}\rangle \qquad a, 1, 1 \vdash t_3 : c}{a \vdash t_2 : a, c}$$

$$(F_3) \qquad 0 \vdash t_2 : 1, 1$$

$$(F_4)\qquad \frac{\textbf{different}<c, a, \text{“}true\text{”}> \qquad \textbf{add}<c, 1, c_1> \qquad \textbf{times}<b, c_1, b_1> \qquad a, b_1, c_1 \vdash t_3 : d}{a, b, c \vdash t_3 : d}$$

$$(F_5)\qquad a, b, a \vdash t_3 : b$$

where the (F_2) and (F_3) rules deal with the **if** subtree, that is named t_2, and (F_4) (F_5) deal with the **while** subtree (t_3). It is obvious here that the performances in time and space of the compiled program are much better than with the source program.

5.2. Application to the ML factorial function

We just give the ML text for factorial, which is:

```
letrec fact=λx.if (equal (x,0))
                  then 1
                  else (times (x,(fact (minus (x,1)))))
in (print (fact (read)))
```

Notice that it uses another method, where the **while** is replaced by recursive calls. This is because the language has changed, and so has the programming style. Also, since ML provides a **minus** operator, the algorithm itself has changed to a simpler one. The partial evaluation is complicated by the fact that ML semantics in TYPOL uses infinite terms, but with some care, we obtain the following "compiled" version:

$$(F'_1)\qquad \frac{\textbf{read}<a, \text{int}> \qquad a \vdash t_2 : b \qquad \textbf{write}<b>}{\vdash t_1}$$

$$(F'_2)\qquad 0 \vdash t_2 : 1$$

$$(F'_3)\qquad \frac{\textbf{different}<a, 0, \text{“}true\text{”}> \qquad \textbf{minus}<a, 1, a_1> \qquad a_1 \vdash t_2 : b_1 \qquad \textbf{times}<a, b_1, b>}{a \vdash t_2 : b}$$

It is remarkable that nothing in these rules is related to ML any more, like nothing was related to ASPLE in the previous section. All that remains is purely TYPOL or PROLOG predicates. Besides, the two compiled programs are very much alike. The only difference reflects the two different underlying algorithms. On the other hand, the difference between **while** and recursive calls has disappeared. Here also, the performances in time and space of the compiled program are much better.

5.3. Application for improving a TYPOL program

This last example is about applying our tactic to a TYPOL program that has a heavy style. What we call heavy style is when one finds too many predicate calls that could be unfolded. Also, it may be bad style to call explicitly the "eq" predicate, since one may directly perform the unification in the text of the program. This can be done using our tactic. Let's consider the following heavy definition of the member predicate:

$$\frac{\text{first}<l, x>}{\text{member}<x, l>}$$

$$\frac{\text{follow}<l, r> \qquad \text{member}<x, r>}{\text{member}<x, l>}$$

$$\frac{\text{eq}<l, []> \qquad \textbf{fail}}{\text{member}<x, l>}$$

$$\frac{\text{eq}<l, [a|\_]> \qquad \text{eq}<a, x>}{\text{first}<l, x>}$$

$$\frac{\text{eq}<l, [\_|b]> \qquad \text{eq}<b, r>}{\text{follow}<l, r>}$$

$$\text{eq}<x, x>$$

Our tactic is started with the question: member$<x, l>$. If we suppose that the tactic knows there are no side effects and all predicates are "normal", then the result is (P'):

$$\text{member}<x, [x|r]>$$

$$\frac{\text{member}<x, r>}{\text{member}<x, [y|r]>}$$

This second program looks much simpler that the first one, and is more efficient.

6. CONCLUSION

This tactic to define a partial evaluation for inference rules seems to apply correctly to TYPOL or PROLOG programs. A main interest is that it gives us something like a compiler, if we provide it an interpreter. What seems pleasant to us is the independance of the code from the source. For instance, nothing indicates that some rules are the compilation of an ASPLE or an ML program. All we get is a very pure TYPOL program, where all source language dependent predicates and constructs have been removed. The compiled files look pretty also because, as we said, only the necessary predicate calls are not unfolded. All the other ones are compacted, and this saves a lot of execution time that was spent in predicate calls. Also, sometimes the reason for a predicate to fail is discovered much earlier, and that saves bactracking.

The main drawback is that it may seem strange to compile into such a high level language, while one would expect assembly language instead! Our answer is that now, all we need is a kind of compiler from TYPOL to assembler, since we have the compilers from anywhere to TYPOL. In our team, some work is already done in that way. Also, although the tactic itself is written in TYPOL, we have not big hopes about applying it to itself, mainly because the tactic uses a lot of exotic PROLOG predicates and side effects. Perhaps life would be easier if we partially executed PROLOG files directly, especially with some "wait" or "freeze" facilities.

As a comparison with the work of N.D.Jones team [Jones 87a][Jones 87b], our partial evaluator also knows how to avoid infinite loops, even the disguised ones, such as $f(1,1)$ calling $f(1,2)$, that calls $f(1,3)$, etc... Also, we have the equivalent of simplifying the store by removing the list of the names of the variables, which are finally useless, and we do this automatically. It seems easier to work with inference rules rather than with LISP, but we cannot apply our evaluator to itself yet.

We would also like to try a comparison with the work of Y.Futamura and K.Nogi, GPC [Futamura 87], where a partial evaluation of a ML-like program gives an improvement of complexity. The method relies on remembering, inside a conditional branch, whether the condition was true or false. In our system, we have to consider two cases. Either the condition may be remembered through unification, and then our system naturally takes profit of this information; in the other case (like for the "difference" relation), we cannot easily use the information, except by implementing the equivalent of a "freeze" predicate. Therefore, from this point of view, we may say our system has "half" the power of the GPC.

REFERENCES

[Bondorf 87b] **Bondorf, A.**, *"Towards a Self-Applicable Partial Evaluator for Term Rewriting Systems"*, D.Bjørner, A.P.Ershov, N.D.Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings North-Holland (1988).

[Bowen 84] **Bowen, D., Byrd, L., Pereira, L., Warren, D.**, *"C-Prolog user's manual, version 1.5"*, Edited by F.Pereira, SRI International, Menlo Park, California, (February 1984).

[Bruynooghe 87] **Bruynooghe, M.**, *"A Framework for the Abstract Interpretation of Logic Programs"*, Katholieke Universiteit Leuven, Report CW 62, (October 1987)

[Clément 85] **Clément, D., Despeyroux, J. and Th., Hascoët, L., Kahn, G.,** *"Natural semantics on the computer"*, Rapport de recherche INRIA, N^o 416, (June 1985).

[Despeyroux 83] **Despeyroux, Th.,** *"Spécifications sémantiques dans le système MENTOR"*, Thèse, Université Paris XI, (1983).

[Despeyroux 84] **Despeyroux, Th.,** *"Executable specification of static semantics"*, Semantics of data types, Lecture Notes in Computer Science, Vol. 173, (June 1984).

[Ershov 77a] **Ershov, A.P.**, *"On the partial computation principle"*, Information Processing Letters, 6(2):pp 38–41, (April 1977).

[Futamura 71] **Futamura, Y.**, *"Partial evaluation of computation process. An approach to a compiler-compiler"*, Systems, Computers, Controls, 2(5):pp 45–50, (1971).

[Futamura 87] **Futamura, Y.**, *"Generalised Partial Computation"*, D.Bjørner, A.P.Ershov, N.D.-Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings North-Holland (1988).

[Gallagher 87] **Gallagher, J., Codish, M.**, *"Specialisation of Prolog and FCP Programs Using Abstract Interpretation"*, D.Bjørner, A.P.Ershov, N.D.Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings New Generation Computing Journal (1988).

[Hascoët 87] **Hascoët, L.**, *"A tactic-driven system for building proofs"*, Actes du 7^{eme} séminaire Programmation en Logique, Trégastel, France, May 25–27, (1988), also INRIA Research Report N^{o}770, (1987).

[Jones 87a] **Jones, N.D., Setsoft, P., Søndergaard, H.,** *"MIX: A self-applicable partial evaluator for experiments in compiler generation"*, Report N^o 87/8, DIKU, University of Copenhagen, (May 1987).

[Jones 87b] **Jones, N.D.,** *"Towards Automating the Transformation of Programming Language Specifications into Implementations"*, Part I: 55p, Part II: 74p, (1987)

[Kahn G. 88] **Kahn, G. et al.,** *"CENTAUR. The system"*, INRIA Report N^o 777 (December 1987).

[Kahn K. 84a] **Kahn, K.M., Carlsson, M.,** *"The compilation of PROLOG programs without the use of PROLOG compiler*, International conference on fifth generation computer systems, Tokyo, Japan, pp 348–355, (1984).

[Kanamori 86] **Kanamori, T.,** *"Soundness and completeness of extended execution for proving properties of PROLOG programs"* In France-Japan Artificial intelligence and computer science symposium, Tokyo, Japan, pp 219–238, (1986).

[Kursawe 87] **Kursawe, P.**, *"Pure Partial Evaluation and Instantiation"*, D.Bjørner, A.P.Ershov, N.D.Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings North-Holland (1988).

[Launchbury 87] **Launchbury, J.**, *"Projections for specialisation"*, D.Bjørner, A.P.Ershov, N.D.-Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings North-Holland (1988).

[Naish 85] **Naish, L.,** *"MU-Prolog 3.2 reference manual"*, Technical report 85-11, University of Melbourne, (1985).

[Schmidt 87] **Schmidt, D.A.**, *"Static Properties of Partial Reduction"*, D.Bjørner, A.P.Ershov, N.D.Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings North-Holland (1988).

[Turchin 87] **Turchin, V.F.**, *"The Algorithm of Generalisation in the Supercompiler"*, D.Bjørner, A.P.Ershov, N.D.Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings North-Holland (1988).

[Venken 87] **Venken, R., Demoen, B.**, *"A Partial Evaluation System for Prolog: Some Practical Considerations"*, D.Bjørner, A.P.Ershov, N.D.Jones, editors, Workshop on Partial Evaluation and Mixed Computation, Gl.Avernaes, Denmark, October 1987. Proceedings New Generation Computing Journal (1988).