PARTIAL EVALUATION WITH INFERENCE RULES

Laurent Hascoët
INRIA Sophia-Antipolis
Rte des lucioles 06565 Valbonne, France

Abstract

Suppose we are given a program P in some language L, and a set of inference rules based on unification 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, therefore proposing a new excutable semantics to our rules. This tactic applied to P and to the dynamic semantics of L yields classicly a new set of specialized inference rules that are a compiled version of P. Our partial evaluation tactic proposes and uses some original improvements, which are applicable to the general field of partial evaluation of unification-based languages.

Keywords: Inference Rules, Partial Evaluation, Prolog, Tactic

1. INTRODUCTION

This paper presents a method to achieve partial evaluation of programs written with unification-based inference rules. Most of the previous work on partial evaluation was done on functional languages, such as ML [11)15)25)]. It is a recent trend to study partial evaluation in logic programming. An interesting point is that some well-known problems of partial evaluation for functional languages also arise with logic programming. The partial evaluation method that we present is interesting because of its choices about call unfolding. There is also an obvious relation with work involving abstract interpretation [3)12)]. The most famous application of partial evaluation is the automatic generation of compilers from interpreters. The idea originates from Ershov [7)]and Futamura [10)]. 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. Therefore this paper focuses on the partial evaluation of interpreters. This is natural since our inference rules are mainly dedicated to defining semantic operations on languages, such as interpreting.

Our group is involved in the development of the CENTAUR system [18]. It is a syntax-directed editor with tools to define semantic operations on syntactic trees. These operations are defined with inference rules, in a language called TYPOL [5,6]. We call our method "natural semantics" [4,17]. We felt the need for a tool to build proof trees using TYPOL rules. We have built a first experimental version of this tool [14]. It is written in TYPOL and Prolog, which are good languages for defining prototypes. This tool allows the user to define tactics about how to build a proof tree made with TYPOL rules. After being introduced to the work of N.D.Jones on partial evaluation [15,16], we felt that 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. This is what we describe in this paper.

In the following section, we briefly present the foundations of TYPOL, as well as the reasons for its syntax. Since TYPOL is considered as yet another unification-based language, we distinguish the contribution of this work to TYPOL and to unification-based languages. Section 3 describes our method of partial evaluation. In section 4, we give a proof of the correctness of our method. Section 5 presents the actual implementation of our tactic. We also show examples of compilation using partial evaluation.

2. SPECIFICATION OF DYNAMIC SEMANTICS WITH TYPOL

2.1. Foundations of TYPOL

Recently, many researchers have begun to use a style of semantic specifications advocated by Plotkin [24]. Instead of using denotational semantics, they use axioms and inference rules that define the various semantic predicates to be proved on a given program. A semantic definition is thus identified with a logic, and reasoning with the language is proving theorems within that logic. This presentation has many advantages, for nondeterministic computation for example. According to the works of Gentzen about Natural Deduction [13], the only semantics of these rules and axioms is the set of finite proof trees that can be built from them. Natural Semantics [4,17]is just one example of this style of semantic specification. Among all the ways of presenting a logic, a syntax close to Natural Deduction was chosen. The resulting language is TYPOL.

A TYPOL definition is an unordered collection of rules, consisting of a numerator and a denominator. Variables may occur in a rule, allowing its instantiation. Variables may be restricted to certain types, and TYPOL provides a type-checker. A numerator is itself an unordered collection of premises. If the premises hold, then the denominator, or conclusion, holds. Premises are either sequents or conditions. The denominator is necessarily a sequent. Sequents have two parts, separated by a turnstile symbol $\vdash$. The left part is the antecedent and may be thought of as the environment in which the sequent is to be proved. The right part is a predicate. A condition is a predicate, which is often defined externally. For instance, the following sequent: $\sigma, i \vdash \textsc{stms} : \sigma', o$ means that in the environment σ, i being the input consumed, the following predicate holds: the interpretation of the statements STMS gives a new environment σ' and a list of values output o. Sequents are named, and their name appears over the $\vdash$ symbol. For readability, the most frequent sequent has just no name.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 of the TYPOL rules defining the semantics of ASPLE:

$$(1)\qquad \frac{\sigma, i_1 \vdash \text{STM} : \sigma', o_1 \qquad \sigma', i_2 \vdash \text{STMS} : \sigma'', o_2 \qquad i = i_1 \oplus i_2 \qquad o = o_1 \oplus o_2}{\sigma, i \vdash \text{STM}\,\mathbf{;}\ \text{STMS} : \sigma'', o}$$

$$(2)\qquad \frac{\sigma \vdash \text{EXP} : \text{“}true\text{”} \quad \sigma, i \vdash \text{STM1} : \sigma', o}{\sigma, i \vdash \mathbf{if}\ \text{EXP}\ \mathbf{then}\ \text{STM1}\ \mathbf{else}\ \text{STM2}\ \mathbf{fi} : \sigma', o}$$

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

(4)

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

$$(5)\qquad \frac{\sigma \vdash \text{EXP} : \text{“}false\text{”}}{\sigma, \emptyset \vdash \mathbf{while}\ \text{EXP}\ \mathbf{do}\ \text{STM}\ \mathbf{end} : \sigma, \emptyset}$$

The intuitive meaning of these rules is straightforward. The operation named $\oplus$ represents the concatenation of lists. External predicates (conditions) such as $\oplus$ are defined here in Prolog.

- Rule (1) means that a sequence of statements is interpreted by interpreting the first statement and the rest of the statements. Notice that, in an operational way, the “tail” part of the list is interpreted with a new store that is output by the first statement.
- Rules (2) and (3) define the classical behavior of the **“if”** statement. To express nondeterminism, one just needs to write two rules for the **“if”** construct.
- Rules (4) and (5) are similar, but deal with the **“while”** statement.

2.2. Execution of TYPOL

Problems arise when trying to execute TYPOL specifications to get real type-checkers, interpreters, compilers, etc. The first choice is to compile TYPOL rules in a straightforward way into Prolog, therefore taking the usual conventions on ordering of rules and of premises. The main advantage of this is an easy way to get running interpreters from TYPOL programs. Another classical advantage is speed of execution. This is the way TYPOL has always been used so far. However this implies restrictions on the TYPOL specifications in order to make them run. Non-determinism is now replaced by a classical backtracking mechanism. Problems occur when external routines are called with unknown arguments while they expect ground terms. It is clear that the TYPOL rules above have been written to be executed, therefore their premises being ordered in a particular way.

Therefore we built an alternative tool to execute TYPOL programs [14], which sticks to the theoretical semantics of TYPOL, *i.e.*, proof trees. It allows to build a proof tree, interactively or allowing the user to define himself – with tactics – which premise to expand and with which TYPOL rule. For instance we designed a tactic simulating Prolog depth first strategy, but many other tactics may be defined.

Sticking to TYPOL semantics, the way we define the execution of TYPOL on a given predicate (goal) is just the set of all completely solved proof trees

– 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).

However, because of these side-effect external predicates, we suppose the solving of these predicates respects an intuitive chronology.

The rest of this section will show an example of TYPOL execution. Let's suppose our ASPLE program we want interpreted 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
```

where **deref x** means the value of x, while **x** alone means the adress of x in the store. Let's suppose that we shall type "3" as the input value, then we may build the evaluation (the proof trees) of the program. Since ASPLE is deterministic (can be proved on the TYPOL file) there is at most one possible proof tree. We can build the proof tree in any order (except for chronology of side effects). 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 **x** is undefined):

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

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), we chose to solve the **"read"**, which asks us for an integer, and we shall type "3" as we said. Then, after some more steps of evaluation, the current proof tree is (c):

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

and the next goal we choose to solve is:

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

Since there are two applicable rules (2) and (3), we shall get two sets of proof trees. However, the set corresponding to (2) is empty, because there is no proof tree yielding that $3 = 9$ returns *"false"*. So we may only apply rule (3). It may be remarked that this is not the way it works with TYPOL rules compiled in Prolog. Instead, the rule (2) is selected, thus leading to a failure and backtracking leads to applying rule (3). The fact that the result is finally the same comes from the fact that this TYPOL file is "executable", *i.e.*, complies with the extra restrictions of section 2.2. Now that rule (3) is applied, the process goes on, and solves the **"else"** part of the **"if"**. At some time, the chosen predicate is **add**$(3, 1, \mathrm{X})$ which is a Prolog predicate, like **read** was. This means that this predicate is solved by calling Prolog, thus instantiating X by 4. From the TYPOL point of view, everything looks as if **add**$(3, 1, 4)$ is a given axiom. When the execution terminates, we obtain the final completely solved proof tree (d):

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

2.3. Contribution of this work to TYPOL

As we said, TYPOL is defined in a theoretical way, by the set of proof trees that can be built from the rules and axioms. This means that there is no notion of order of evaluation, unlike in other unification-based languages, such as Prolog. However, TYPOL is usually interpreted in the Prolog way, leading to painful restrictions. The contribution of this work to TYPOL itself is that partial evaluation of TYPOL files is defined in terms of proof trees, *i.e.*, independently from any execution tactic. The only restrictions are due to side-effect predicates, which should not appear in real TYPOL anyway. In fact, the actual implementation of the partial evaluation tactic slightly depends on the way TYPOL is executed. But minor changes in this implementation will suffice to match other TYPOL execution tactics.

In other words, the partial evaluation of a given goal is a TYPOL file F′, which is equivalent to the original one, independent of the execution tactic. The set of proof trees built from F′ is the same as the ones built from the original TYPOL file on the same goals. Our partial evaluation tactic can be viewed as another TYPOL execution tactic. In fact it is the first "execution" of TYPOL that really sticks to TYPOL official semantics.

2.4. Contribution of this work to unification-based languages

This partial evaluation tactic may be applied when the execution tactic is defined. For instance we may pick depth-first left-to-right strategy. Therefore, some parts of our

method may be of interest for general unification-based languages. Examples are our way to prevent infinite unfolding and combinatorial explosion of the specialized rules.

We also propose a way to use more information when collecting the specialized rules, as we explain in tactic **(TC_3)**. This method sometimes allows the partial evaluator to prove inductive properties on the execution of the program. Then these properties are used on the fly to get more efficient partially evaluated files. It is interesting to compare these parts of our tactic with other partial evaluators for Prolog-like languages.

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 to 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. Here we have to paraphrase it: 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 inference system 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 cannot be decided by the system. The remaining (unknown) input is again a substitution of the variables in the goal. The result of partial evaluation is another set of rules, and the remaining input is given by another goal. When we give a method for partial evaluation, we 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

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

The proof tree(s) T' obtained by executing (P$'$) with input i_2 (and somehow remembering i_1) must be equivalent to the proof tree(s) 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 inefficient, since in that case (P$'$) equals (P).

3.2. 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 expanding 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 set 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. 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.

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'):

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

3.3. Idea of the method: 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 independent of 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 (2) or (3). 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.

3.4. 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 instantiated 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 A. Bondorf's paper [1)], which deals with rewrite rules instead of inference rules.

Tactic (TC_1): When some unsolved goal in the border of the currently built 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 external predicates. 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. This tactic depends on the TYPOL execution strategy.

Tactic (TC_2): When some external predicate in the border of the currently build proof tree may be solved at partial evaluation time, and 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 definition of the external 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, because we are out of clean TYPOL.

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 (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 are given in section 3.6. 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 is given in section 3.6. This can be related to the projection method in [22].

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.

3.5. Classical problems

3.5.1. *Infinite unfolding*

The first classical problem that we meet is infinite unfolding at partial evaluation time. As shown by inference rule (4), 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 $(\mathbf{TC}_1)$ that was applied before should be replaced by tactic $(\mathbf{TC}_3)$. 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 $(\mathbf{TC}_3)$, to a call to G_2 (G_1 close to G_2). Then we have to apply tactic $(\mathbf{TC}_3)$ 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 [3)27)], propose related methods.

3.5.2. *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 always the case. This is a known problem analysed for instance in [21)28)].

When we know that all the predicates involved are "clean", then backwards unification may be rewarding. In the nicest case, if in some proof tree, we meet the following goal:

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

the tactic $(\mathbf{TC_3})$ should be applied. But, while partially evaluating the predicate through rule (3), we meet something like $\mathbf{equal}\langle\text{“}\mathit{true}\text{”}, \text{“}\mathit{true}\text{”}, \text{“}\mathit{false}\text{”}\rangle$. We know then that the set of proof trees going through rule (3) is empty. There is thus only one skeleton that may be applied: the one that uses (2). 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 (2)

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

is applied directly at compile time, and that spares one inference at run time.

3.6. 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):

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

Now, in application of tactic $(\mathbf{TC_3})$, two more proof trees are built for the predicate about **"if"**, because two rules apply, (2) and (3). We shall only show the two simplified rules obtained after tactic $(\mathbf{TC_4})$:

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

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

Now we complete tactic $(\mathbf{TC}_3)$. The answer to

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

is in all cases an instance of

$$\mathrm{X}' = \{[\mathbf{x} \mapsto 9], \emptyset \vdash \mathbf{if}...\mathbf{then}...\mathbf{else}...\mathbf{fi} : [\mathbf{x} \mapsto 0], \emptyset\} \bigcap \{[\mathbf{x} \mapsto v], \emptyset \vdash \mathbf{if}...\mathbf{then}...\mathbf{else}...\mathbf{fi} : [\mathbf{x} \mapsto w], \emptyset\}$$
$$= [\mathbf{x} \mapsto v'], \emptyset \vdash \mathbf{if}...\mathbf{then}...\mathbf{else}...\mathbf{fi} : [\mathbf{x} \mapsto w'], \emptyset$$

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:

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

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):

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

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

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

4. PROOF OF EQUIVALENCE (CORRECTNESS)

In the following, we only expect some basic knowledge of unification-based languages. 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 an environment $\rho_{i(nitial)}$, *i.e.* a mapping from variables to terms. When the information is known, the actual predicate to solve will be: $\mathrm{Q}_i = \rho_i \bullet \mathrm{Q}$. 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 ∪ P program is a correct p.e. (*cf* degenerate cases above). Suppose now that we obtained some set of skeletons S_n, such that S_n∪ 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 S_{n+1}∪ 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. 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 most general 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: 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 $\rho_1 \bullet \rho_2$. Now, since $\rho \bullet \mathrm{Y} = \rho \bullet \rho_1 \bullet \mathrm{X}$, and of the above compatibility, we get by composing both sides with ρ_2 that $\rho \bullet \rho_2$ is a unifier of Y and $\rho_2 \bullet \rho_1 \bullet \mathrm{X}$. Therefore, it is less general than ρ'.
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 is 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 comes. So

$$\mathrm{S}_n = \dfrac{\dfrac{\rho_n \bullet \mathrm{A}_1}{\vdots} \quad \dfrac{\rho_n \bullet \mathrm{X}}{\vdots} \quad \dfrac{\rho_n \bullet \mathrm{A}_2}{\vdots}}{\rho_n \bullet \mathrm{Q}}$$

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 $\frac{\mathrm{LP}}{\mathrm{Y}}$ is the only rule that expands $\rho_n \bullet \mathrm{X}$, giving the environment ρ_{n+1}. Then we know the value of S_{n+1} (Definition 2):

$$\mathrm{S}_{n+1} = \dfrac{\dfrac{\rho_{n+1} \bullet \rho_n \bullet \mathrm{A}_1}{\vdots} \quad \dfrac{\rho_{n+1} \bullet \mathrm{LP}}{\vdots} \quad \dfrac{\rho_{n+1} \bullet \rho_n \bullet \mathrm{A}_2}{\vdots}}{\rho_{n+1} \bullet \rho_n \bullet \mathrm{Q}}$$

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). The various cases we are going to examine depend on the compatibility between ρ_i, ρ_n and ρ_{n+1}. Remark that 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

$$(\mathrm{B}_1) \qquad \rho_i \bullet \rho_n \bullet \mathrm{A}_1 \qquad \rho_i \bullet \rho_n \bullet \mathrm{X} \qquad \rho_i \bullet \rho_n \bullet \mathrm{A}_2$$

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

$$(\mathrm{B}_2) \qquad \rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet \mathrm{A}_1 \qquad \rho_i \bullet \rho_{n+1} \bullet \mathrm{LP} \qquad \rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet \mathrm{A}_2$$

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 $\rho_i \bullet \rho_n \bullet A_1$ has no proof (fails), and S_n is equivalent to S_{n+1}, or it succeeds, giving an environment ρ_a that is propagated. The new border to solve is:

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

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 has a set of proofs SP, we first know that each proof of $\rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1$ corresponds to a proof in SP (Corollary 3.1). then for each proof in SP, see if the resulting environment ρ_a^k is compatible with $\rho_i \bullet \rho_n \bullet \rho_{n+1}$. If so refer to (Case 5) else to (Case 4).

Case 4: In that case, the execution using the selected proof tree to solve $\rho_i \bullet \rho_n \bullet A_1$, if performed through S_n, goes on like this: It is impossible to solve $\rho_a^k \bullet \rho_i \bullet \rho_n \bullet X$ because the only applicable rule gives ρ_{n+1} incompatible with ρ_a^k. The execution performed through S_{n+1} cannot even use the selected proof tree. Thus the behaviors are equivalent.

Case 5: In that case, the execution using the selected proof tree to solve $\rho_i \bullet \rho_n \bullet A_1$, if performed through S_n, may be continued by solving $\rho_a^k \bullet \rho_i \bullet \rho_n \bullet X$ through the only applicable rule (ρ_{n+1} compatible with ρ_a^k, and Lemma 1). The proof tree is completed if we find proofs of

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

The execution performed through S_{n+1} may use the selected proof tree to solve $\rho_i \bullet \rho_{n+1} \bullet \rho_n \bullet A_1$ because of (Lemma 4). The resulting environment is $\rho_a^k \bullet \rho_{n+1}$. Then the proof tree is completed if we find proofs of

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

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

4.3. Proof of (TC_3)

Let $\rho_n \bullet X$ be a predicate on the border of S_n, for which we are going to apply tactic **(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) $$\frac{\rho_{n+1}^j \bullet LP^j}{\rho_{n+1}^j \bullet Y^j}$$

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

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 $\rho^j_{n+1} \bullet Y^j = \rho^j_{n+1} \bullet \rho_n \bullet X$, 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 any order specified by the user. 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 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)

$$\frac{\overset{\text{command}}{\sigma_1 \vdash \text{“Expand”} : \sigma_2} \qquad \overset{\text{number\_rules}}{\sigma_2 \vdash 0} \qquad ! \qquad \textbf{fail}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Try expand”}}}$$

$(\mathbf{TC}_3)$

$$\frac{\overset{\text{next\_rule}}{\sigma_3 \vdash \sigma_4} \qquad \overset{\text{command}}{\sigma_4 \vdash \text{“Apply”} : \sigma_5} \qquad \overset{\text{next\_tactic}}{\sigma_5 \vdash \text{“Partial eval”}} \qquad \textbf{fail}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“Try expand”}}}$$

$$\overset{\text{focus\_on\_goal}}{\sigma_1 \vdash \sigma_2} \qquad \overset{\text{command}}{\sigma_2 \vdash \text{“Expand”} : \sigma_3}$$

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 [23], 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 some 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.

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 (4), 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:

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

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:

$$(\mathrm{F}_1)\qquad \frac{\textbf{read}\langle a, \mathrm{int}\rangle \qquad a \vdash \mathrm{t}_2 : b, c \qquad \textbf{write}\langle c\rangle}{\vdash \mathrm{t}_1 : \mathrm{in}[a], \mathrm{out}[c]}$$

$$(\mathrm{F}_2)\qquad \frac{\textbf{different}\langle a, 0, \text{“}true\text{”}\rangle \qquad a, 1, 1 \vdash \mathrm{t}_3 : c}{a \vdash \mathrm{t}_2 : a, c}$$

$$(\mathrm{F}_3)\qquad 0 \vdash \mathrm{t}_2 : 1, 1$$

$$(\mathrm{F}_4)\qquad \frac{\textbf{different}\langle c, a, \text{“}true\text{”}\rangle \quad \textbf{add}\langle c, 1, c_1\rangle \quad \textbf{times}\langle b, c_1, b_1\rangle \quad a, b_1, c_1 \vdash \mathrm{t}_3 : d}{a, b, c \vdash \mathrm{t}_3 : d}$$

$$(\mathrm{F}_5)\qquad a, b, a \vdash \mathrm{t}_3 : b$$

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:

$$(\mathrm{F}'_1)\qquad \frac{\textbf{read}\langle a, \mathrm{int}\rangle \qquad a \vdash \mathrm{t}_2 : b \qquad \textbf{write}\langle b\rangle}{\vdash \mathrm{t}_1}$$

$$(\mathrm{F}'_2)\qquad 0 \vdash \mathrm{t}_2 : 1$$

$$(\mathrm{F}'_3)\qquad \frac{\textbf{different}\langle a, 0, \text{“}true\text{”}\rangle \qquad \textbf{minus}\langle a, 1, a_1\rangle \qquad a_1 \vdash \mathrm{t}_2 : b_1 \qquad \textbf{times}\langle a, b_1, b\rangle}{a \vdash \mathrm{t}_2 : b}$$

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.

6. CONCLUSION AND RELATED WORKS

This tactic to define a partial evaluation for inference rules seems to apply correctly to TYPOL 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 independence 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 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 group, some work is already being done in that direction. Also, although the tactic itself is written in TYPOL, we don't have big hopes about applying it to itself, mainly because the tactic uses a lot of exotic Prolog predicates and side effects. Implementation of the tactic should be refined.

From the point of view of prolog-like languages, the results look pretty because 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.

From the TYPOL point of view, the most interesting point is that the tactic used is independent from any execution strategy chosen for TYPOL, thus giving a general method for partial evaluation of proof trees and inference rules.

As a comparison with the work of N.D.Jones'group [15)16)], 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 [11)], 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.

In the field of partial evaluation of Prolog, our method may be related to abstract interpretation of Prolog programs [3)]. The paper [12)]presents a method of partial evaluation of FCP programs that uses abstract interpretation, and which is close to the tactic presented here. The papers [9)8)]present an interesting way to partially evaluate directly Prolog programs. The strategies used share many features with our tactic. In [8)], the method used to prevent infinite unfolding is related with "wait" declarations of MU-prolog, where the user says in what syntactic conditions a call may or may not be unfolded. In [9)], the method is closer to ours, with an automatic system checking if a call unfolding generates an identical call later. In both papers, the method gives one specialized Prolog rule for each possible behavior, and this may lead to combinatorial explosion. We think that our method avoids this explosion by keeping when necessary the structure of the original program. Our tactic ($\mathbf{TC_3}$) builds one sub-skeleton per rule, instead of one skeleton per rule. Unfold-Fold transformations [26)]are a more powerful tool than oour system, since we have

no equivalent tool for folding. Its proof is also based on proof trees, independent of the evaluation heuristics. However we think we can avoid to use folding, since our tactic limits the number of generated inference rules.

REFERENCES

1) **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).

2) **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).

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

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

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

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

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

8) **Fujita, H., Furukawa, K.**, *" A Self-Applicable Partial Evaluator and its Use in Incremental Compilation"*, 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).

9) **Furukawa, K., Takeuchi, A.**, *"Partial Evaluation of Prolog Programs and its Application to Meta Programming"*, Proceedings IFIP 10$^{\text{th}}$ World Computer Congress, Dublin, Ireland, pp 415–420 (1986).

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

11) **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).

12) **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).

13) **Gentzen, G.**, *"The Collected Papers of Gerhard Gentzen"*, E. Szabo, North-Holland, Amsterdam, (1969).

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

15) **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).

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

17) **Kahn, G.** *"Natural Semantics"*, Proceedings of STACS 1987, Lecture Notes in Computer Science, Vol 247, Springer-Verlag (March 1987).

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

19) **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).

20) **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).

21) **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).

22) **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).

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

24) **Plotkin, G.D.**, *"A structural approach to operational semantics"*, Technical Report DAIMI-FN-19, University of Aarhus, (September 1981).

25) **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).

26) **Tamaki, H., Sato, T.**, *"Unfold-Fold Transformation of Logic Programs"*, 2$^{\text{nd}}$ International Logic Programming Conference, Uppsala, pp 127–137 (1984).

27) **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).

28) **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).