Formal specification and prototyping of a program specializer

Sandrine Blazy, Philippe Facon

Formal Specification and Prototyping of a Program Specializer

Sandrine Blazy, Philippe Facon
CEDRIC IIE
18 allée Jean Rostánd
91025 Evry Cedex, France
{blazy, facon}@iie.cnam.fr

Abstract. This paper reports on the use of formal specifications in the development of a software maintenance tool for specializing imperative programs, which have become very complex due to extensive modifications. The tool is specified in terms of inference rules and operates by induction on the abstract syntax. The correctness of these rules is proved using rule induction. A Prolog prototype has been derived for Fortran programs, using the Centaur programming environment.

Keywords: structured operational semantics, VDM, software maintenance, program specialization, proof of correctness, rule induction, Centaur.

1 Introduction

We have developed an original technique for specializing programs which are too difficult to maintain because they are too general. These programs are written in an imperative language (noted L in the sequel of this paper). This technique aims at understanding old programs, which have become very complex due to extensive modifications. From a given program and some form of restriction of its usage (e.g. the knowledge of some specific values of its input variables), this technique provides a simplified program, which behaves like the initial one when used according to the restriction. This approach is particularly well adapted to programs which have evolved as their application domains increase continually.

Our technique is a variant of partial evaluation, a well known technique that has been used for optimization and to derive compilers from interpreters [10]. Partial evaluation of a subject program P with respect to input variables $x_1, ..., x_m, y_1, ..., y_n$ for the values $x_1 = c_1, ..., x_m = c_m$ gives a residual program P', whose input variables are $y_1, ..., y_n$ and the executions of $P(c_1, ..., c_m, y_1, ..., y_n)$ and $P'(y_1, ..., y_n)$ produce the same results. Such a program is obtained by replacing variables by their constant values, by propagating constant values, and by modifying statements, for instance replacing each alternative whose condition simplifies to a constant value by the corresponding branch or unfolding loops when possible. But, our aim differs from the one of traditional partial evaluation. We do not try to optimize code but to improve its readability, mainly by simplifying it. For example, we never expand loops.

In another paper we have explained the aim and use of our tool [3], but not it's development. Here we will instead focus on the formal concepts we have used for developing it. Residual programs are used during maintenance. They are either visualized to locate anomalies while debugging or used as independent programs, instead of the initial ones. Thus, our tool - as software maintenance tool - must introduce absolutely no unforeseen changes in programs. Therefore, we have first to formally specify the specializer, then to prove the correctness of that specification with respect to the standard semantics, and eventually (third step) to prove the correctness of the implemented tool with respect to the specialization rules. We will develop here only the first two steps of the process. Indeed, from that specification, a Prolog prototype has been almost systematically derived by using the Centaur programming environment [4]. An industrial tool is being developed from that specification.

Our specification is expressed by inference rules operating on the abstract syntax of the language L. More precisely, we have used the natural semantics formalism [9], augmented with some VDM [8] operators. Natural semantics has its origin in the work of G.Plotkin ([7], [11]). Under the name “structured operational semantics”, he gives inference rules as a direct formalization of an intuitive operational semantics: his rules define inductively the transitions of an abstract interpreter. Natural semantics extends that work by applying the same idea (use of a formal system) to different kinds of semantic analysis (not only interpretation, but also typing, translation, etc.)

This paper is organized as follows. First, we detail in section 2 some inference rules that formally specify our specializer. Next, section 3 presents proofs of correctness of our specializer rules with respect to the dynamic semantics of the imperative language L. Section 4 explains how we have prototyped our specializer and gives some quantitative results about the implementation of a prototype for specializing Fortran programs. Section 5 presents conclusions and future work.

2 Inference Rules for Specialization

2.1 The Specialization Strategy

As explained in the introduction, we want to specialize a program for readability purposes, not for optimization ones, that is we want only to simplify it. What does it mean to simplify a program in that context? We believe that to remove useless code is always beneficial to program understanding. In that case the objective is compatible with that of program optimization (dead code elimination [2]), but this is certainly not the case in general. On the other hand, the replacement of (occurences of) variables by their values is not so obvious. The benefit depends on what these variables mean for the user: variables like PI, TAX_RATE, etc. are likely to be kept in the code; on the contrary, intermediate variables used only to decompose some computations may be not so meaningful for the user, and he may prefer to have them removed.

Replacing variables by their values may lead to dead code (by making the assignments to these variables useless) and thus gives more opportunities to remove code. However, this is certainly not a sufficient reason to do systematic replacement. Of course, even when there is no replacement, the known value of a variable is kept in the environment of our simplification rules, as it can give opportunities to remove useless code, for instance if the condition of an alternative may be evaluated thanks to that knowledge (and thus a branch may be removed).

The benefit of replacement depends not only on the kind of variable but also on the kind of user: a user who knows the application program well may prefer to keep the variables the meaning of which is already known to him; a user trying to understand an application program he does not know at all may prefer to see as few variables as possible. In fact, our experiments have shown that the system must be very flexible in that respect. Thus, our system works as follows. There are three options: no replacement, systematic replacement, and each replacement depending on the user.

To specify the partial evaluation, we use inference rules operating on the L abstract syntax. The following part of this section first presents rules defining both the constant propagation process and the simplification process. Then, a third part details the rules for partial evaluation of statements. These new rules combine the propagation rules and the simplification rules. Note that these techniques are not new, but we specify and use them in a novel way. In this paper, we present only the rules for assignment and alternative statements. The other rules can be found in [3].

2.2 Propagation and Simplification Rules

In the following, we use sequents such as $H \overset{propag}{\vdash} I{:}H'$ (propagation), $H \overset{simpl}{\vdash} I \longrightarrow I'$ (simplification), and the combination of both $H \overset{PE}{\vdash} I \longmapsto I', H'$ (propagation and simplification). In these sequents:

- H is the environment associating values to variables whose values are known before executing I. It is modelled by a VDM-like map [8], shown as a collection of pairs contained in set braces such as {variable → constant, ...}, where no two pairs have the same first elements. Our system initializes such maps by the list of variables and their initial values, supplied by the user.
- I is a statement (expressed in a linear form of the abstract syntax of L).
- I' is the simplified statement under the hypothesis H.
- H' is H which has been modified by the execution of I.
- The superscript of the turnstile such as ***propag***, ***simpl*** or ***PE*** denotes the set of rules the sequent belongs to.

In the sequents exhibited in this paper, we use the map operators *dom*, $\cup$, $\cap$, =, † and ⩤.

- The domain operator *dom* yields the set of the first elements of the pairs in the map.
- The union operator $\cup$ yields the union of maps whose domains are disjoint (in VDM, this operator is undefined if the domains overlap).
- The intersection operator $\cap$ of two maps yields the pairs common to both maps.
- The equality operator = of two maps yields *true* if and only if each pair of one map is a pair of the other map (and reciprocally).
- The map override operator † whose operands are two maps, yields a map which contains all of the pairs from the second map and those pairs of the first map whose first elements are not in the domain of the second map.
- When applied to a set and a map, the map deletion operator ⩤ yields those pairs in

the map whose first elements are not in the set.

The examples of Figure 1 illustrate these definitions of map operators.

$m = \{X \rightarrow 5, B \rightarrow true\}$	$dom(m) = \{X,B\}$
	$m \cup \{Y \rightarrow 7\} = \{Y \rightarrow 7, X \rightarrow 5, B \rightarrow true\}$
	$\{B\} ⩤ m = \{X \rightarrow 5\}$
	$m \cap \{X \rightarrow 5, B \rightarrow false\} = \{X \rightarrow 5\}$
$n = \{C \rightarrow false, X \rightarrow 8\}$	$m \dagger n = \{X \rightarrow 8, B \rightarrow true, C \rightarrow false\}$
	$n \dagger m = \{X \rightarrow 5, B \rightarrow true, C \rightarrow false\}$

Fig. 1. Some map operators

We have written some inference rules to explain how sequents are obtained from other sequents. Propagation rules are a special case of program verification rules: they perform only forward analysis and they propagate only equalities between variables and constants. Figure 2 presents six of the eight simplification and propagation rules for alternatives. If the condition C of an alternative evaluates to true, then:

- the environment H' resulting from the propagation of H through the alternative is obtained by propagating H through the statements I_1 of the then-branch (first rule: propagation),
- the simplification of the alternative is the simplification of its then-branch (second rule: simplification).

In the same way, there are two rules for an alternative whose condition evaluates to false (in these rules "true" becomes "false", "I_1" becomes "I_2", and "then" becomes "else"). Since these rules are very similar to the first two rules, they do not appear in Figure 5. They are shown with partial evaluation rules in Figure 4.

(1)
$$\frac{H \overset{eval}{\vdash} C \longrightarrow bool(\text{true}) \qquad H \overset{propag}{\vdash} I_1 : H'}{H \overset{propag}{\vdash} \text{if C then } I_1 \text{ else } I_2 \text{ fi: } H'}$$

(2)
$$\frac{H \overset{eval}{\vdash} C \longrightarrow bool(\text{true})}{H \overset{simpl}{\vdash} \text{if C then } I_1 \text{ else } I_2 \text{ fi} \longrightarrow I'_1}$$

Fig. 2 (begin) Propagation and simplification rules for alternatives

If the condition C of an alternative is only partially evaluated to C', the propagation and the simplification proceed along both branches of the alternative:

- the propagation of H through the then-branch I_j leads to an environment H_j, j=1,2.

The intersection of both environments is the final environment: if a variable has the same value in both environments H_1 and H_2, that value is kept in the final environment, otherwise it is removed from the final environment (third rule: propagation).

the simplification of the alternative yields the alternative whose condition is the partially evaluated condition C' and whose branches are the simplified branches of the initial alternative (fourth rule: simplification).

$$(3)\quad \frac{H \overset{eval}{\vdash} C \longrightarrow C' \qquad C' \neq bool(\mathrm{B}) \qquad H \overset{propag}{\vdash} I_1 : H_1 \qquad H \overset{propag}{\vdash} I_2 : H_2}{H \overset{propag}{\vdash} \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi: } H_1 \cap H_2}$$

$$(4)\quad \frac{H \overset{eval}{\vdash} C \longrightarrow C' \qquad C' \neq bool(\mathrm{B}) \qquad H \overset{simpl}{\vdash} I_1 \longrightarrow I'_1 \qquad H \overset{simpl}{\vdash} I_2 \longrightarrow I'_2}{H \overset{simpl}{\vdash} \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi} \longrightarrow \text{if } C' \text{ then } I'1 \text{ else } I'_2 \text{ fi}}$$

Fig. 2 (cont'd) Propagation and simplification rules for alternatives

The fifth rule of Figure 2 is a propagation rule. It shows that information can sometimes be derived from the equality tests that control alternatives. If the condition of an alternative is expressed as an equality such as X=E, where X is a variable that does not belong to the domain of the environment H and E evaluates to a constant N, then the pair (X, N) is added to the environment related to the then-branch.

Since the statements *ifX≠E then I_1 else I_2fi* and *ifX=E then I_2 else I_1fi* are semantically equivalent, there is a corresponding rule (sixth rule) for a condition of an alternative expressed as an inequality such as X≠E: in that case, the pair (X,N) is added to the environment related to the else-branch. Rules 5 and 6 express that only equalities between variables and constants can be added to the environment. Thus, if other information is expressed in the condition, it is not taken into account by the partial evaluator.

Rules 5 and 6 have been generalized to conditions of alternatives expressed as conjunctions of equalities and disjunctions of inequalities (rules 5' and 6'). In these rules, we have used generalized AND (denoted $\underset{i=1,n}{\wedge}$) and OR (denoted $\underset{i=1,n}{\vee}$).

$$(5) \quad \frac{H \overset{eval}{\vdash} E \longrightarrow number(N) \quad X \notin dom(H) \quad H \cup \{X \to N\} \overset{propag}{\vdash} I_1 : H_1 \quad H \overset{propag}{\vdash} I_2 : H_2}{H \overset{propag}{\vdash} \text{if} \quad (X = E) \quad \text{then } I_1 \text{ else } I_2 \text{ fi: } H_1 \cap H_2}$$

$$(6) \quad \frac{H \overset{eval}{\vdash} E \longrightarrow number(N) \quad X \notin dom(H) \quad H \overset{propag}{\vdash} I_1 : H_1 \quad H \cup \{X \to N\} \overset{propag}{\vdash} I_2 : H_2}{H \overset{propag}{\vdash} \text{if} \quad (X \neq E) \quad \text{then } I_1 \text{ else } I_2 \text{ fi: } H_1 \cap H_2}$$

$$(5') \quad \frac{\begin{array}{l} \forall\, i, H \overset{eval}{\vdash} E_i \longrightarrow number(N_i) \\ \forall\, i, X_i \notin dom(H) \quad H \bigcup_{i=1,n} \{Xi \to Ni\} \overset{propag}{\vdash} I_1 : H_1 \quad H \overset{propag}{\vdash} I_2 : H_2 \end{array}}{H \overset{propag}{\vdash} \text{if} \quad (\bigwedge_{i=1,n} Xi = Ei) \text{ then } I_1 \text{ else } I_2 \text{ fi: } H_1 \cap H_2}$$

$$(6') \quad \frac{\begin{array}{c} \forall\, i, H \overset{eval}{\vdash} E_i \longrightarrow number(N_i) \\ \forall\, i, X_i \notin dom(H) \quad H \overset{propag}{\vdash} I_1 : H_1 \quad H \bigcup_{i=1,n} \{Xi \to Ni\} \overset{propag}{\vdash} I_2 : H_2 \end{array}}{H \overset{propag}{\vdash} \text{if} \ (\bigvee_{i=1,n} Xi \neq Ei) \text{ then } I_1 \text{ else } I_2 \text{ fi: } H_1 \cap H_2}$$

Fig. 2-(end) Propagation and simplification rules for alternatives

Since the simplification is performed in the context of the propagation, and the propagation uses the simplification of expressions, we have chosen to combine propagation and simplification in our rules.

2.3 Combined Rules

For every FORTRAN statement, we have written rules that describe the combination of the propagation and simplification systems. This combination $\relbar\!\!\blacktriangleright$ of these two systems is defined by:

$$H \overset{PE}{\vdash} I \relbar\!\!\blacktriangleright I', H' \quad \text{iff} \quad H \overset{propag}{\vdash} I{:}H' \quad \text{and} \quad H \overset{simpl}{\vdash} I \longrightarrow I'$$

From this rule, we may define inductively the $\relbar\!\!\blacktriangleright$relation. For instance, Figure 3 specifies the rules for partial evaluation of assignments. The *eval* notation refers to the formal system of rules which simplifies the expressions.

If the expression E evaluates to a numerical constant N, the environment H is modified: the value of X is N whether X had already a value in H or not. With the kind of propagation we perform, the assignment X := E can be removed only if all possible uses of that occurrence of X do not use another value of X. For instance, in the sequence

$$X := 2;\ \text{if CODE} \neq 5 \text{ then } X := X{+}1 \text{ fi};\ Y := X,$$

the value 2 of X is propagated in the expression X+1 but the assignment X:=2 can not be removed because in the assignment Y:=X, X comes either from X:=2 (value 2) or from X:=X+1 (value 3). Thus, that sequence is only simplified to

$$X := 2;\ \text{if CODE} \neq 5 \text{ then } X := 3 \text{ fi};\ Y := X.$$

To eliminate assignments that become useless after the partial evaluation, we use classical dead code elimination algorithms [2]. Thus, elimination of redundant assignments is performed in a separate optimization phase.

If E is only partially evaluable into E', the expression E is modified as part of the assignment X:= E and the variable X is removed from the environment if it was in it, because its value has become unknown.

$$\frac{H \overset{eval}{\vdash} E \longrightarrow \mathit{number}(N)}{H \vdash X := E \longrightarrow X := N,\ H \dagger \{X \rightarrow N\}}$$

$$\frac{H \overset{eval}{\vdash} E \longrightarrow E' \qquad E' \neq \mathit{number}(N)}{H \vdash X := E \longrightarrow X := E',\ \{X\} \lhd H}$$

Fig. 3. Partial evaluation of assignments

The following examples illustrate these two cases. In *Ex.1*, as the value of the variable A is known, the new value of the assigned variable C is introduced in the environment. We suppose that the assignment C := A+1 can be removed from the reduced program. In *Ex.2*, after the partial evaluation of the expression A+B, the value of C has become unknown. Such a case only happens when A and B do not have both constant values.

$$\textit{Ex.1} \quad \{A \rightarrow 1, C \rightarrow 4\} \vdash C := A + 1 \longrightarrow \text{skip}, \{A \rightarrow 1, C \rightarrow 2\}$$

$$\textit{Ex.2} \quad \{A \rightarrow 1, C \rightarrow 2\} \vdash C := A + B \longrightarrow C := 1 + B, \{A \rightarrow 1\}$$

The rules for partial evaluation of alternatives are defined in Figure 4. If the condition C of an alternative evaluates to a logical constant, this alternative can be simplified to the corresponding simplified branch. If C is only partially evaluated to C', the partial evaluation proceeds along both branches of the alternative and the final environment is the intersection of the two environments resulting from the simplification of both branches (as explained previously, Fig. 2, rule (4)).

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{false}) \qquad H \vdash I_2 \longrightarrow I'_2, H'}{H \vdash \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi} \longrightarrow I'_2, H'}$$

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{true}) \qquad H \vdash I_1 \longrightarrow I'_1, H'}{H \vdash \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi} \longrightarrow I'_1, H'}$$

$$\frac{H \overset{eval}{\vdash} C \longrightarrow C' \qquad C' \neq \mathit{bool}(B) \qquad H \vdash I_1 \longrightarrow I'_1, H'_1 \qquad H \vdash I_2 \longrightarrow I'_2, H'_2}{H \vdash \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi} \longrightarrow \text{if } C' \text{ then } I'_1 \text{ else } I'_2 \text{ fi}, H'_1 \cap H'_2}$$

Fig. 4. Partial evaluation of alternatives

3 Correctness of the Partial Evaluation

Our aim in this section is to show how to prove that the specialization presented above is correct, with respect to the dynamic semantics of L, given in the natural semantics formalism.

We will show that this is expressed by two inference rules, one expressing soundness (each result of the residual program is correct with respect to the initial program) and one expressing "R-completeness" (each correct result is computed by the residual program too). We use the term "R-completeness" (result-completeness) to avoid confusion with the completely different notion of specialization completeness (i.e. no further specialization could be done, which is not an issue here). As both programs are deterministic, we could have only one rule using equality, but the demonstration of our two rules is not more complicated and is more general (being also applicable for non-deterministic programs). Examples of proofs for the assignment and alternative statements are detailed in this section.

3.1 Rules Proving Soundness and R-completeness

To prove the simplification, we need a formal dynamic semantics of L and we must prove the soundness and R-completeness of the simplification rules with respect to that dynamic semantics. To express this dynamic semantics, we use the same formalism (natural semantics [9]) as for simplification. Thus, the semantic rules we give have to generate theorems of the form $H \overset{sem}{\vdash} I\colon H'$, meaning that in environment H, the execution of statement I leads to the environment H' (or the evaluation of expression I gives value H'). These rules are themselves not proved: they are supposed to define ex nihilo the semantics of L, as G.Plotkin [11] and G.Kahn [9] did for languages like ML.

To prove these rules would mean to have another formal semantics (e.g. a denotational one) and prove that the rules are sound and complete with respect to it. But there is no such official semantics for any imperative language. Thus that proof would rather be a proof of consistency between two dynamic semantics we give. That is outside the scope of our work: we want to prove consistency between simplification and dynamic semantics, not between two dynamic semantics.

Now how can we prove that the specialization system is sound and R-complete with respect to the dynamic semantics system? Instead of the usual situation, that is a formal system and an intended model, we have two formal systems: the specialization system (noted *PE*) and the dynamic semantics system (noted *sem*). A program P is simplified to P' under hypothesis H_0 on some input variables if and only if $Ho \vdash P \rightarrow P'$ is a theorem of the specialization system.

Let us call H the environment containing the values of the remaining input variables. Thus, $H_0 \cup H$ is the environment containing the values of all input variables. With that initial environment, P' evaluates to H' if and only if $Ho \cup H \overset{sem}{\vdash} P' : H'$ is a theorem of the dynamic semantics (*sem*) system. In a similar way, P evaluates to H' if and only if

$Ho \cup H \overset{sem}{\vdash} P : H'$ is a theorem of the *sem* system

Now, soundness of specialization with respect to dynamic semantics means that each result computed by the residual program is computed by the initial program. That is, for each P, P', H_0, H, H': if P is simplified to P' under hypothesis H_0 and P' executes to H' under hypothesis $H_0 \cup H$, then P executes to H'under hypothesis $H_0 \cup H$. Thus soundness of simplification with respect to dynamic semantics is formally expressed by the first rule of Figure 5.

R-completeness of simplification with respect to dynamic semantics means that each result computed by the initial program P is computed by the residual program P'. Thus, it is expressed by the second inference rule of Figure 5. In fact, our approach to prove simplification is very close to the approach of [5] to prove the correctness of translators: in that paper, dynamic semantics and translation are both given by formal systems and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules (that are proved by induction on the length of the proof; here we will use rule induction instead).

Note that both rules are not the most restricting rules (for instance their initial environment is Ho ∪ H and not only H, to allow partial simplification).

$$\frac{Ho \vdash P \rightarrow P' \qquad Ho \cup H \overset{sem}{\vdash} P' : H'}{Ho \cup H \overset{sem}{\vdash} P : H'} \qquad \textit{(soundness)}$$

$$\frac{Ho \vdash P \rightarrow P' \qquad Ho \cup H \overset{sem}{\vdash} P : H'}{Ho \cup H \overset{sem}{\vdash} P' : H'} \qquad \textit{(R-completeness)}$$

Fig. 5. Correctness of the program simplification

To prove both rules of Figure 5 concerning programs, we prove that they hold for any statement we specialize (remember that we do not analyze data declarations). Thus, we have to prove that both rules of Figure 6 hold. In these rules, I denotes a statement and I' denotes the corresponding specialized statement.

$$\frac{Ho \overset{\mathit{PE}}{\vdash} I \longrightarrow I', H' \qquad Ho \cup H \overset{\mathit{sem}}{\vdash} I' : H''}{Ho \cup H \overset{\mathit{sem}}{\vdash} I : H''} \qquad \mathit{(soundness)}$$

$$\frac{Ho \overset{\mathit{PE}}{\vdash} I \longrightarrow I', H' \qquad Ho \cup H \overset{\mathit{sem}}{\vdash} I : H''}{Ho \cup H \overset{\mathit{sem}}{\vdash} I' : H''} \qquad \mathit{(R\text{-}completeness)}$$

Fig. 6. Correctness of the statements partial evaluation

The dynamic semantics of L has been formalized by the *sem* system. The dynamic semantics rules for assignments and alternatives are propagation rules, as shown in Figure 7. For that reason, in the *sem* system, we have overloaded the ":" symbol representing the system *propag*, instead of using a new symbol.

sem

$$\frac{H \overset{\mathit{eval}}{\vdash} E \longrightarrow \mathit{number}(N)}{H \vdash X := E : H\dagger\{X \rightarrow N\}}$$

$$\frac{H \overset{\mathit{eval}}{\vdash} C \longrightarrow \mathit{bool}(\text{true}) \qquad H \vdash I_1 : H'}{H \vdash \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi}: H'}$$

Fig. 7. Dynamic semantics rules for assignments and alternatives

To prove the validity of the R-completeness and soundness rules, we use rule induction on the partial evaluation, and on the dynamic semantics. Indeed, the *PE* and *sem* systems have been defined inductively.

Our inductive hypothesis for soundness is the following property Π_s, defined as follows: $\Pi_s\,(Ho,I,I',H') \Leftrightarrow (\forall\, Ho,I,I',H' \mid Ho \overset{\mathit{PE}}{\vdash} I \longrightarrow I', H' :$

$$(\forall\, H,H''\ \ Ho \cup H \overset{\mathit{sem}}{\vdash} I' : H'' \Rightarrow Ho \cup H \overset{\mathit{sem}}{\vdash} I : H'')).$$

The inductive hypothesis Π_c for R-completeness is defined in a similar way. The rule induction operating on a formal system (either *PE* or *sem*) states that quadruples are only obtained by the rules belonging to this formal system.

To construct our proof trees we use property *Prop.1*, which states that if some (variable, value) pairs are added to an environment Ho, what had been already proved in this environment Ho still holds in the new environment Ho ∪ H.

$$\textit{Prop.1}\ \boxed{\dfrac{Ho \overset{eval}{\vdash} C \longrightarrow V}{Ho \cup H \overset{eval}{\vdash} C \longrightarrow V}}$$

3.2 Examples of Proofs of Soundness

The following examples deal only with proofs of soundness. Proofs of R-completeness are similar. We start with treating simple statements, which are not composed of other statements. They form the basic cases of the proof. Figure 8 shows a proof of such a statement. The possible removal of assignment does not appear in the proof tree, since it is performed during a dead code elimination phase, subsequently to the evaluation of the expression of the assignment.

If we assume that the partial evaluation rule

$$\frac{H \overset{eval}{\vdash} E \longrightarrow \textit{number}\,(N)}{H \vdash X := E \longrightarrow X := N,\ H\dagger\{X \to N\}}$$

holds, and by the rule of the sem system about assignment:

$$\frac{H \overset{eval}{\vdash} E \longrightarrow \textit{number}\,(N)}{H \vdash X := E\ H\dagger\{X \to N\}}$$

then for any H_0 and H'' such that Ho ∪ H $\vdash$ X:=N : H'',

we have, H'' = (Ho ∪ H)†{X→N}, *thus proving that the following proof tree holds:*

$$\cfrac{\cfrac{H \overset{eval}{\vdash} E \longrightarrow \textit{number}\,(N)}{Ho \cup H \overset{eval}{\vdash} E \longrightarrow \textit{number}\,(N)}\ \textit{Prop.1}}{Ho \cup H \vdash X := E \longrightarrow X := N,\ H\dagger\{X \to N\}}\ \begin{array}{l}\textit{rule for the assignment}\\ \textit{belonging to the sem system}\end{array}$$

thus proving that Π_s *(H, X:=E, X:=N,* H†{X→N}*) holds.*

Fig. 8. Proof of soundness of an assignment partial evaluation

Once simple statements have been proved, we have to prove that the soundness rule holds for composite statements. Figure 10 shows a proof of soundness for an alternative whose condition evaluates to true. There is a similar proof for the case when the condition evaluates to false.

If we assume that the rule

$$\frac{Ho \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{true}) \qquad Ho \overset{PE}{\vdash} I_1 \longrightarrow I'_1, H'}{Ho \overset{PE}{\vdash} \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi} \longrightarrow I'_1, H'}$$

and the inductive hypothesis Π_s *(H,*I_1*,I'*$_1$*,H') hold, then we can deduce that:*

$$(\forall\, H_0, H'': \quad Ho \cup H \overset{sem}{\vdash} I'_1 : H'' \Rightarrow Ho \cup H \overset{sem}{\vdash} I_1 : H'').$$

Furthermore, the dynamic semantics of the if statement states that

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{true}) \qquad H \vdash I_1 : H'}{H \vdash \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi}: H'}$$

Then, we can state that:

$$(\forall\, Ho, H'': \quad Ho \cup H \overset{sem}{\vdash} I'_1 : H'' \Rightarrow \quad Ho \cup H \overset{sem}{\vdash} \text{if } C \text{ then } I_1 \text{ else } I_2 \text{ fi}: H'').$$

thus proving that Π_s *(H, if C then I1else I2 fi, I'*$_1$*,H') holds.*

Fig. 9. Proof of soundness of an alternative whose condition evaluates to true

4 Implementation of a Prototype

This section describes the overall architecture of our specializer. Then, it gives quantitative results measured for a Fortran specializer.

4.1 Architecture of the Specializer

The specialization rules are very close to the ones we have implemented in the Centaur/ L environment. The Centaur system [4] is a generic programming environment parametrized by the syntax and semantics of programming languages. When provided with the description of a particular programming language, including its syntax and semantics, Centaur produces a language specific environment. The resulting environment consists of a structured editor, an interpreter/debugger and other tools, together with an uniform graphical interface. Furthermore, in Centaur, program texts are represented by abstract syntax trees. The textual (or graphical) representation of abstract syntax trees nodes may be specified by pretty-printing rules. Centaur provides a default representation.

We have used such a resulting environment, Centaur/L, to build our specializer. From Centaur/L we have implemented an environment for specialization of programs written in L. Figure 10 shows the overall architecture of this environment, where Centaur/L is represented by the grey part.

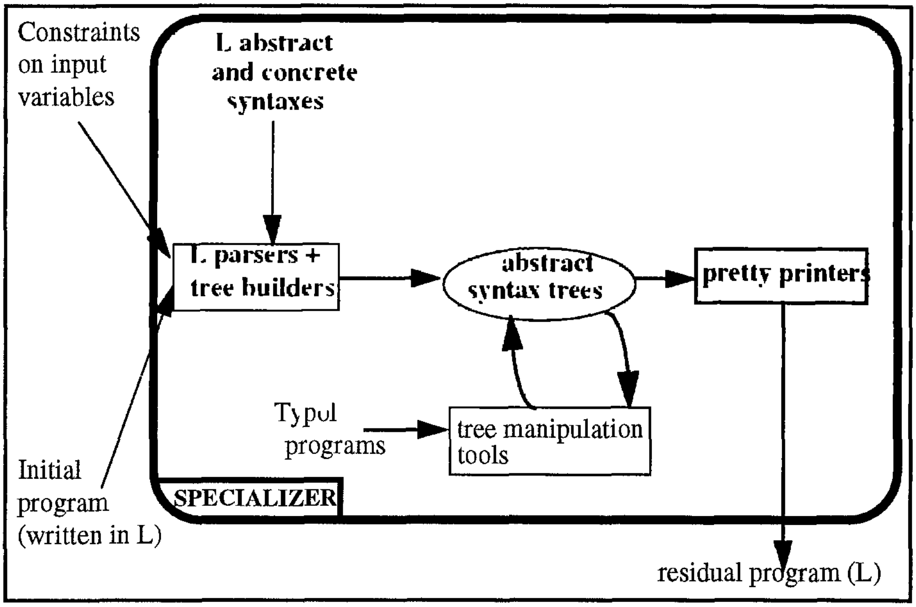
Constraints on input variables
L abstract and concrete syntaxes
L parsers + tree builders
abstract syntax trees
pretty printer
Typol programs
tree manipulation tools
SPECIALIZER
Initial program (written in L)
residual program (L)

Fig. 10. The Centaur/L environment

From a Centaur/L environment, we have written Typol programs to implement our specification rules. Typol is a language for specifying the semantic aspects of languages; it is included in Centaur, so that the system is not restricted to manipulations that are based solely on syntax. Typol is an implementation of natural semantics. It can be used to specify and implement static semantics, dynamic semantics and translations. Typol programs are compiled into Prolog code. When executing these programs, Prolog is used as the engine of the deductive system.

Figure 11 shows two examples of Typol rules: a rule specializing an assignment whose expression does not evaluate to a constant value (1) and a rule specializing an alternative in its else-branch (2). These rules show Prolog primitives implementing map operators (their identifiers are italicized), nodes of abstract syntax trees (they are written in bold), and calls to Typol programs (their printing stands out in relief).

```
eval (H |- E -> E') & nonvar (E') & deletion (H, X, H')
---------------------------------------------------------------------     (1)
Env |- ass (name X, E) -> ass (name X, E'), Env' ;

eval (H |- E -> logic_cst E') & false (E') & H |- I2 -> I2', H'
---------------------------------------------------------------------------     (2)
H |- struct_if (Tag, E, I1, I2) -> l_stat[I2'], H' ;
```

Fig. 11. Some Typol rules for assignments and alternatives

4.2 Quantitative Results for Fortran Programs

We have chosen Fortran to implement our technique for specializing imperative programs because Fortran is still widespread in scientific programming. We have exhibited a large class of scientific applications our approach is particularly well adapted to. We have written about 200 Typol rules to implement a Fortran specializer. 10 rules express how to reach abstract syntax nodes representing simplifiable statements. 90 rules perform the normalization of expressions. Among the 100 rules for simplification, 60 rules implement the simplification of expressions. The 40 other rules implement the statements simplification. We have written about 25 Prolog predicates to implement the VDM operators we have used to specify the simplification. Thus, these operators are used in Typol rules as in the formal specification of the simplification.

The partial evaluator may analyze any Fortran program, but it simplifies only a subset of Fortran 77. This subset is a recommended standard for developping the scientific applications we have studied. For instance, it does not analyze any goto statement (they are not recommended), but only goto statements that implement specific control structures (e.g. a while-loop).

The average initial length of programs or subroutines we have analyzed is 100 lines of FORTRAN code, which is lengthier than the recommended length (60-70 lines). The reduction rate amounts from 25% to 80% of lines of code. That length reduction is obtained mostly by removal of code which is useless in the given context. Thus, it implies a direct improvement in readability. This reduction is specially important when there is a large number of assignments and conditionals. This is the case for most subroutines implementing mathematical algorithms. For subroutines whose main purpose is editing results or calling other subroutines, the reduction is generally not so important.

5 Conclusion

We have used partial evaluation for programs which are difficult to maintain because they are too general. We have formally specified our specialization with inference rules expressed in the natural semantics formalism and augmented with some VDM operators. We have shown how to prove by rule induction the correctness of our formal system, given the standard semantics of the programming language.

A protoype has been derived from this specification. We are now focusing on an industrial implementation of this prototype. This tool will be used by maintainers at the EDF, the national French company that provides and distributes electricity to the whole country. It will be developed by CEDRIC IIE and Simulog, a company that provided us with some basic tools including Centaur/Fortran. To obtain an insdustrial tool from the current prototype, we will take into account new operators from Fortran 90. Most of this work will consist first in adding new language concepts, that is new abstract syntax operators, and seconly in defining for each new operator how the specialization rules are modified.

Furthermore, the formal specification of our specializer is used as a reference document between people involved in the development of the specializer. It allows us to:

- express the real semantics of each languge construct we simplify,
- define precisely what are the interesting simplifications of statements,
- exhibit and prove the specialization rules. With all extensions we take into account, the whole proof become rather tedious. Thus, we expect to use a theorem prover as COQ [6].

References

1. *FORTRAN.* ANSI standard X3.9, 1978.
2. A.Aho, R.Sethi, J.Ullman, *Compilers.* Addison-Wesley eds., 1986.
3. S.Blazy, P.Facon *SFAC, a tool for program comprehension by specialization* IEEE Workshop on program comprehension, Washington, November 1994.
4. *Centaur 1.1 documentation.* INRIA, January 1990.
5. J.Despeyroux, *Proof of translation in natural semantics.* Symposium on Logic in Computer Science, Cambridge USA, June 86.
6. G.Dowek et al. *The Coq proof assistant user's guide* INRIA report 134, December 1991.
7. M.Hennessy, *The semantics of programming languages.* Wiley eds., 1990.
8. C.B.Jones, *Systematic software development using VDM.* Prentice-Hall, 2nd eds., 1990.
9. G.Kahn, *Natural semantics.* Proceedings of STACS'87, Lecture Notes in Computer Science, vol.247, March 1987.
10. U.Meyer, *Techniques for evaluation of imperative languages.* ACM SIGSOFT, March 1991, pp.94-105.
11. G. Plotkin, *A structural approach to operational semantics.* Report DAIMI FN-19, University of Aarhus, 1981.