Partial evaluation and symbolic computation for the understanding of Fortran programs

Sandrine Blazy, Philippe Facon

Partial Evaluation and Symbolic Computation for the Understanding of Fortran Programs

Sandrine Blazy
DER EDF
1, avenue du Général de Gaulle
92141 Clamart Cedex, France
Sandrine.Blazy@der.edf.fr

Philippe Facon
CEDRIC IIE
18 allée Jean Rostand
91025 Evry Cedex, France
facon@cnam.cnam.fr

Abstract. We describe a technique and a tool supporting partial evaluation of Fortran programs, i.e. their specialization for specific values of their input variables. We aim at understanding old programs, which have become very complex due to numerous extensions. From a given Fortran program and these values of its input variables, the tool provides a simplified program, which behaves like the initial one for the specific values. This tool uses mainly constant propagation and simplification of alternatives to one of their branches. The tool is specified in inference rules and operates by induction on the Fortran abstract syntax. These rules are compiled into Prolog by the Centaur/Fortran environment.

1. Introduction

Program understanding is the most expensive phase of the software life cycle. It is said that 40% of the maintenance effort is spent trying to understand how existing software works [Sneed89]. All maintenance problems do not require complete program understanding, but each problem requires at least a limited understanding of how the source code works, and how it is related to the external functions of the application. There exists now a wide range of tools to support program understanding [VanZuylen92].

Program slicing is a technique for restricting the behaviour of a program to some specified subset of interest. The slice of a program P on variable X at location i is the set of statements that influence the value of X at i. This is an executable program that is obtained by data flow analysis. Program slicing can be used to help maintainers understand and debug foreign code [GallagherLyle91].

We have developed a complementary technique: reduction of programs for specific values of their input variables. It aims at understanding old programs, which have become very complex due to extensive modifications. From a given Fortran program and some form of restriction of its usage (e.g. the knowledge of some specific values of its input variables), the tool 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.

Partial evaluation is an optimization technique used in compilation to specialize a program for some of its input variables. 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 such that the executions of $P(c_1...c_m,y_1...y_n)$ and $P'(y_1...y_n)$ produce the same results [Meyer91]. Such a program is obtained by replacing variables by their constant values, by propagating constant values and simplifying statements, for instance replacing each alternative whose condition simplifies to a constant value (true or false) by the corresponding branch.

Partial evaluation has been applied to program optimization and compiler generation from interpreters (by partially evaluating the interpreter for a given program) [JonesSestoftSondergaard89]. In this context, previous works have especially dealt with functional [Ambriola&al.85] and logical languages [Sahlin91]. The structure of the program may be modified (using loop expansion, subroutines expansion and renaming [ErshovOstrovski87]) in order to optimize the residual code.

As far as imperative languages are concerned, partial evaluation has been used to software reuse improvement by restructuring software components to improve their efficiency [Coen-Porisini&al.91], [Andersen91]. Partial evaluation has been applied to numerical computation to provide performance improvements for a large class of numerical programs, by eliminating data abstractions and procedure calls [BerlinWeise90].

Our goal is different. We remove groups of statements that are never used in the given context, but we do not expand statements. This does not change the original structure of the code. We transform general-purpose programs into shorter and easier to understand special-purpose programs. This transformational approach aims at improving a given program without disturbing its correctness when used in a given restricted and stable context. However, unlike [Kasyanov91], we do not aim at improving a program according to a performance criterion (e.g. memory), but at improving the readability of programs.

This paper is organized as follows. First, we justify our interest in scientific applications written in Fortran in section 2. Next, we present in section 3 the two main tasks of our partial evaluator: constant propagation and simplification. In section 4, we describe our transformational semantics as a set of inference rules for partial evaluation, and we show how these rules combine constant propagation and simplification rules. Section 5 presents conclusions and future work.

2. Scientific programming

A number of scientific applications, written in Fortran for decades, are still vital in various domains (management of nuclear power plants, of telecommunication satellites, etc.). Even though, more recent languages are used to implement the most external parts of these applications. It is not unusual to spend several months to understand such applications before being able to maintain them. In a recent follow up about maintenance practices of scientific applications [Haziza&al.92], we have noticed that understanding a 120 000 Fortran lines application took nine months. So, providing the maintainer with a tool, which finds parts of lost code semantics, allows to reduce this compulsory period of adaptation.

2.1. Characteristics

One of the peculiarities of scientific applications is that the technological level of scientific knowledge (linear systems resolution, turbulence simulation, etc.) is higher than the knowledge usually necessary for data processing (memory allocation, data representations, etc.). The discrepancy is increased by the widespread use of Fortran, which is an old-fashioned language. Furthermore, for large scientific applications at EDF Fortran 77, which is quite an old version of the language, is exclusively used to guarantee the portability of the applications on different machines (mainframes, workstations, vectorial computers, etc.) [ANSI78].

2.2. General Purpose Applications

Our study has highlighted common characteristics in Fortran programming at EDF. These scientific applications have been developed a decade ago. During their evolution, they had to be reusable in new and various contexts. For example, the same thermohydraulic code implements both general design surveys for a nuclear power plant component (core, reactor, steam generator, etc.) and subsequent improvements in electricity production models. The result of this encapsulation of several models in a single large application domain increases the program complexity, and thus amplifies the lack of structures in Fortran programming language.

This generality is implemented by Fortran input variables whose value does not vary in the context of the given application. We distinguish two classes of such variables:

- *data about geometry*, which describe the modelled domain. They appear most frequently in assignment statements (equations that model the problem).
- data taking a finite number of values, which can be represented by logical variables. These are either *filters* necessary to switch the computation in terms of the context (modelled domain), or *tags* allowing to minimize errors risks about the precision of the output values.

Figure 1 shows an example of program reduction. The code section of figure 1-a is extracted from one of the applications we have studied [Nicolas&al.89]. The partial evaluation of this code section according to the simplification criteria of figure 1-b yields the code section of figure 1-c. A maintenance team is used to update a specific version of the application. These people know some filters properties ($IC = 0$ and $IREX = 1$) as well as data about geometry ($DXLU = 0, 5$). Furthermore IM is a tag whose value is 20.

The knowledge of these values of input variables simplifies the code (as shown in figure 1-c). Because of the truth of the relation $IREX = 1$, two alternatives are simplified (1). The first alternative is simplified to its then-branch. In this branch, the variable DXLU is replaced by its value, which modifies the variables X(I) and DX(I) (2). The variable IM is replaced by its value too (3). The condition of the next alternative is simplified (4) since the value of IC is known. Furthermore, the relation $IC = 0$ simplifies the following alternative to its then-branch (5), which allows to compute the value of the variables ZERO, IREGU and IDECRI (6). Because the values of these three variables are constant values, the three corresponding assignments are removed from the code. Then other alternatives are simplified (7).

```
(1)    IF ( IREX .NE. 0 ) THEN
(3)        DO 111 , I = 1 , IM
(2)            X(I) = XMIN + FLOAT(I-1) * DXLU
       111 CONTINUE
(3)        DO 112 , I = 1 , IM
(2)            DX(I) = DXLU * I
       112 CONTINUE
         ELSE
           READ (NFIC11,*, ERR=1103) X
           DO 121 , I = 1 , IM
               DX(I) = X(I+1) - X(I)
       121 CONTINUE
        ENDIF
(4)     IF ( IMATSO .EQ. 0 .AND. IC .GE. 0) THEN
(6)         ZERO = 0.
(5)         IF ( IC .EQ. 0 ) THEN
(6)             IREGU = 1
            ELSE IF (IC .GE. IM) THEN
                IREGU = 0
            ENDIF
(1)         IF ( IREX .EQ.2 ) THEN
                READ (NFIC11,'(A)',ERR=5,END=5) L
                IF ( INDEX (L,'I') .NE. 0 ) THEN
                  IDECRI = 1
                ENDIF
            ELSE
(6)             IDECRI = 2
            ENDIF
(7)         IF ( IDECRI .EQ. 1 ) THEN
                IF ( IREGU .EQ. 0 ) THEN
                  IMIN = 2
                  IMAX = IM
                ELSE
                  IMIN = IM
                  IMAX = IM
                ENDIF
(7)         ELSE IF ( IDECRI .EQ. 2 ) THEN
(7)             IF ( IREGU .EQ. 0 ) THEN
                  JMIN = 2
                  JMAX = JM
                ELSE
                  JMIN = JM
                  JMAX = JM
                ENDIF
             ENDIF
        ENDIF
```

Fig. 1-a. Initial code

```
IREX = 1
IC = 0
IM = 20
DXLU = 0,5
```

Fig. 1-b. Constraints on input variables

```
   DO 111 , I = 1 , 20
     X(I) = XMIN + 0,5*FLOAT(I-1)
 111 CONTINUE
   DO 112 , I = 1 , 20
    DX(I) = 0,5*I
 112 CONTINUE
   IF ( IMATSO .EQ. 0 ) THEN
    JMIN = JM
    JMAX = JM
   ENDIF
  END
```

Fig. 1-c. Simplified code

Fig. 1. An example of program simplification

This reduction is specially important thanks to the high 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.

The links between the initial code and the simplified one are not shown in figure 1. But, if the user wishes to see how the simplified code has been obtained from the initial one, he can visualize both codes in a same window in which the simplified code is written in a different colour.

3. Two Aspects of Partial Evaluation Applied to Imperative Programs

Our partial evaluator performs two main tasks: constant propagation through the code and simplification of statements. The tool can execute one of the two independently of the other. For instance, if the user is a physicist who is familiar with the equations implemented in the code, he wishes to locate in Fortran statements these equations and their variables, as they appear in the formulae of these equations. But if the user is a maintainer who does not know the application well, he would rather visualize the code as simplified as possible. In most cases however and for an optimal partial evaluation, the tool performs both tasks.

3.1. Constant Propagation

Constant propagation is a well-known global flow analysis technique used by compilers. It aims at discovering values that are constant on all possible executions of a program and to propagate forward through the program these constant values as far as possible. Some algorithms now exist to perform fast and powerful constant propagation [WegmanZadeck91].

We describe in this section our constant propagation process. It modifies most expressions by replacing some variable occurrences by their values and by normalising all expressions through symbolic computation. Presently, our tool propagates only *equalities between variables and constants*. Of course, that limits the precision of the analysis.

Substitution. Before running the partial evaluator, the user specifies numerical values for some input variables of the program (thanks to his personal knowledge of the application). Constant propagation spreads this initial knowledge supplied by the user. In a first stage, the partial evaluator replaces each specified variable by its value. Then, expressions whose operands are all constant values are computed and these resulting values are propagated forward through the whole program. This technique allows to remove from the code all occurrences of variables identifiers that are not meaningful. The substituted values may be visualized differently from other values (with a different colour).

Normalisation. For any numerical expression, we have to recognize if it reduces to a constant value (e.g. x+3-x reduces to 3). In the same way, for any logical expression, we have to recognize if it reduces to a conjunction of equalities or to a disjunction of inequalities. In the first case (respectively the second case), we will be able to propagate equalities in the then- (respectively else-) branch of alternatives. To do this, we perform constant propagation. To propagate constant values as most as possible, our system normalises each expression into a canonical form: a polynomial form for numerical expressions and a conjunctive normal form for the logical expressions.

In a polynomial form, expressions are simplified by computing the values of the coefficients of the polynomial. Polynomial forms are written according to the decreasing powers order. When some terms of a polynomial have the same degree (e.g. z^2, x^2 and t.u), they are sorted according to a lexicographical order (e.g. $t.u < x^2 < z^2$). The canonical form of a relational expression is obtained from the canonical forms of its two numerical subexpressions. In normalized relational expressions, all variables and values occur on only one side of the operator.

Because of these modifications of expressions, overflow, underflow or round-off errors may happen. Therefore, the normalization of expressions may cause run-time errors. It is possible to obtain programs that will cause machine errors when compiled and executed. Conversely, some run-time errors may vanish thanks to the partial evaluation. As most partial evaluation systems [KemmererEckmann85], our tool ignores such problems. In this case only, the tool does not yield a program which behaves like the initial one.

3.2. Simplification

Simplification is an option of the partial evaluator. First, the expressions are simplified by the propagation as explained above. Then, the simplification process reduces the size of the code by removing both assignments of variables which can be evaluated to constants and statements which are never used for the specified values. This simplification includes the elimination of redundant tests and in particular the simplification of alternatives to one case thanks to the evaluation of their conditions. To simplify a statement means to remove or to modify it. Its components must be simplified, but in different contexts. This section defines the simplification for each statement.

A *write* statement is simplified by simplifying its parameters that are expressions. A *read* statement is simplified by removing its parameters whose values are known input values. If all its parameters have known values, the read statement is removed (replaced by an empty statement). Since the removed parameters do not appear in the residual program any more, their initialization is not missing in the code that is therefore still executable.

An *assignment* simplification consists in removing it, if its expression has been evaluated to a constant.

An *alternative* is simplified into one of its branches when its condition has been evaluated to either true or false. Otherwise, the statements of the two branches are simplified. In this case a branch may become the empty statement.

Loops that are never entered are removed. When discovered, infinite loops are left unchanged. Otherwise, if some loop invariant holds, the only statements of the loop which are simplified by the knowledge of variables values, are those statements whose all variables belong to the invariant. Thus, we do not expand loops because we want to keep the original structure of the code. Furthermore, Fortran loops are implemented using labels and goto statements. So, when a loop is removed, its label statement is kept since other statements may contain goto statements to such labels (the label is left unchanged and the statement is replaced by a skip statement).

A *call* statement simplification consists in replacing its actual parameters whose values are known by these values. The identifier of the called subroutine is left unchanged in the current program (subroutines are not specialized) and the user has to run the partial evaluator on this subroutine code if he wants to simplify it too.

The partial evaluation does not simplify other statements. Let us notice that our tool yet can neither deal with inequalities nor with literal values.

4. Inference Rules for Partial Evaluation

To specify the partial evaluation, we use inference rules operating on the Fortran abstract syntax and expressed in the natural semantics formalism [Kahn87], augmented by some VDM [Jones90] operators. This section first presents rules defining on the one hand the constant propagation process and on the other hand the simplification process. Then, it details the rules for partial evaluation of statements. These new rules combine the propagation rules and the simplification rules. Let us notice that the techniques we implement are not new, but we specify and use them in a novel way.

4.1. Propagation and Simplification Rules

In the following, we use sequents such as $H \vdash I{:}H'$ (propagation), $H \vdash I \longrightarrow I'$ (simplification), and the combination of both $H \vdash I \relbar\joinrel\blacktriangleright I', H'$ (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 [Jones90], 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 Fortran statement.
- I' is the simplified statement under the hypothesis H.
- H' is H which has been modified by the execution of I.

The sequents $H \vdash I{:}H'$ express the propagation relation. The sequents $H \vdash I \longrightarrow I'$ express the simplification under hypotheses. It depends of the components of I, which are themselves simplified under other hypotheses. Thus, the definition of the simplification relation uses the definition of the propagation relation.

In the sequents, we use the map operators *dom*, $\cup$, $\dagger$, $\triangleleft$ and $⩤$.

- The domain operator *dom* is 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 (this operator is undefined if the domains overlap).
- The map override operator $\dagger$ 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 operand.

- The map restriction operator $\lhd$ is defined with a first operand which is a set and a second operand which is a map; the result is all of those pairs in the map whose first elements are in the set.
- 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 example of figure 2 illustrates these definitions.

$$m = \{X \rightarrow 5, B \rightarrow true\} \qquad dom(m) = \{X, B\}$$

$$m \cup \{Y \rightarrow 7\} = \{Y \rightarrow 7, X \rightarrow 5, B \rightarrow true\}$$

$$\{X, Z\} \lhd m = \{X \rightarrow 5\}$$

$$\{B\} ⩤ m = \{X \rightarrow 5\}$$

$$n = \{C \rightarrow false, X \rightarrow 8\} \qquad m \dagger n = \{B \rightarrow true, C \rightarrow false, X \rightarrow 8\}$$

$$n \dagger m = \{B \rightarrow true, C \rightarrow false, X \rightarrow 5\}$$

Fig. 2. Some map operators

We have written some rules to explain how sequents are obtained from other sequents. A rule is composed of a possibly empty set of sequents on the numerator, the rule premises, and of a sequent at the denominator, the conclusion of the rule. If the premises hold, then the conclusion holds.

The rules we present in figure 3 express the simplification of logical or numerical expressions. They belong to the *eval* system, which is a subsystem of the simplification system $\longrightarrow$. The first rule has no premise. It specifies that a variable X which belongs to the environment is simplified into a constant which is equal to its value C. To evaluate an expression E1 OP E2 to the value T, its two operands E1 and E2 must have been evaluated to E'1 and E'2 respectively, and the value T is the result of the computation of E'1 OP E'2 (through the *comp* system). If E'1 and E'2 are both constants (respectively N1 and N2), the computation of T is processed by the application of the *app* primitives to the operator OP and to its two operands N1 and N2.

eval

$$H \cup \{X \rightarrow C\} \vdash \mathrm{id}(X) \longrightarrow C$$

$$\frac{H \vdash E1 \longrightarrow E'1 \qquad H \vdash E2 \longrightarrow E'2 \qquad \overset{comp}{\vdash} OP, E'1, E'2 : T}{H \vdash E1\ OP\ E2 \longrightarrow T}$$

$$\frac{E'i \neq \mathit{number}\ (N)}{\overset{comp}{\vdash} OP, E'1, E'2 : E'1\ OP\ E'2} \quad \text{for } i=1,2$$

$$\frac{\mathit{app}\ (OP, N1, N2, T)}{\overset{comp}{\vdash} OP, \mathit{number}(N1), \mathit{number}(N2) : T}$$

with

$$\begin{cases} \mathit{app}\ (+, I, J, \mathit{number}(Z)) \text{ :- } Z \text{ is } I + J \\ \mathit{app}\ (=, I, J, \mathit{bool}(\text{true})) \text{ :- } I = J \\ \mathit{app}\ (=, I, J, \mathit{bool}(\text{false})) \text{ :- not } (I = J) \\ \ldots, \end{cases}$$

by using C-Prolog like evaluation predefinite primitives.

Fig. 3. Simplification of expressions

The rule of figure 4-a expresses the propagation through a sequence of statements.

To propagate the environment H through the sequence of statements I1;I2, H is propagated through the statement I1, which updates H in H1. This new environment H1 is propagated through I2, which updates H1 in H2. H2 is the environment resulting from the propagation through the sequence.

$$\frac{H \vdash I1 : H1 \qquad H1 \vdash I2 : H2}{H \vdash I1;\ I2 : H2}$$

Fig. 4- a. Propagation through a sequence of statements

The rule of figure 4-b expresses the simplification of such a sequence. Given an environment H, to simplify a sequence of statements I1;I2 the first statement I1 is simplified in I'1, and the environment H is propagated through I1. In this new environment H1, the second statement I2 is then simplified.

$$\boxed{\dfrac{H \vdash I1 \longrightarrow I'1 \qquad H \vdash I1{:}H1 \qquad H1 \vdash I2 \longrightarrow I'2}{H \vdash I1;\, I2 \longrightarrow I'1;\, I'2}}$$

Fig . 4- b. Simplification of a sequence of statements

Figure 5 presents some 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 of the then-branch (first rule: propagation),
- the simplification of the alternative is the simplification of its then-branch (second rule: simplification).

The last rule of figure 5 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 this equality is added to the environment related to the statements of the then-branch (but this equality is not inserted as assignment in the code).

There is a corresponding rule for a condition of an alternative expressed as an inequality such as X≠E: this condition is transformed in an equality such as X=E which is added to the environment related to the statements of the else-branch. Such rules have been generalized to conditions of alternatives expressed as conjunctions of equalities and disjunctions of inequalities. The last rule expresses that only equalities between variables and constants can be added to the environment. Thus, if other information are expressed in the condition, they are not taken into account by the partial evaluator.

$$\frac{\overset{\textit{eval}}{H \vdash C \longrightarrow \text{true}} \qquad H \vdash \text{I1:H'}}{H \vdash \text{if C then I1 else I2 fi: H'}}$$

$$\frac{\overset{\textit{eval}}{H \vdash C \longrightarrow \text{true}} \qquad H \vdash \text{I1} \longrightarrow \text{I'1}}{H \vdash \text{if C then I1 else I2 fi} \longrightarrow \text{I'1}}$$

$$\frac{\overset{\textit{eval}}{H \vdash E \longrightarrow \text{number(N)}} \quad X \notin \text{dom(H)} \quad H \cup \{X \rightarrow N\} \vdash \text{I1:H1} \quad H \vdash \text{I2:H2}}{H \vdash \text{if } (X = E) \text{ then I1 else I2 fi: H1} \cap \text{H2}}$$

Fig. 5. Some rules for alternatives

The propagation does not depend on the simplification. But since the simplification is performed in the context of the propagation, we have chosen to represent rules grouping propagation and simplification. They are very close to the rules we have implemented in the Foresys [Foresys93] toolkit, which compile them into Prolog. Foresys has been built upon the Centaur/Fortran environment [Centaur90].

4.2. Combined rules

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

$$H \vdash I \twoheadrightarrow I',H' \quad \text{iff} \quad H \vdash I{:}H' \quad \text{and} \quad H \vdash I \longrightarrow I'$$

From this rule, we may define inductively the $\twoheadrightarrow$ relation. For instance, figure 6 shows the rule for a sequence of statements. A sequence is evaluated from left to right: The partial evaluation of a sequence of two statements I1 and I2 consists in simplifying I1 in I'1 then in updating (add, delete or modify) the data environment H. In this new data environment H1, I2 is simplified in I'2 and H1 is modified in H2.

$$\frac{H \vdash \text{I1} \twoheadrightarrow \text{I'1, H1} \qquad \text{H1} \vdash \text{I2} \twoheadrightarrow \text{I'2, H2}}{H \vdash \text{I1; I2} \twoheadrightarrow \text{I'1; I'2, H2}}$$

Fig. 6. Partial evaluation of a sequence of statements

In the sequel of this paper, we call such rules partial evaluation rules.

Figure 7 specifies the rules for partial evaluation of assignments. The *eval* notation refers to the formal system of rules which simplifies the expressions, that we have previously presented.

If the expression E evaluates to a numerical constant N, the assignment X := E can be removed in the simplified program, and the data environment H is modified: the value of X is N whether X had already a value in H or not.

If E is only partially evaluable in E', the expression E is modified in consequence in the assignment X:= E and the variable X is removed from the environment if it was in it.

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

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

Fig. 7. Partial evaluation of assignments

The following examples illustrate these two cases. In example Ex.1, as the value of the variable A is known, the assignment C := A+1 can be removed from the simplified program and the new value of the assigned variable C is introduced in the data environment. In example 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.

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

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

The rules for partial evaluation of alternatives are defined in figure 8. If the condition C evaluates to a logical constant, the alternative with condition C can be simplified to the corresponding simplified branch. If C is only partially evaluated in C', the partial evaluation proceeds along both branches of the alternative. It leads to two different environment H'1 and H'2. Their intersection is the final environment.

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{false}) \qquad H \vdash I2 \relbar\joinrel\blacktriangleright I'2, H'}{H \vdash \text{if } C \text{ then } I1 \text{ else } I2 \text{ fi} \relbar\joinrel\blacktriangleright I'2, H'}$$

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{true}) \qquad H \vdash I1 \relbar\joinrel\blacktriangleright I'1, H'}{H \vdash \text{if } C \text{ then } I1 \text{ else } I2 \text{ fi} \relbar\joinrel\blacktriangleright I'1, H'}$$

$$\frac{H \overset{eval}{\vdash} C \longrightarrow C' \qquad C' \neq \mathit{bool}(B) \qquad H \vdash I1 \relbar\joinrel\blacktriangleright I'1, H'1 \qquad H \vdash I2 \relbar\joinrel\blacktriangleright I'2, H'2}{H \vdash \text{if } C \text{ then } I1 \text{ else } I2 \text{ fi} \relbar\joinrel\blacktriangleright \text{if } C' \text{ then } I'1 \text{ else } I'2 \text{ fi}, \ H'1 \cap H'2}$$

Fig. 8. Partial evaluation of alternatives

As for alternatives, the rules for partial evaluation of loops, presented in figure 9, depend on the ability to evaluate the truth or falsity of the condition C from the current environment H. The first rule specifies that if the loop is not entered, it is removed from the code. There is no specific rule for the case where C evaluates to true, because we do not expand loops, not to alter the structure of the code. Figure 9 shows the rules for while-statements, but similar rules exist for repeat-statements.

In the second rule, if C evaluates in C' (and the value of C' differs from false), the statements I of the loop can be simplified, given H restricted to a loop invariant Inv(I). Inv(I) is a pessimistic estimation of the variables that are not modified in the loop. It is calculated by the partial evaluator and consists in a list of variables whose values are known and that are neither in a left-hand side of an assignment, nor a parameter of a call or a read statement. The sequent that transforms I into I' belongs to the simplification system. Since we have imposed a pessimistic loop invariant, we did not write a sequent referring to the system $\relbar\joinrel\blacktriangleright$: performing the propagation through I would not have modified the restricted environment.

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{false})}{H \vdash \text{while } C \text{ do } I \text{ end} \relbar\joinrel\blacktriangleright \text{skip}, H}$$

$$\frac{\mathrm{Inv}(I) \lhd H \overset{eval}{\vdash} C \longrightarrow C' \qquad C' \neq \mathit{bool}(\text{false}) \qquad \mathrm{Inv}(I) \lhd H \vdash I \longrightarrow I'}{H \vdash \text{while } C \text{ do } I \text{ end} \relbar\joinrel\blacktriangleright \text{while } C' \text{ do } I' \text{ end}, \ \mathrm{Inv}(I) \lhd H}$$

Fig. 9. Partial evaluation of while-loops

5. Conclusion

We have used partial evaluation for programs that are difficult to maintain because they are too general. Specialized programs for some values of their input variables are obtained by propagating these constant values (through a normalisation of the expressions) and by performing simplifications on the code, for instance assignments are removed and alternatives are reduced to one of their branches. This technique helps the maintainer to understand the program behaviour in a particular context. The residual program is furthermore more efficient because many statements and variables have been removed in it, and no additional statement has been inserted. Another advantage of this technique is that it can also be applied to abstractions at a higher level than the code.

Let us notice that our tool may be used in two ways: by visualizing the residual program as a part of the initial program (for documentation or for debugging) or by generating this residual program as an independent (compilable) program.

We are now focusing on the possibility for the user to supply general properties about input variables. These general properties are relational expressions composed of some literal values (e.g. x<z+4) instead of equalities to constant values. We will consequently take into account that kind of information in the conditions of alternatives and loops. We intend to apply linear resolution methods and symbolic manipulation packages to propagate such properties.

References

[ANSI78] *Programming language Fortran* ANSI standard X3.9 1978.

[Andersen91] L.O.Andersen *C program specialization* Master's thesis, University of Copenhagen, May 1992, 160 pages.

[BerlinWeise90] A.Berlin, D.Weise *Compiling scientific code using partial evaluation*, Computer, December 1990, 25-37.

[Centaur90] Centaur group *The Centaur 1.1 documentation* January 1990.

[Coen-Porsini&al.91] A.Coen-Porsini, F.De Paoli, C.Ghezzi, D.Mandrioli *Software specialization via symbolic execution* IEEE Transactions on Software Engineering, 17(9), September 1991, 884-899.

[ErshovOstrovski87] A.P.Ershov, B.N.Ostrovski *Controlled mixed computation and its application to systematic development of language-oriented parsers* Program Specification and Transformation, IFIP'87, 31-48.

[Foresys93] Connexite *Reference manual of the Foresys line of software products* 1993.

[GallagherLyle91] K.B.Gallagher, J.R.Lyle *Using program slicing in software maintenance* IEEE Transactions on Software Engineering, 17(8), August 1991, 751-761.

[Haziza&al.92] M.Haziza, J.F.Voidrot, E.Minor, L.Pofelski, S.Blazy *Software maintenance: an analysis of industrial needs and constraints* IEEE Conference on Software Maintenance, Orlando, USA, November 1992.

[Jones90] C.B.Jones *Systematic software development using VDM* Prentice-Hall, 2nd eds., 1990.

[JonesSestoftSondergaard89] N.D.Jones, P.Sestoft, H.Sondergaard *MIX: a self-applicable* *partial evaluator for experiments in compiler generation* Lisp and Symbolic Computation 2, 1989, 9-50.

[Kahn87] G.Kahn *Natural semantics* Proceedings of STACS'87, Lecture Notes in Computer Science, vol.247, March 1987.

[Kasyanov91] V.Kasyanov *Transformational approach to program concretization* Theoretical computer science, 90, 1991, 37-46.

[KemmererEckmann85] R.Kemmerer, S.Eckmann *UNISEX: a UNIX-based Symbolic Executor for Pascal Software Practice and Experience*, 15(5), 1985, 439-457.

[Meyer91] U.Meyer *Techniques for evaluation of imperative languages* ACM SIGSOFT, March 1991, 94-105.

[Nicolas&al.89] G.Nicolas, S.Aubry, E.Briere *A finite volume approach for 3D two phase flows in tube bundles: the THYC code* Kernforschungscentrum, Karlsruhe, Vol.2, 1989, 1247-1253.

[Sahlin92] D.Sahlin *An automatic partial evaluator for full Prolog* Ph.D. thesis, Swedish Institute of Computer Science, March 1991, 170 pages.

[Sneed89] T.H.Sneed *The myth of 'top-down' software development and its consequences* IEEE Conference on Software Maintenance, Miami, USA, October 1989, 22-29.

[VanZuylen92] H.J.Van Zuylen *Understanding in reverse engineering* In REDO team *The REDO handbook* Wiley eds., September 1992.

[WegmanZadeck91] M.N.Wegman, K.Zadeck *Constant propagation with conditional branches* ACM Transactions on Programming Languages and Systems 13(2), April 1991, 181-210.