Application of Formal Methods to the Development of a Software Maintenance Tool

Sandrine Blazy, Philippe Facon
CEDRIC IIE, 18 allée Jean Rostand, 91 025 Évry Cedex, France
{blazy, facon}@iie.cnam.fr http://www.iie.cnam.fr/~{blazy, facon}/

Abstract

Partial evaluation is an optimization technique traditionally used in compilation. We have adapted this technique to the understanding of scientific application programs during their maintenance and we have implemented a tool. This tool analyzes Fortran 90 application programs and performs an interprocedural pointer analysis. This paper presents how we have specified this analysis with different formalisms (inference rules with global definitions and set and relational operators). Then we present the tool implementing these specifications. It has been implemented in a generic programming environment and a graphical interface has been developed to visualize the information computed during the partial evaluation (values of variables, already analyzed procedures, scope of variables, removed statements, ...).

Keywords: program understanding, partial evaluation, formal specification, interprocedural analysis, alias analysis

1. Introduction

A wide range of software maintenance tools analyze existing application programs in order to transform them. Some of these transformations aim at facilitating the understanding of programs and they may perform rather complex analyses. This is due either to the programming language itself (e.g. `COMMON` in Fortran) or to the analysis itself (e.g. an interprocedural alias analysis). As software maintenance tools, these tools must introduce absolutely no unforeseen changes in programs. To overcome these problems, we have used formal specifications to develop a software maintenance tool. In our framework, a formal specification yields:

• A basis for expressing precisely which transformations are performed. The formal specification can be seen as a reference document between specifiers and end-users. Formal concepts are powerful enough to clarify concepts of programming languages and to model complex transformations. In our context, end-users were software maintainers who had a strong background in mathematics. Thus, they were disposed to understand our formal specifications.

• A mathematical formalism for proving and validating properties of program transformations.

• A framework for simplifying the implementation of a tool.

This tool aims at improving the understanding of scientific application programs. These application programs are difficult to maintain mainly because they have been developed a few decades ago by experts in physics and mathematics, and they have become very complex due to extensive modifications. For a maintenance team working on a specific application program, one of the most time consuming steps was to extract by hand in the code the statements corresponding to their specific context [4].

Our tool is based on partial evaluation, an optimization technique, also known as program specialization. When given a program and known values of some input data, a partial evaluator produces a so-called residual or specialized program. Running the residual program on the remaining input data will yield the same result as running the original program on all of its input data [11]. Partial evaluation has been applied to generate compilers from interpreters (by partially evaluating the interpreter for a given program). In this context, previous work has primarily dealt with functional [9] and declarative languages [13]. Partial evaluation has also been applied to improve speedups of imperative programs ([2]-[3]). We have adapted this technique to program understanding.

Usually, the chief motivation for doing partial evaluation is speed. The residual program is faster than the initial one because statements have been unfolded each time they could be replaced by faster statements ([2]-[3]). Statements unfolding replaces procedure calls and loops by their unfolded body. We have not used this partial evaluation technique because it modifies the structure of the code. In the same way, our partial evaluator does not generate new variables nor rename variables, as it is done in classical

partial evaluation for optimizing the residual code. The residual code we generate is easier to understand because many statements and variables have been removed and no additional statement or variable has been inserted. The known values of variables like PI or TAX_RATE are propagated during partial evaluation but these variables are likely to be kept in the code (2*PI+1 should be easier to understand than 7.28). The benefit of replacing variables by values depends also on the kind of user (see [4] for details about our specialization strategy).

Fig.1 briefly illustrates how an initial code is specialized into a residual code, with respect to constraints on input variables. [4] explains what these constraints mean. In Fig.1 the initial code that has been removed is written in bold. In initial and residual codes, simplified expressions are underlined. Known values of variables are propagated in called procedures. Called procedures have been replaced by their specialized versions and a comment recalls the name of the called procedure and its initial known values. Other information are computed and displayed during the partial evaluation (e.g. final values of some variables), but they are not shown in Fig.1 not to overload it. To make Fig.1 clearer, some Fortran statements are written on a same line.

At the very beginning, our aim was to specify and develop a generic prototype tool that could simplify application programs written in any imperative language. This tool was implementing a general (but simple) intraprocedural analysis that was simplifying some assignments and alternatives [5]. The formal specification was consisting only of inference rules in natural semantics operating on abstract syntax trees [12].

These rules were quite easy to understand: they were made of sequents defining a propagation relation ($S_1 \vdash I: S_2$ means that the execution of the statement I modifies the initial state S_1 into the final state S_2), a simplification relation ($S \vdash I_1 \rightarrow I_2$ means that given the state S, the statement I_1 simplifies into I_2), and the combination of both for defining a partial evaluation relation ($S_1 \vdash I_1 \twoheadrightarrow I_2, S_2$ means that given the state S_1, the specialization of the statement I_1 yields a simplified statement I_2 and a new state S_2). In natural semantics, each rule expresses how to deduce sequents (the denominator of the rule) from other sequents (the numerator of the rule). Our sequents were simple because propagated data were only made of a map S from variables to their values (when a variable has a known value at the current program point). Since the formal specifications were simple, it was also easy to derive from the specifications an implementation of a prototype tool [5].

We have then added to our prototype a very precise interprocedural analysis. To specify in our interprocedural analysis side-effects on global variables and side-effects accomplished through parameter passing, we need information about the data that a procedure inherits and about the side effects of procedures that it invokes. To account for this effect, we must model the transmission of values from within a procedure back to the call site that invoked it. The last analysis we have specified is a pointer analysis for Fortran 90. The partial evaluation simulates the run time memory management. Due to the implicit connections through paths within a pointer structure, the side-effects of pointer assignments have been modeled by other information than those for modeling assignments to a simple variable.

Natural semantics rules are useful to show how relations are recursively called. This formalism is concise and comprehensible enough to specify a simple partial evaluation process. We have extended it to specify an interprocedural alias analysis. To this end we have used in our natural semantics rules various set and relational operators and we have structured data appearing in the rules. We have modeled the links between these data by object diagrams. The diagrams show variables used in rules and other variables that are defined outside the rules not to overload them.

The aim of this paper is twofold: to show how we have specified these extensions to our partial evaluator, and to detail the implementation of the partial evaluator. Compared to our previous work, our program analyses are much more refined (by alias analysis) and modular. Furthermore, we have implemented a graphical interface. Before specifying natural semantics rules, we have defined object diagrams for structuring modelled data. We have also adapted our specifications to allow local definitions. The specifications presented here focus on the reuse of specialized versions and on the pointer analysis. In these specifications, we have not mixed both tasks not to overload the specifications. But of course, our final specification and implementation combine both tasks. The rest of this paper is structured as follows. First, Section 2 recalls some concepts of Fortran 90 and explains our specialization strategy for reusing specialized procedures. Then, Section 3 details the specification of the interprocedural pointer analysis. Section 4 is devoted to the implementation of our tool.

2. Background

2.1 Fortran 90

Fortran procedures may be subroutines or functions and parameters are passed by reference. Variables are usually local entities. However, variables may be grouped in common blocks (a common block is a contiguous area of memory) and thus shared across procedures. Common blocks may also be inherited in a procedure. They have a

```
SUBROUTINE INIG (X,DX, IDEC, DXL)
COMMON /GEO1/ IM,IMM1,JM, KM, IMATSO
COMMON / GEO2 / INDX_I, INDX_J, INDX_K
COMMON / FILE1 / NFIC11 , NFIC12 , NFIC6
IF (IREX .NE. 0) THEN
   IF (DXL .EQ. 0) THEN  WRITE ( NFIC12 , 1001 ) DXL;  CALL STOP ( 'INIGEO' )
   ENDIF
   IF ( IM.NE.0 ) THEN READ ( NFIC11 , * , ERR = 1102 ) XMIN
   ELSE   XMIN = 0.
   ENDIF
   DO 111 , I = 1 , IM
      X(I) = XMIN + FLOAT(I-1) * DXL
111 CONTINUE
   IMM1 = IM - 1
   DO 112 , I = 1 , IMM1
      DX(I) = DXL
112 CONTINUE
ELSE
   READ (NFIC11,*,ERR=1103) X
   DO 121 , I = 1 , IMM1
      DX(I)= X(I+1) - X(I)
121 CONTINUE
ENDIF
IF ( IMATSO .EQ. 0 AND IM .GE. 10) THEN
   CALL VALMEN (MAT , IMM1 , -1)
   IF ( IREX .EQ. 0 ) THEN  WHAT = '  K'
   ELSE
      CALL VALMEN (MAT , IM , 3)
      IF ( IDESCREG.EQ.0 ) THEN  IREGU = 0
      ELSE IREGU = 1; IDESC = IDESCREG
      ENDIF
      IF ( IDESC.EQ.0 .AND. IREGU.EQ.0 ) THEN WHAT = '  K'; IDEC = 3
      ELSE
        IF ( INDX_I .NE. 0 ) THEN WHAT = '  I'; IDEC = 1
        ELSE IF ( INDX_J .NE. 0 ) THEN WHAT = '  J'; IDEC = 2
        ELSE IF ( INDX_K .NE. 0 ) THEN WHAT = '  K'; IDEC = 3
        ELSE   CALL STOP ( 'INIG' )
        ENDIF
      ENDIF
      IF ( IDEC.EQ.1 ) THEN
        IF ( IREGU.EQ.0 ) THEN IMIN = 2; IMAX = IM
        ELSE IMIN = IM; IMAX = IM                ENDIF
      ELSE IF ( IDEC.EQ.2 ) THEN
        IF ( IREGU.EQ.0 ) THEN JMIN = 2; JMAX = JM
        ELSE JMIN = JM; JMAX = JM                ENDIF
      ELSE IF ( IDEC.EQ.3 ) THEN
        IF ( IREGU.EQ.0 ) THEN KMIN = 2; KMAX = KM
        ELSE KMIN = KM; KMAX = KM                ENDIF
      ENDIF
   ENDIF
ENDIF
IF (IWARNI . GE . 9 . OR . IREX . EQ . 1) THEN  WRITE ( NFIC6 , 6060 ) X ENDIF
CALL STOP ( 'INIG' )
END
```

Initial code

```
IREX = 1
IDESCREG = 3
INDX_I = 2
IM = 20
IND_X = 2
DXL = 0.5
```

Constraints on input variables

```
SUBROUTINE INIG (X,DX,IDEC, DXL)
COMMON /GEO1/ IM,IMM1, JM, KM, IMATSO
COMMON /GEO2/ INDX_I , INDX_J, INDX_K
COMMON /FILE1/ NFIC11, NFIC12, NFIC6
XMIN = 0.
DO 111 , I = 1 , 20
   X(I) = XMIN + FLOAT(I-1) * 0.5
111 CONTINUE
IMM1 = 19
DO 112 , I = 1 , 19
   DX(I) = 0.5
112 CONTINUE
IF ( IMATSO .EQ. 0) THEN
   CALL VALMEN_v1 (MAT , 19 , -1)
C specialized version of ... with ...
   CALL VALMEN_v2 (MAT , 20 , 3)
C specialized version of ... with ...
   IREGU = 1
   IDESC = 3
   WHAT = '  I'
   IDEC = 1
   IMIN = 20
   IMAX = 20
ENDIF
WRITE ( NFIC6 , 6060 ) X
CALL STOP_v1 ( 'INIG' )
C specialized version of STOP with ...
END
```

Specialized code

Figure 1. An example of program specialization

scope in that procedure but they have not been declared in it. If a common block is neither declared in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block are undefined. The only exceptions are variables that have been defined in a DATA statement (this statement allows initialization of variables) and never changed.

In Fortran 90 a structure consisting of a list of fields, each of some particular type, is a type. The fields types may include pointers to structures of the type being defined, of a type previously defined, or of a type to be defined. A pointer variable, or simply a pointer may point to either another data object which has the TARGET attribute, or an area of dynamically allocated memory, or the NULL value. In Fortran 90, a pointer should be thought of as a variable associated dynamically with or aliased to another data object where the data is actually stored - the target [14]. There is no notation for representing pointed variables (dereferencing is automatic in Fortran 90).We will then use a C-notation when needed (e.g. Fig.2 and Fig.3).

```
TYPE node
   REAL:: name                    ! data field
   TYPE(node), POINTER:: next !pointer field
END TYPE node
TYPE(node), POINTER :: p, q
...
q => p%next              ! q points to *(p->next)
p%name = 3.4             ! the value 3.4 is assigned to
                           the field 'name' of p
q%name = 6.2
```

Figure 2. An example of Fortran 90 program

Fig.2 shows an example of a Fortran 90 program, where a new type called node is defined. It will be constructed from two values representing a name and a pointer to the next field in a linked list. Two variables of type pointer to node are also declared. Then, the values 3.4 and 6.2 are inserted in the list in that order.

Abstract syntax

VarName $\subseteq$ Lhs

deref: Lhs $\rightarrow$ Lhs

field_lhs: Lhs $\times$ VarName $\rightarrow$ Lhs

Examples

concrete syntax	**abstract syntax**
v	VarName (*v*)
person%address%town	**field_lhs** (**field_lhs** (*person*, *address*), *town*)
*(*p->next*)	**deref** (**field_lhs** (**deref** (*p*), *next*)

Figure 3. Abstract syntax rules and examples of links with concrete syntax

A variable identifier is either a simple identifier (e.g. *v*), or a compound left-hand side (e.g. *person%address% town*), or a pointer dereference (e.g. **p*, *(**p.next*)). This is represented by the abstract syntax of Fig.3. The set of simple variables identifiers is denoted by VarName. The set of left-hand sides is denoted by Lhs. The example in this figure shows the connection between some concrete Fortran 90 variables and the corresponding abstract syntax notations.

2.2 Interprocedural Partial Evaluation

The specialization proceeds depth-first in the call-graph to preserve the order of side-effects. Thus, the specialization of a call statement first runs the specializer on the called procedure SP. This yields a specialized version of SP and the call statement is replaced by a call to this specialized version. A procedure is specialized with respect to specific values of some of its input data. At the end of its specialization, the known values of variables belong to its output static data, and a new name is given to the new specialized version (if any). Fig.4 presents data modeling specialized versions. It shows that a specialized version of a procedure consists of a name, input data, output data and statements. In other words, a version is represented by a quintuplet (name of original procedure, version name, input data, output data, statements).

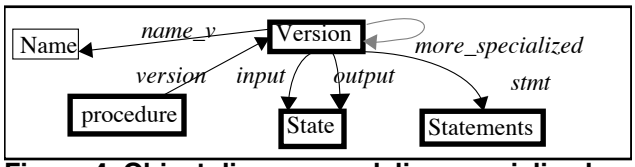

Figure 4. Object diagram modeling specialized versions

To improve the specialization, specialized versions of procedures are propagated and reused, as shown in Fig.5. Thus, given a set of specialized procedures, when a call to a procedure SP is encountered in the current procedure, if the set of static data of SP and their values:

- is the *same* as those of a previous call (as in ① of Fig.5), then the corresponding version is directly reused,
- *strictly includes* those of a previous call (as in ② and ③), then the corresponding version is specialized and added to the already specialized versions. If several versions match, the following selections are successively made:
  - most specialized versions, that is the versions with the largest set of static data, as in ③ where SP4 is selected,
  - shortest version among most specialized versions (as in ② where SP3 is selected).

A specialized version v of a procedure SP is more specialized than a version v' of the same procedure if the input static data of v' is included in those of v. For instance, in Fig.5, SP1 is more specialized than SP2. Thus, we define:

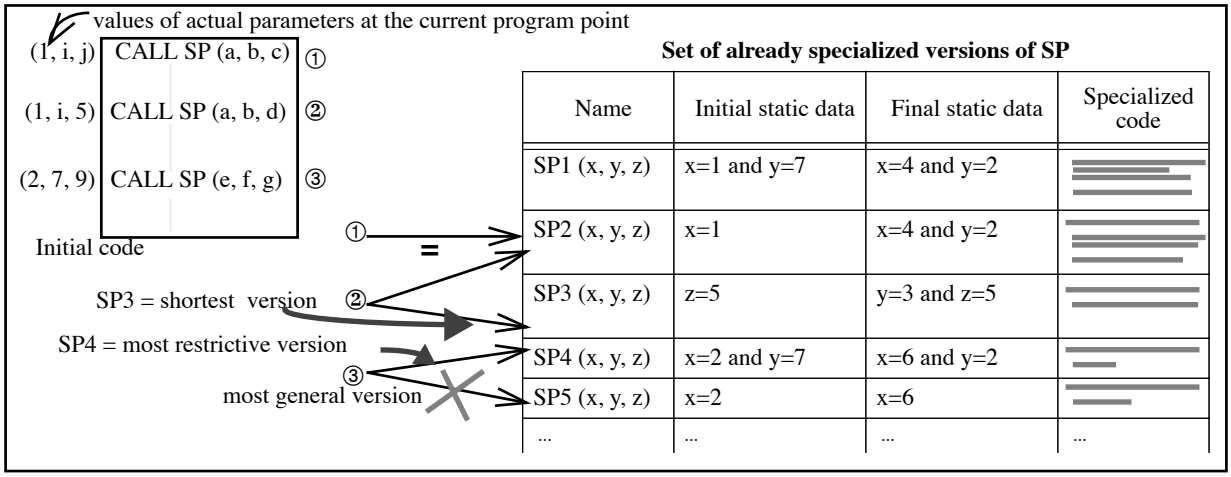

Figure 5. Reuse of specialized versions

more_specialized (v, v') Δ *input* (v') $\subseteq$ *input* (v), where Δ means "is defined as". Given input static data In, v is one of the most specialized versions of SP iff v is maximal among the specializable versions of SP with respect to In. This is expressed by the following predicate: *most_specialized* (v, SP, In) Δ *input* (v) $\subseteq$ In $\wedge$ ($\nexists$ v' $\in$ *version* (SP) . *input* (v) $\subset$ *input* (v') $\subseteq$ In). When *input* (v) = In, v is the only specialized version of SP with respect to In.

The number of versions of a procedure may theoretically grow exponentially, but our experiments showed that this seldom happens. However, as the number of specialized versions is finite (an option of the specializer enables changing it), if a version must be removed (from the set of versions), either the most restrictive or the most general one is removed. With a general strategy, specialized procedures are more often reused than in the restrictive strategy, but more statements should also be specialized. In a general framework and without any further analysis on the call graph, both strategies are worthwhile, depending on the application to specialize. Thus, an option of the specializer enables changing this strategy and keeping preferably the most general procedures.

3. Formal Specification of the Partial Evaluation

3.1 Definitions

We define in this section some notations, especially set operators, that we use in our specifications. **PROCNAME** denotes the set of possible identifiers of procedures and **VALUE** denotes the set of possible values of variables. The **eval** function either yields the value of an expression (if it is known) or gives a residual expression. We introduce useful set operators, similar to those defined in the formal specification languages B [1] and VDM [10]: mainly inverse ($^{-1}$), domain (**dom**), range (**ran**), $\cup$, override ($\dagger$), restrictions ($\rhd$ and $\lhd$), composition (**;**) and direct product ($\otimes$). These operators are written in bold in this paper. In the following definitions s denotes a set, r and p denote binary relations (sets of pairs), m and n denote maps (specific binary relations where each element has at most one image).

- $r^{-1} = \{ x \rightarrow y \mid y \rightarrow x \in r\}$
- $m \dagger n = \{x \rightarrow y \mid x \rightarrow y \in n \vee (x \rightarrow y \in m \wedge x \notin \mathbf{dom}(n))\}$
- $m \rhd s = \{x \rightarrow y \mid y \in s\}$
- $s \lhd m = \{x \rightarrow y \mid x \notin s\}$
- $r \mathbf{;} p = \{x \rightarrow z \mid \exists y . x \rightarrow y \in r \wedge y \rightarrow z \in p\}$
- $r \otimes p = \{x \rightarrow (y,z) \mid x \rightarrow y \in r \wedge x \rightarrow z \in p\}$
- Given s a set of pairs of maps, we define **Corres** (s) = $\cup \{x^{-1} \mathbf{;} y \mid x \rightarrow y \in s\}$. We use **Corres** (s) to bind variables of a common block to their corresponding values. Each pair of s corresponds to a common block with its list of variables. Variables of common blocks are shared among procedures (their values are inherited in each called procedure) but their names may change in each procedure. Thus, for common blocks, variable names and their values are specified by two different maps from integers (the position in the list of declared variables of the common block) to respectively variable names and values.

3.2 Interprocedural Analysis

As previously [5], we have specified with inference rules both the constant propagation and the statements simplification performed by our specializer. But more data are propagated in the inference rules. Given a Fortran program, we propagate:

- an environment, that represents what does not vary during the partial evaluation (mainly formal parameters, declared data and statements),
- a state modeling relations between variables and values at the current program point,
- specialized versions,
- inherited common blocks of the current procedure.

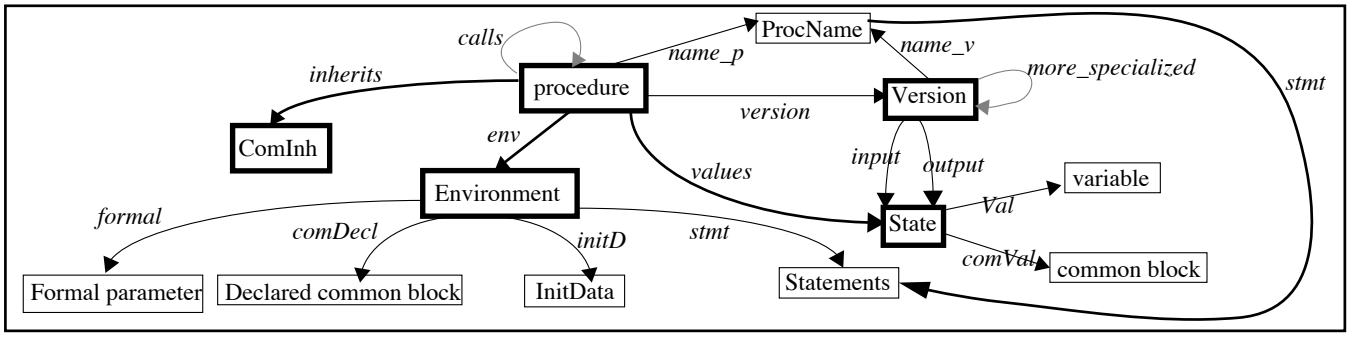

Figure 6. Object diagram of data for specifying the interprocedural analysis

[4] gives examples of these data and Fig.6 models the whole propagated data. This model represents objects and access functions between them. It extends the model of Fig.4. In both figures, the object procedure denotes a called procedure at a given program point. In a procedure, known values are related either to variables of common blocks (and accessed through the *comVal* function) or to other variables (*Val* function).

Fig.7 specifies the simplification of a call statement to a procedure SP when specialized versions of SP are reused. In the definitions part of the figure, some definitions are factorized. They are here to introduce some useful variables appearing in the rules. Definitions are here "macros" that are supposed to be applied to the rules containing the variables. [4] illustrates and details these definitions and explains the propagation rule through a call statement.

The two rules of Fig.7 correspond to the following situations that may occur.

Definitions

EnvSP $\underline{\Delta}$ *calls*(SP) **;** *env*	**results from the**
StaticFormal $\underline{\Delta}$ (*formal*$^{-1}$ (EnvSP) **;** LParam) **; eval** (*Val* (State)) $\triangleright$ **VALUE**	**propagation through**
StaticCom1 $\underline{\Delta}$ **Corres** (*comDecl*$^{-1}$(EnvSP) **;** *comVal* (State))	**the call statement**
Input $\underline{\Delta}$ (*SV* (State) † *initData* (EnvSP)) $\cup$ (StaticFormal † StaticCom1)	**call SP (LParam)**
ComInh' $\underline{\Delta}$ **dom** (*comDecl* (Env)) $\cup$ ComInh $-$ **dom** (*comDecl* (EnvSP))	
State1 $\underline{\Delta}$ **(Val** $\otimes$ *comVal*)$^{-1}$ (Input, StaticCom1)	
NewVersion $\underline{\Delta}$ (SP, NewName, State, State2, SP')	new version of SP

Simplification rules

Rule 1: the called procedure has already been specialized with respect to the same static data

$$\frac{V \in \textit{version}\,(SP) \qquad \textit{input}\,(V) = State1 \qquad EnvSP, \textit{output}\,(V), ComInh' \overset{\textit{Propagation}}{\vdash} \textit{stmt}\,(V) : State2}{Env, State, ComInh, Version \vdash \textbf{call}\ SP\ (LParam) \rightarrow \textbf{call}\ \textit{name_v}\ (V)\ (LParam), Version} \quad (1)$$

Rule 2: the specialized version of the called procedure is not as specialized as wanted

$$\frac{State1 \notin \textit{version ; input}\,(SP) \qquad V \in \textit{version}\,(SP) \qquad \textit{most_specialized}\,(V, SP, State1) \qquad EnvSP, \textit{output}\,(V), ComInh', Version \vdash \textit{stmt}\,(V) \rightarrow SP', Version' \qquad EnvSP, \textit{output}\,(V), ComInh' \overset{\textit{Propagation}}{\vdash} SP' : State2 \qquad NewName \in \textbf{PROCNAME -}\ ProcName}{Env, State, ComInh, Version \vdash \textbf{call}\ SP\ (LParam) \rightarrow \textbf{call}\ NewName\ (LParam), Version' \cup \{\textit{NewVersion}\}} \quad (2)$$

Figure 7. Simplification of call statements

• (1) SP has already been specialized in a procedure called V, with respect to the initial static data State1 (State1 results from the propagation through the call statement to SP). In this case, the call to SP is replaced by the call to V with the same parameters, and the data related to SP are propagated through V, yielding a new state State2. These data are:

  • the environment of SP (it is accessed through *calls*): *calls* (SP) ; *env*,

  • the initial state of SP, that is the final static data of V*: output (*V*)*,

  • common blocks ComInh' that are inherited by SP (they are defined from ComInh).

• (2) SP has already been specialized in V, but with less constraints on its initial static data. Since SP has not already been specialized with respect to State1, then necessary State1 is not an initial static data of a specialized version of SP: State1 $\notin$ *version* **;** *input (*SP*)*.

  • Since SP has already been specialized, *version (*SP*)* $\neq \emptyset$ and a version V is selected in *version (*SP*)*. As explained in section 2.2, V is one of the most specialized versions of SP with respect to State1, that is *most_specialized* (V, SP, State1).

  • Then, the statements of V are specialized in the statements SP' and as in the first rule, data are propagated through SP'.

  • A new name is created for SP' and the call to SP is replaced by the call to the name of SP' with the same parameters. This new name is a possible name that is not already a procedure name: NewName $\in$ **PROCNAME -** ProcName.

  • The new version is also added among specialized versions of SP.

When there is no specializable version of SP with respect to State1, the rule for simplifying the call statement is similar to the second rule. Thus, it has not been detailed in Fig.7.

3.3 Pointer Analysis

The aim of pointer analysis is for every pointer variable to approximate the set of objects it may point to. Here, an object is a location that can store information (for example, variables). A pointer analysis is equivalent to an alias analysis. An alias occurs when the left values of two objects coincide. Taditionally, aliases are represented as an equivalence relation over abstract locations [16]. Our analysis is precise enough to represent variables which do not explicitly appear in the code but this precision is lost while analyzing conditional statements (we do not propagate conditional expressions).

In our specification, we use stores to represent associations between variables and their values. The variables are represented by locations in stores. The set of values (denoted by Value) includes integers and other values (such as locations denoted by Loc). The dynamic semantics of pointers is modeled by the following functions that are defined in Fig.8:

• *loc_of* maps (simple) identifiers to their locations.

• The map *loc_of_gen* extends the *loc_of* map to left-hand sides and dereferences. The location of a pointed record is the value of its first field.

loc_of $\in$ VarName $\rightarrow$ Loc
loc_of_gen $\in$ Lhs $\rightarrow$ Loc
  i $\in$ VarName, l $\in$ Lhs
    loc_of_gen (i) = *loc_of* (i)
    loc_of_gen (**deref** (l)) = *store* (*loc_of_gen* (l))
    loc_of_gen (**field_lhs** (l,i)) =
                access_field (*loc_of_gen* (l),i)

store $\in$ Loc $\rightarrow$ Value $\cup$ Loc $\cup$ 'NULL'
Value $\subseteq$ **VALUE**
access_field $\in$ Loc $\times$ VarName $\nrightarrow$ Loc

Figure 8. Dynamic semantics of some variables

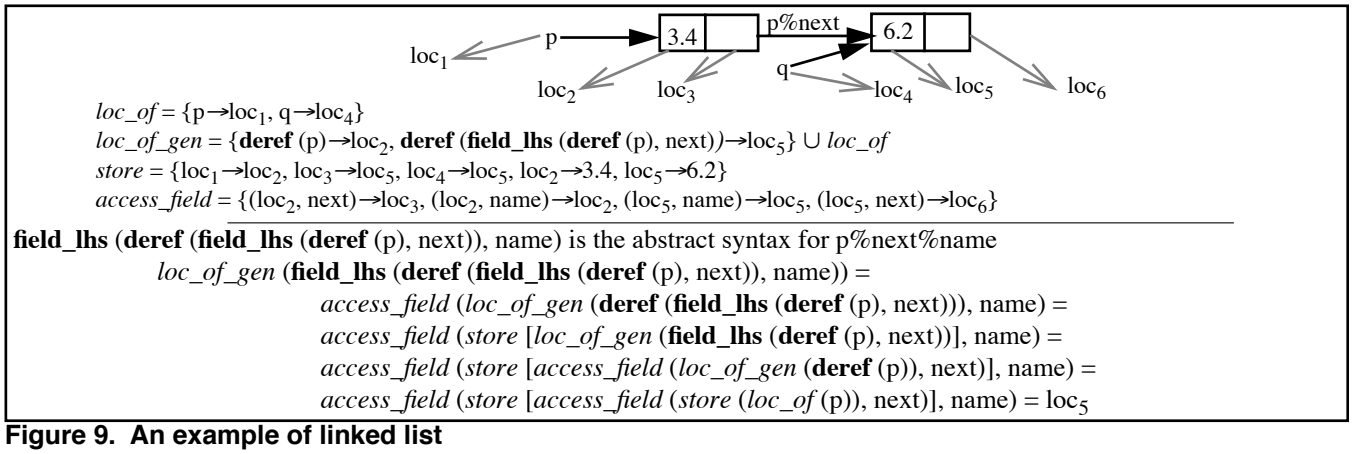

Figure 9. An example of linked list

• The store is modeled as a map *store* from locations to values. The locations give in turn access to the current values stored in variables. The value of a variable is looked up in the store through the *loc_of* map. The store of a pointer is either the location of its pointed object (if the pointer points to a target) or the NULL value.
• given the location of a record *r* and a field *f*, *access_field* yields the location of *r.f*. This is a partial function since only record names with their corresponding fields may have a location.

Fig.9 (see previous page) represents in diagrammatic form the linked list created by the statements of Fig.2. The rest of the figure shows the dynamic semantics of the corresponding statements. All pointer chaining are resolved before the two assignments, so any node can be referred to directly by its location. Each node has been dynamically allocated. Thus, each node has a unique location, as shown in the map *loc_of_gen*. The definition of this map is illustrated in the last part of Fig.9.

The map from pointers to their targets or to the NULL value is then defined as follows:

$$points_to \triangleq loc_of_gen \,;\, store \,;\, loc_of_gen^{-1}$$

Ex. *points_to* = {p → **deref** (q), r → loc, s → NULL}

$$\frac{q \in \textbf{dom}\ (points_to)}{\begin{array}{l} store, loc_of_gen \vdash p \Rightarrow q : \\ \quad store \dagger \{loc_of_gen\ (p) \rightarrow loc_of_gen\ (q)\}, \\ \quad loc_of_gen \dagger \{\textbf{deref}\ (p) \rightarrow store\ (loc_of_gen\ (q))\} \end{array}}$$

$$\frac{q \notin \textbf{dom}\ (points_to)}{\begin{array}{l} store, loc_of_gen \vdash p \Rightarrow q : \{loc_of_gen\ (p)\} \lhd store\ , \\ \quad \{\textbf{deref}\ (p)\} \lhd loc_of_gen \end{array}}$$

Figure 10. Propagation of a pointer assignment

Fig.10 shows the propagation through a pointer assignment p => q. The store is updated by the alias introduced by that assignment. q points to a target t or to the NULL value if q ∈ **dom** (*points_to*) (rule 1). Then, when p is affected by q, p points to t: *store (loc_of_gen* (p)) becomes *loc_of_gen* (q) and the location of *p becomes the location of t, *store (loc_of_gen* (q)). If there is no location pointed by q (rule 2) then p does not point to any location and *p has no location anymore.

4. The Tool

We have implemented our partial evaluator on top of a kernel that has been generated by the Centaur system [7]. The Centaur system 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 intermediate format for representing program texts is the abstract syntax tree. We have merged two specific environments (one dedicated to Fortran and an other to a language that we have defined for expressing the scope of general constraints on variables) into an environment for partial evaluation. This environment consists of structured editors for constraints and Fortran procedures (provided by Centaur), a partial evaluator, together with an uniform graphical interface. Fig.11 shows the architecture of our tool, its inputs and outputs.

The formal specifications have been implemented in a language provided by Centaur and called Typol. Typol is an implementation of natural semantics. Typol programs are compiled into Prolog code. When executing these programs, Prolog is used as the engine of the deductive system. Set and relational operators as definitions have been written directly in Prolog [8]. They are called from Typol rules. Thus, the Typol rules operate on the abstract syntax and they are close to the formal specification rules

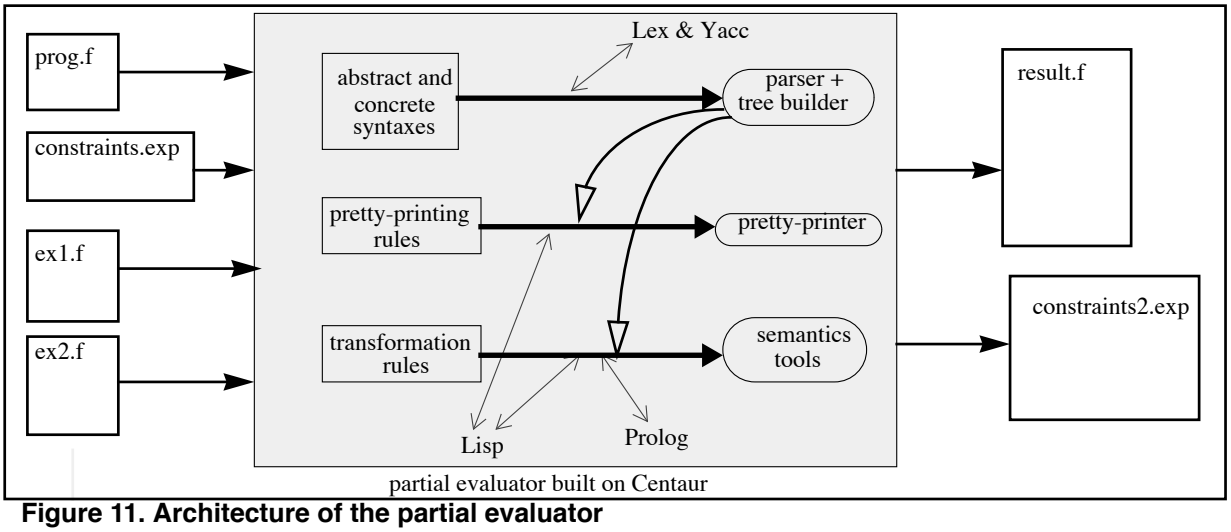

Figure 11. Architecture of the partial evaluator

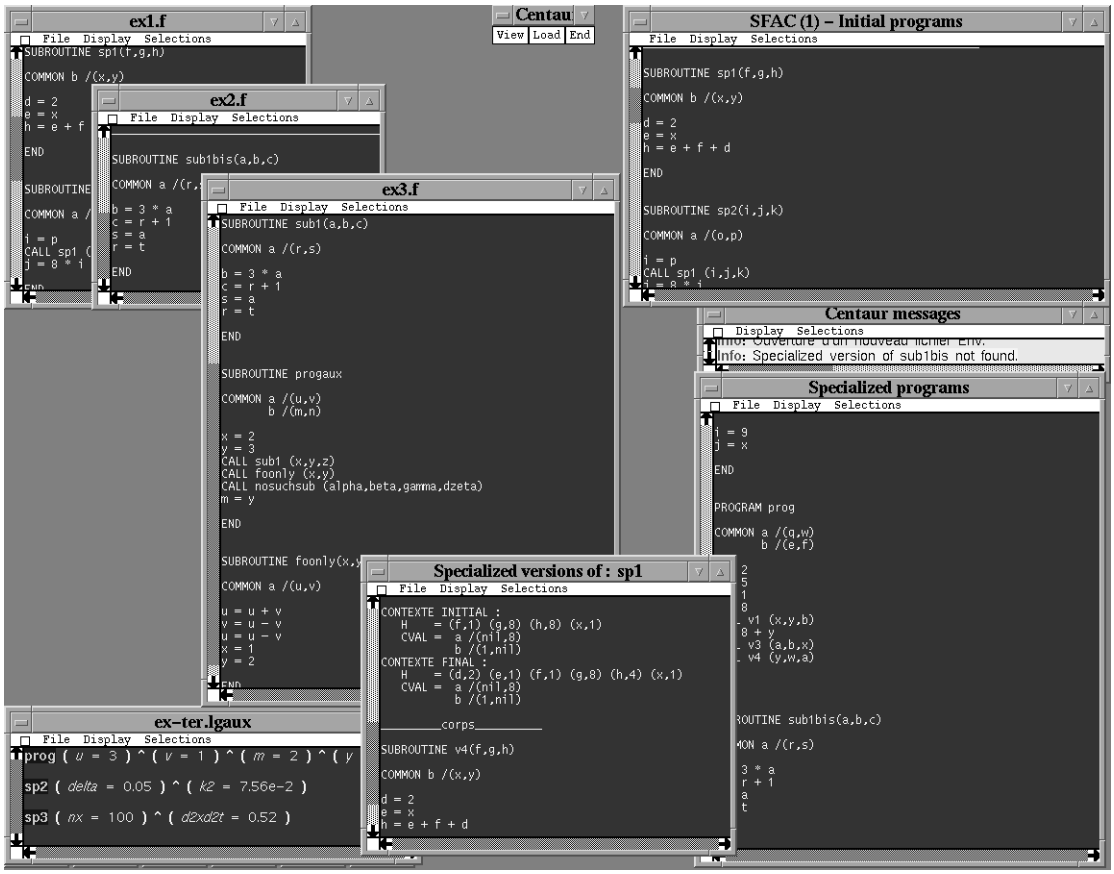

Figure 12. Partial evaluation of a Fortran application program (with reuse of specialized versions)

as shown in [4].

The abstract syntax of Fortran 90 is general and close to those of any imperative language. For instance, to be more general, our specifications assume a dereferencing operator that does not exist in Fortran 90. The only peculiarities of Fortran 90 are the parameter passing (by reference only) and the use of common blocks instead of global variables. Except the corresponding specification rules, other rules are abstract enough to be those of any other imperative language. Thus, our partial evaluation method could be easily adapted to any imperative language suitable for scientific application programs.

We have implemented a graphical interface to facilitate the exploration of Fortran application programs [15]. It has been written in Lisp, enhanced with structures for programming communication between graphical objects and processes. It is shown in Fig.12 and used as follows.

The user starts to define the application program to be specialized. For example, in Fig.12, the user has selected the files called `ex1.f`, `ex2.f` and `ex3.f`. The constraints related to this application program are called through a popup menu button (in Fig.12 they are written in the file called `ex-ter.lgaux`). When the partial evaluation is triggered, two windows are displayed. The first one (called "Initial programs" in Fig.12) displays the whole procedures to specialize. It is especially useful if some Fortran files have not been already displayed. The second window displays the whole specialized procedures.

Hyperlinks have been added to visualize with color:
- all occurrences in all displays with a special selection,
- specialized versions of a procedure,
- propagated data,
- warning messages in a special message window that will open automatically.

The user may trigger several instances of the tool together. Fig.12 shows only an instance numbered SFAC(1) (the number is written in the title of the "Initial programs" window). Each window depends on an instance and it will be killed automatically when the instance will be killed.

5. Conclusion

This paper has presented an approach to the understanding of application programs during their maintenance. The approach relies on partial evaluation, a technique that we have adapted to program understanding. The partial evaluation performs an interprocedural pointer analysis. We have formally specified our partial evaluation

process. In these specifications, inference rules in natural semantics show how statements are simplified from data propagation and simplification of other statements. A lot of data are propagated in these rules. The computations performed on these data are expressed with set and relational operators. Propagated data have been structured not to overload the rules.

From the specifications, we have implemented a tool. A graphical interface has also been implemented to visualize program dependencies (mainly between variables and values and between reused versions of procedures). The tool has been tested at EDF (the French national company that produces and distributes electricity), that provided us with scientific application programs [4]. The first results are very encouraging. We are planning more empirical work to validate these preliminary results: we intend to test other application programs made of a great deal of pointers.

We are currently investigating on an automatic proof of the soundness and correctness of the partial evaluation with respect to a dynamic semantics of Fortran. We had already proved it by hand but only for a subset of Fortran 77 and in the framework of a simple intraprocedural partial evaluation [5]. Another current focus is in improving the analysis by propagating general constraints between variables instead of only equalities between variables and values.

6. References

[1] J.R.Abrial *The B -Book Assigning programs to meanings* Cambridge University Press, 1996

[2] L.O.Andersen *Program analysis and specialization for the C programming language* Ph.D.Thesis, Univ. of Copenhagen, Denmark, DIKU rep. 94/19, 1994.

[3] R.Baier, R.Glück, R.Zöchling *Partial evaluation of numerical programs in Fortran* ACM SIGPLAN Workshop on Partial Evaluation and semantics based Program Manipulation, Melbourne, 1994, 119-132.

[4] S.Blazy, P.Facon *SFAC, a tool for program comprehension by specialization* IEEE 3rd Workshop on Program Comprehension, Washington D.C., 14-15 November 1994, 162-167.

[5] S.Blazy, P.Facon *Formal specification and prototyping of a program specializer* TAPSOFT'95, Aarhus, Denmark, May 1995, LNCS 915, 666-680.

[6] S.Blazy, P.Facon *An automatic interprocedural analysis for the understanding of scientific application programs* Dagstuhl Seminar on partial evaluation, Germany, February 1996, LNCS 1110, 1-16.

[7] *Centaur 2.0 documentation*, INRIA, 1995.

[8] N.Dubois,P.Sayarath *Aide à la compréhension et à la maintenance: pointeurs pour la spécialisation de programmes* MSc. thesis, IIE-CNAM, June 1996.

[9] A.Haraldsson *A partial evaluator and its use for compiling iterative statements in Lisp* 5th POPL, Tucson, 1978, 195-202.

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

[11] N.D.Jones, C.K.Gomard, P.Sestoft *Partial evaluation and automatic program generation* Prentice-Hall, 1993.

[12] G.Kahn, *Natural semantics* Proc. of STACS'87, LNCS, vol.247, March 1983.

[13] Komorovski *An abstract Prolog machine* Integrated interactive computing systems 1983, 183-191.

[14] S.Ramsden, F.Lin, M.A.Pettipher, G.S.Noland, J.M.Brooke *FORTRAN 90: a conversion course for Fortran 77 programmers* Manchester and North HC T&EC, 3rd eds., 1995.

[15] R.Vassallo *Ergonomie et évolution d'un outil d'aide à la compréhension de programmes* MSc. thesis, IIE-CNAM, June 1996.

[16] S.Zhang, B.Ryder, W.Landi *Program decomposition for pointer aliasing: a step toward practical analyses* 4th Symp. on the Foundations of Software Engineering, San Francisco, October 1996, 81-92.