An automatic interprocedural analysis for the understanding of scientific application programs

Sandrine Blazy, Philippe Facon

To cite this version:

Sandrine Blazy, Philippe Facon. An automatic interprocedural analysis for the understanding of scientific application programs. Dagstuhl seminar on partial evaluation, Feb 1996, Saarbrucken, Germany. pp.1-16, 10.1007/3-540-61580-6 . inria-00165927

HAL Id: inria-00165927

https://hal.inria.fr/inria-00165927

Submitted on 30 Jul 2007

An Automatic Interprocedural Analysis for the Understanding of Scientific Application Programs

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

Abstract

This paper reports on an approach for improving the understanding of old programs which have become very complex due to numerous extensions. We have adapted partial evaluation techniques for program understanding. These techniques mainly use propagation through statements and simplifications of statements.

We focus here on the automatic interprocedural analysis and we specify both tasks (propagation and simplification) for call-statements, in terms of inference rules with notations taken from the formal specification languages B and VDM.

We describe how we have implemented that interprocedural analysis in a tool, and how it can be used to improve program understanding. The difficulty of that analysis is due to the lack of well defined interprocedural mechanisms and the complexity of visibility rules in Fortran.

Keywords: software maintenance of legacy code, program understanding, program specialization, interprocedural analysis, inference rules, natural semantics, formal specification, Fortran.

1 Introduction

Older software systems are inherently difficult to understand (and to maintain). Much of the effort involved in software maintenance is in locating the relevant code fragments that implement the concepts of the application domain. The maintainer is often faced with the problem of locating specific program features or functionalities within a large and partially understood system [19].

First, there exists now a wide range of tools to support program understanding [15]. Either they transform programs given a criteria (for instance they restructure programs) or they represent programs according to various formalisms (for instance graphic formalisms showing data and control flows). Hierarchies constructed by these tools should reflect semantics [14]. Program understanding involves recognizing meaningfull entities and their dependencies: calling relationships between subroutines, data flow relationships between variables, definition relationships between variables and types. They are important for understanding the code, but many of their code is difficult to find because it is often fairly deeply embedded within the program.

Next, such tools are fully automated and not customizable. But, there will always be users who will want something else. No tool can foresee all the situations a user will encounter. Customizations, extensions and new applications inevitably become necessary. Thus, instead of supporting a non flexible builder-oriented approach, a program understanding tool should support diverse user preferences as the users'view of the code.

Last, most market tools apply to whole programs or files. For every large system, the information generated by a tool is often prodigious. Presenting the user with reams of data is insufficient. Only the knowledge of this data is important for the user. In a sense, a key to program understanding is deciding what to look and what to ignore.

Scientific programming is a good example that shows the difficulties of the program understanding task. Many scientific application programs, written in Fortran for decades, are still vital in various domains (management of nuclear power plants, of telecommunication satellites, etc.), even though more modern languages are used to implement their user interfaces. It is not unusual to spend several months to understand such application programs before being able to maintain them. For example, understanding an application program of 120,000 lines of Fortran code took nine months [9]. So, providing the maintainer with a tool, which finds parts of lost code semantics ,enables reducing this period of adaptation.

Such observations in an industrial context [9] led us to develop a sofware maintenance tool to help in understanding scientific application programs. The peculiarities of our tool are the following:

• the tool is adpated to scientific application programs. In such programs, 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). The discrepancy is increased by the widespread use of Fortran, which is old for a programming language (it lacks programming structures).

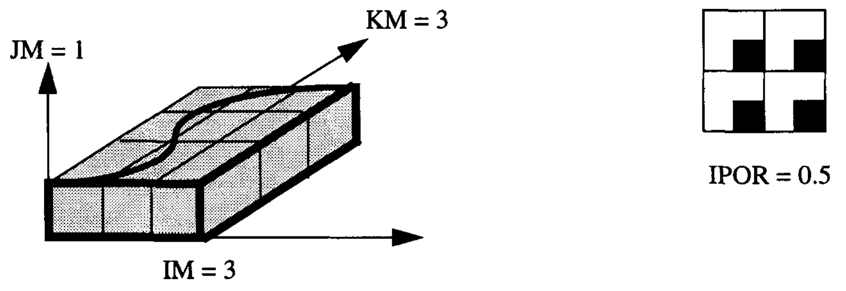
For large scientific application programs, Fortran 77 [8], which is quite an old version of the language, is used exclusively to guarantee the portability of the applications between different machines (mainframes, workstations, vector computers). The tool may analyze any Fortran program, but it simplifies only a subset of Fortran 77. This subset is a recommended standard for developping the scientific applications we have studied. The tool can not cope with `EQUIVALENCE` statements nor with any `GOTO` statement (they are not recommended), but only goto statements that implement specific control structures (e.g. a while-loop).

Scientific application programs we have studied have been developed a decade ago. During their evolution, they had to be reusable in new various contexts. For example, the same 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 program complexity, and thus amplifies the lack of structures in the 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 [5] two classes of such variables (data about geometry and control data) and we give in Fig. 1. an example of such variables.

• the tool allows the user to formulate hypotheses about the code and investigate whether they hold or must be rejected. As detailed in [17] such a process is one of the major components of dynamic code cognition behaviors. The tool helps to find parts of lost code semantics. It aims at specializing the application program according to specific values of input variables. For instance, if a team maintains only a specific application program that concerns the geometry detailed in Fig. 1, this team can focus on the application program it maintains (instead of the genaral application program that is used by several teams). Another example is the specialization of a 3D-application program into a 2D-one by setting the value of a co-ordinate.

• the tool does not change the original structure of the application program as explained in [4]. The tool is a polyvariant on-line specializer. It is based on partial evaluation but we have adapted partial evaluation for program understanding. In traditional specialization, call statements are unfolded on the fly: during specialization the call is replaced by a copy of the statements of the called procedure, where every argument expression is substituted for the corresponding formal parameter ([3], [7], [11]). This strategy aims at improving efficiency of programs.

As our goal is to facilitate the comprehension of programs, we do not change their structure (as explained in [4]) and we do not change the arity of called procedures. We do not unfold statements. Thus, the size of the code does not increase. Therefore, we are neither faced with the problem of infinite unfolding and termination of the specialization process nor with the problem of duplication of code (recursion does not exist in Fortran 77).



1	2	3
1.22	.4178	12

PRESSURE [2]

THERMODYNAMICAL_MODEL = 2

In this example we consider a liquid flow along the surface of a nuclear power plant component. This component is partitioned along the three axes, with a number of partitions of respectively *IM*, *JM* and *KM*. Moreover, the surface being porous, on a regular basis, *IPOR* is the relative side length of the solid part for each elementary cubic partition. *THERMODYNAMICAL_MODEL* is the number of the law that characterizes the liquid. We also have *PRESSURE*, with integer values that correspond, by a table, to real pressure values, each one with a specific precision.

Fig. 1. Some constraints on input data

Program slicing [18] consists of finding all statements (the slice) in a program that affect the value of a variable occurrence. Static slicing and dynamic slicing have been employed for program understanding. However for code implementing complex functionalities, they produce slices that do not help to understand the code. Other definitions of slicing have been introduced. For instance, quasi static slicing [16] combines together program slicing and partial evaluation, so allowing a better reduction of the program with respect to the slicing criterion.

We have presented in previous papers ([4]-[6]) the development and experiments of a first version of our tool. We give in Fig. 2. (see next page) an example of the program specialization performed by this version. In that first version, no interprocedural analysis was really performed: at each procedure call, the most pessimistic hypothesis about possible changes of variables values was applied and the user had to run the tool on each procedure. As most programs we analyze are large-scale Fortran programs, made of many procedures with complex interactions, that limitation was too severe. Thus we decided to extend the tool by a very precise interprocedural analysis.

Our software maintenance tool must introduce absolutely no unforeseen changes in programs. Therefore, we have first specified the specializer, then we have proven the correctness of that specification with respect to the standard semantics. [6] details this development process in a general framework. This paper describes how we have specified, implemented, and used interprocedural analysis to improve our tool for a better program understanding. Section 2 explains our interprocedural specialization strategy for Fortran programs. Section 3 gives some definitions and shows which data are needed for the specialization of procedures. Section 4 details the interprocedural specialization process. Section 5 is devoted to the implementation of that interprocedural analysis and section 6 offers conclusions and future work.

2 An Interprocedural Specialization Strategy

Fortran 77 [8] characteristics for interprocedural specialization mainly concern static side-effects. Fortran procedures may be subroutines or functions and parameters are passed by reference. In Fortran, 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 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. Variables and common blocks may be made remanent by a SAVE statement. It specifies that they retain their values between invocations. Constants may also be defined by a PARAMETER statement (for example, `PARAMETER pi=3.1416`).

Program specialization processes are based on propagation of static data. As far as we

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

Initial code

```
IREX = 1
IC = 0
IM = 20
DXLU = 0.5
```

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
         ZERO = 0
         IREGU = 1
         IDECRI = 2
         JMIN = JM
         JMAX = JM
    ENDIF
```

Specialized code

Fig. 2. An example of code specialization (without interprocedural analysis)

are concerned, these are variables, parameters or common blocks. Procedure specialization aims at specializing a procedure with respect to static data: static variables (as in specialization of other statements) and also static parameters and static common blocks. The specialization must proceed depth-first to preserve the order of side-effects [2]. This means that a called procedure must be specialized before the statements following its call.

To improve the specialization, specialized procedures and their initial and final static data are kept and reused if necessary. Thus, in a program, when several calls to the same procedure are encountered, at a given call if the set of static data and their values:

• is the *same* as the set of static data (and their values) of a previous call, then the corresponding version is directly reused,

• *strictly includes* in the set of static data (and their values) of a previous call, then the corresponding version is specialized and added to the list of specialized versions. If several versions match, the following selections are successively made: 1) version with the largest set of static data (default strategy), 2) shortest version.

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 list 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.

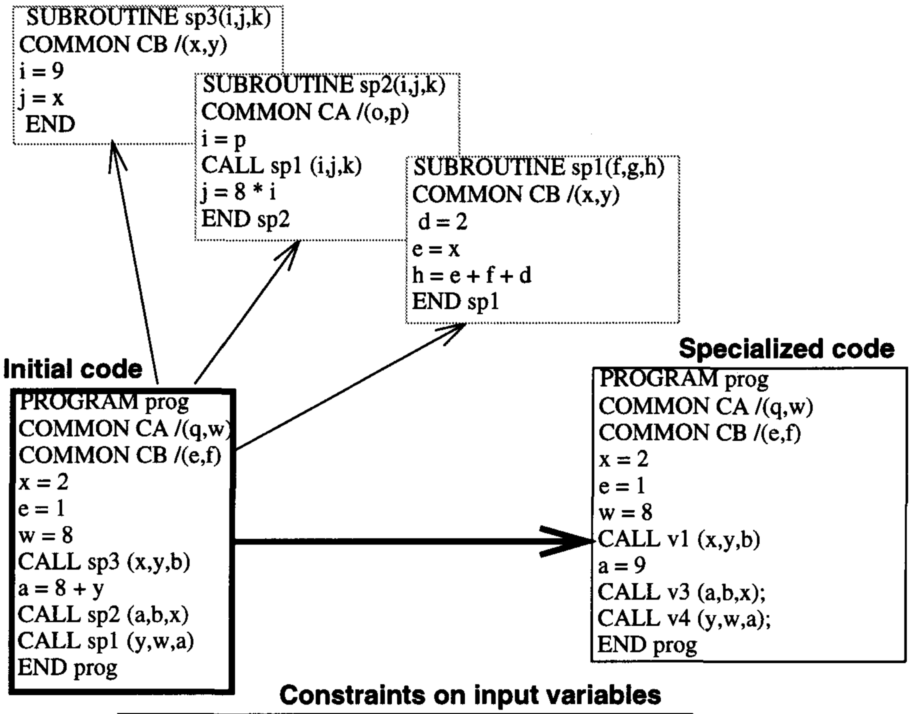
A version is characterized by its name, its statements and its initial and final static data. A version name is generated each time a new version is created. The old name is added in the caller as a comment of the call. Fig. 3 (see next page) shows an example of interprocedural specialization. In this figure, ? stands for unknown values of (dynamic) variables.

3 Notations for Specifying Interprocedural Specialization

3.1 Definitions

For a given environment, an expression is static if its subexpressions are all static. The environment is usually a function associating values to variables. For our interprocedural analysis, we need more information than such an environment to take into account side-effects due to parameters and common blocks.

We define in this section some notations, especially set operators, that are useful to specify interprocedural specialization. In these specifications we use maps associating values to identifiers. A map is a finite function. It is represented by a set of pairs of the



```
SUBROUTINE sp3(i,j,k)
COMMON CB /(x,y)
i = 9
j = x
 END
```

```
SUBROUTINE sp2(i,j,k)
COMMON CA /(o,p)
i = p
CALL sp1 (i,j,k)
j = 8 * i
END sp2
```

```
SUBROUTINE sp1(f,g,h)
COMMON CB /(x,y)
 d = 2
e = x
h = e + f + d
END sp1
```

Initial code

```
PROGRAM prog
COMMON CA /(q,w)
COMMON CB /(e,f)
x = 2
e = 1
w = 8
CALL sp3 (x,y,b)
a = 8 + y
CALL sp2 (a,b,x)
CALL sp1 (y,w,a)
END prog
```

Specialized code

```
PROGRAM prog
COMMON CA /(q,w)
COMMON CB /(e,f)
x = 2
e = 1
w = 8
CALL v1 (x,y,b)
a = 9
CALL v3 (a,b,x);
CALL v4 (y,w,a);
END prog
```

Constraints on input variables

```
 Static Variables:   m=2  u=3   v=1  y=5
Common blocks:  CA /(q=?  , w=8)   CB /(e=1 , f=?)
```

	Initial static data	Final static data	Version

v1
(sp3)
	i=2 j=5 x=1
	COMMON CA/ (?,8)
	COMMON CB/ (1, ?)
		i=9 j=1 x=1
		COMMON CA/ (?,8)
		COMMON CB/ (1, ?)
			SUBROUTINE v1 (i,j,k)
			COMMON CB /(x,y)
			i = 9
			j = **1**
			END v1

v2
(sp1)
	f=8 h=9 x=1
	COMMONCA/ (?,8)
	COMMON CB/ (1, ?)
		d=2 e=1 f=8 h=11 x=1
		COMMON CA/ (?,8)
		COMMON CB/ (1, ?)
			SUBROUTINE v2 (f,g,h)
			COMMON CB /(x,y)
			d = 2
			e = **1**
			h = **11**
			END v2

v3
(sp2)
	i=9 k=9 p=8
	COMMONCA/ (?,8)
	COMMON CB/ (1, ?)
		i=8 j=64 k=11 p=8
		COMMON CA/ (?,8)
		COMMONCB/ (1, ?)
			SUBROUTINE v3(i,j,k)
			COMMON CA /(o,p)
			i = 8
			CALL v2 (i,j,k)
			j = **64**
			END v3

v4
(sp1)
	f=1 g=8 h=8 x=1
	COMMON CA/ (?,8)
	COMMON CB/ (1, ?)
		d=2 e=1 f=1 g=8 h=4 x=1
		COMMON CA/ (?,8)
		COMMON CB/ (1, ?)
			SUBROUTINE v4 (f,g,h)
			COMMUN CB /(x,y)
			d = 2
			e = **1**
			h = **4**
			END v4

Fig. 3. An example of interprocedural specialization

form $x \rightarrow y$, where no two pairs have the same first elements. The set of all other possible values is noted *Values*. *Ident* denotes the set of all identifiers. The *Eval* function either yields the value of an expression (if it is static) or gives a residual expression (if it is dynamic).

We introduce a data type constructor named composite type (also called record) and useful set operators, similar to those defined in the formal specification languages B [1] and VDM [10]: mainly domain, $\cup$, override, various forms of restriction, and composition. These operators are written in bold in this paper.

• To create values of a composite type, a "make" function is used, called *mk-T* $(x_1, x_2, ...)$, where the x_i are the appropriate values for the fields and the result is a value of type T.

• To access fields of a composite type T, we use the record notation (for instance$T.x_i$).

• The domain (*dom*) operator applies to a map. It yields the set of the first elements of the pairs in the map.

• The union operator $\cup$ is defined only on two maps whose domains are disjoint. Thus it yields a map which contains all pairs of the maps.

• The map override operator $\dagger$ 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.

• The map restriction operator $\lhd$ (resp. $\rhd$) is defined with a first operand which is a set (resp. a map) and a second operand which is a map (resp. a set); the result is all of those pairs in the map whose first (resp. second) elements are in the set.

• When applied to a set and a map, the map deletion operator $⩤$ (resp. $⩥$) yields those pairs in the map whose first (resp. second) elements are not in the set.

• The forward composition $r ; p$ of two maps r and p is the map made of pairs $x \rightarrow z$ where there exists some y such that $x \rightarrow y \in r$ and $y \rightarrow z \in p$.

• Given two maps m, n and a set of pairs of maps s we define *Corres* (m,n) and *GalCorres* (s) such that: *Corres* $(m,n) = m^{-1} ; n$ and *GalCorres* $(m) = \bigcup \{Corres\ (x,y) \mid x \rightarrow y \in m\}$. *Corres* and *GalCorres* are relations (not necessary maps) but in this paper we use them only in contexts where they are maps. *GalCorres* (m) is only applied to maps *Corres* (x,y) with pairwise disjoint domains.

The two following examples show how both relations are used in the framework of an interprocedural analysis.

Ex1. Let $Formal = \{1 \rightarrow a,\ 2 \rightarrow b\}$ and $LParam = \{1 \rightarrow\ x+y,\ 2 \rightarrow\ 27\}$. Then, *Corres* $(Formal, LParam) = Formal^{-1} ; LParam = \{a \rightarrow x+y,\ b \rightarrow 27\}$. □

Ex2. Let $ComDecl = \{A \rightarrow \{1\rightarrow e,\ 2\rightarrow f\},\ D \rightarrow \{1\rightarrow l,\ 2\rightarrow m\}\}$ and $ComVal = \{A \rightarrow \{1\rightarrow 2,\ 2\rightarrow 4\},\ B \rightarrow \{1\rightarrow 8,\ 2\rightarrow 1\},\ D \rightarrow \{1\rightarrow 5,\ 2\rightarrow 7\}\}$. Then, *Corres* $(ComDecl, ComVal) = ComDecl^{-1} ; ComVal = \{\{1\rightarrow e, 2\rightarrow f\}\rightarrow\{1\rightarrow 2,\ 2\rightarrow 4\},\ \{1\rightarrow l,\ 2\rightarrow m\}\rightarrow\{1\rightarrow 5,\ 2\rightarrow 7\}\}$ and *GalCorres* $(ComDecl^{-1} ; ComVal) = Corres\ (\{1\rightarrow e,\ 2\rightarrow f\},\ \{1\rightarrow 2,\ 2\rightarrow 4\}) \cup\ Corres\ (\{1\rightarrow l,\ 2\rightarrow m\},\ \{1\rightarrow 5,\ 2\rightarrow 7\}) =$

$\{ e \rightarrow 2, f \rightarrow 4, l \rightarrow 5, m \rightarrow 7\}$. □

3.2 Propagated Data

During interprocedural specialization, data are propagated through statements in order to simplify them. They mainly store definitions of Fortran objects (formal parameters, common blocks, etc.) and relations between variables and values. The values are related to a program point. First, this section details these data. Then, it specifies how they are modified during the specialization of procedures.

Information associated to the current program point consist of:

- an environment (*Env*) providing information that do not change in the program. These are:
  - formal parameters. They are represented by a map *Formal* from integers (the positions in the list of formal parameters) to the names of the corresponding formal parameters. For instance, the map corresponding to the declaration `SUBROUTINE SP(a,b,c)` is $\{1 \rightarrow a, 2 \rightarrow b, 3 \rightarrow c\}$.
  - declared common blocks. They are similarly represented by a map *ComDecl* from common block names to the maps of their variable names. For instance, the map corresponding to the declarations `COMMON A/e,f` and `COMMON D/l,m` is the following map: $\{A \rightarrow \{1 \rightarrow e, 2 \rightarrow f\}, D \rightarrow \{1 \rightarrow l, 2 \rightarrow m\}\}$. The order of variable names associated to common blocks must be kept in the map because of the correspondence between the variables of a common block: the variable names of a same common block may vary from a procedure to another, and their values are passed solely thanks to the position of the variable in the declaration of the common block.
  - saved data. They are represented by a set *SavData* of variables and common block names that have become remanent after a `SAVE` statement. For instance, the set corresponding to both statements `SAVE COMMON A` and `SAVE X,Y` is $\{A,X,Y\}$.
  - initialized data. These are variables defined in a `DATA` statement and constants defined in a `PARAMETER` statement. They are represented by a map *InitData* from their names to their initialized values.
  - statements of the program (*Stmts*).
- a state (*State*) representing relations between variables and values. The type of *State* is the composite type *TState* which fields *SV* and *ComVal* are:
  - the mapping *(SV)* from Static variables to their current Values. If interprocedural analysis would not have been performed, only this map would have been used to specialize programs, as it was done in [5].
  - the mapping (*ComVal*) between common block names and the values of their static variables. For instance, in the map $ComVal = \{ A \rightarrow \{ 3 \rightarrow 2\} \}$ the common

named *A* is such that its third variable evaluates to *2* and its other variables are dynamic.

• the mapping (*Called*) between names of called procedures and pairs. Such a pair consists of:

• the environment (*EnvCalled*) of the called procedure,

• the specialized versions (*Versions*) of the called procedure. This is a set made of quadruples (*Name, Input, Output, V*), where the data type of *Input* and *Output* is *TState*, *V* denotes the whole specialized procedure and *Name* denotes the name of the procedure *V*. The **add** primitive adds a quadruple to a set of versions.

• the inherited common blocks of the current procedure *P*. They are represented by a set *ComInh* of common block names that are declared in one of the procedures of the chain of callers, but not in *P*.

4 Interprocedural Specialization

Our specializer performs two main tasks: data propagation through the code and simplification of statements. Both tasks are detailed in [4] In this section, we specify the propagation through a call-statement and the simplification of a call-statement.

4.1 The Propagation Process

While encoutering a call-statement, the propagation process propagates first the current state of the caller through the called procedure. Then, a most specialized version is selected, yielding an updated state. Last, the code generation process is returned to the calling procedure and the updated state is propagated through the caller, because of side-effects.

In the rest of this paper, *EnvSP* denotes the environment of the current called procedure, that is *EnvSP* $\underline{\Delta}$ *Called(SP).EnvCalled.* During the propagation through the called procedure formal parameters, local variables and common blocks of the called procedure *SP* modifiy the current state:

• If a local variable *V* is initialized in *SP* by a value, this value becomes the new value of *V* whether *V* had a value in the static variables *State.SV* or not. Thus, *State.SV* becomes *State.SV* † *EnvSP.InitData.*

• Due to correspondences between actual and formal parameters and also between variables of same common blocks, formal parameters and variables of declared common blocks may become static. In this case, they are added in the current state. Thus, the forward propagation updates the current state in the following way. *State.SV* becomes *Input*, with:

$$Input \ \underline{\Delta} \ (State.SV \dagger EnvSP.InitData) \cup (StaticFormal \dagger StaticCom)$$

The definitions of *StaticFormal* and *StaticCom* are explained below. As formal (resp. actual) parameters are specified by the map *Formal* (resp. *LParam*) from integers (the ranks of parameters in the list) to the names of parameters, the link between these formal and actual parameters is specified by the map *Corres* (*EnvSP.Formal, LParam*) (see Ex.1 for an example of such a map). Furthermore, only static actual parameters give values to their corresponding static formal parameters. Thus, these static formal parameters are:

$$StaticFormal \triangleq [Corres\ (EnvSP.Formal, LParam)\ ;\ eval\ (State.SV)] \rhd Values,$$

where *Corres* (*EnvSP.Formal, LParam*) ; *eval* (*State.SV*) evaluates actual parameters that are expressions, and $\rhd$ *Values* restricts this result to static formal parameters.

Given a common block *C* that is declared or inherited in the caller, if *V* is the n'th variable of *C*, then its corresponding variable *V'* in a called procedure *SP* is the n'th variable of *C*. If *C* is declared between the caller and *SP* (in the chain of callers), the names of *V* and *V'* may differ, but these variables share common values. For instance, at the program point `call SP(LParam)`, the value of *V* is the initial value of *V'*. Thus, if *V* is static, then at the entry point of *SP*, *V'* is initially static (with the same value of *V'*). This transmission of values from common blocks of the caller to corresponding common blocks of the called procedure is specified by the map *GalCorres* (*EnvSP.ComDecl*$^{-1}$; *State.ComVal*) $\rhd$ *Values*.

$$StaticCom1 \triangleq GalCorres\ (EnvSP.ComDecl^{-1}\ ;\ State.ComVal)$$

Ex.3. If the called procedure *SP* is such as $State.SV = \{i \rightarrow 5,\ j \rightarrow 3,\ a \rightarrow 1\}$ with the following declarations (without initialized data) `COMMON A/e,f,g` and `COMMON D/ l,m`, then $EnvSP.ComDecl = \{A \rightarrow \{1 \rightarrow e, 2 \rightarrow f, 3 \rightarrow g\}, D \rightarrow \{1 \rightarrow l, 2 \rightarrow m\}\}$. If for instance $ComVal = \{A \rightarrow \{1 \rightarrow 2, 2 \rightarrow 4\}, B \rightarrow \{2 \rightarrow 1\}, D \rightarrow \{1 \rightarrow 5\}\}$, then (from Ex.2) $GalCorres\ (EnvSP.ComDecl^{-1}\ ;\ ComVal) \rhd Values = \{e \rightarrow 2, f \rightarrow 4, l \rightarrow 5\}$. Thus, *State.SV* becomes $\{e \rightarrow 2, f \rightarrow 4, l \rightarrow 5, i \rightarrow 5, j \rightarrow 3, a \rightarrow 1, b \rightarrow 3\}$. End of Ex.3 □

Last, common blocks that have a scope in the caller are either declared or inherited in the caller. These are *dom* (*Env.ComDecl*) ∪ *ComInh*. They are inherited by the called procedure *SP* except if they are re-declared in this procedure. Thus, these common blocks are $ComInh' \triangleq dom\ (Env.ComDecl) \cup ComInh - dom\ (EnvSP.ComDecl)$.

The fields of the state (*State1*) resulting from the propagation through the called procedure are *Input* and *StaticCom1*, that is $State1 \triangleq mk\text{-}TState\ (Input, StaticCom1)$. After this first propagation, *State1* and the environment *EnvSP* of the called procedure are propagated through the statements of the called procedure, yielding a state *State2* (propagation rule). Then, the propagation through the caller is performed.

As in the propagation through the called procedure, actual parameters, local variables and common blocks of the caller may become static and modify the current state *State2*. In *State2*, the new values of the actual parameters (resp. common blocks) become *StaticActual* (resp. *StaticCom2*) whether they had a value in the state or not. Thus, *State2.SV* becomes $State2.SV \dagger StaticActual \dagger StaticCom$. This map is restricted to remanent variables: data saved in the called procedure are removed from the current state. Thus, the final state is:

$$\mathit{SV'} \;\Delta\; \mathit{EnvSP.SavData} \lhd (\mathit{State2.SV} \dagger \mathit{StaticActual} \dagger \mathit{StaticCom2}).$$

The definitions of *StaticActual* and *StaticCom2* are given below. In the called procedure, if a formal parameter is:

• dynamic, then in the state its corresponding actual parameter is suppressed from the static data,

• static, then its value becomes the new value of the corresponding actual parameter, whatever its previous value was.

$\mathit{Corres}\ (\mathit{LParam}, \mathit{Formal});\ \mathit{eval}\ (\mathit{State2.SV}) \rhd \mathit{Values}$ maps actual parameters to the values of their corresponding static formal parameters, as in the definition of *StaticFormal*. As expressions are not handled in static variables maps even if they are static, this map is restricted to identified actual parameters (information such as $x+y$ evaluates to *3* are lost). Thus, the static actual parameters are:

$$\mathit{StaticActual} \;\Delta\; \mathit{ident} \lhd (\mathit{SV} \dagger (\mathit{Corres}\ (\mathit{LParam}, \mathit{Formal});\ \mathit{eval}\ (\mathit{State2.SV}) \rhd \mathit{Values})).$$

In the caller, the values of some static variables of common blocks are updated in a similar way. These are variables whose value is given by the map *State2.Common*. Thus, in the caller the static variables of common blocks are:

$$\mathit{StaticCom2} \;\Delta\; \mathit{GalCorres}\ (\mathit{Env.ComDecl}^{-1}\ ;\ \mathit{State2.ComVal}).$$

Last, in the called procedure, remanent common blocks (*RemCom*) have became remanent by a SAVE statement of the called procedure (they belong to *EnvSP.SavData*), or they exist in the called procedure (either they have been declared in it or they have been inherited from the caller). Thus:

$$\mathit{RemCom} \;\Delta\; \mathit{dom}\ (\mathit{EnvSP.ComDecl}) \cup \mathit{ComInh'} \cup \mathit{EnvSP.SavData}.$$

State2.ComVal maps static variables of common blocks to their corresponding values. These are common blocks of the caller if remanent common blocks of the called procedure have been removed from the map. Thus, *Env.ComVal* becomes *ComVal'* with $\mathit{ComVal'} \;\Delta\; \mathit{RemCom} \lhd \mathit{State2.ComVal}$.

The fields of the final state *State'* are *SV'* and *ComVal*, that is $\mathit{State'} \;\Delta\; \mathit{mk\text{-}TState}\ (\mathit{SV'}, \mathit{ComVal'})$.

Fig. 4. (see next page) recalls whole definitions and the corresponding propagation rule explained in this section. While implementing such rules, variables should be replaced in the rules by their definition.

4.2 Simplification

Recall the interprocedural specialization strategy: while simplifying a call-statement, the specializer checks first whether the called procedure has been specialized in a similar context (that is with the same or less restrictive static data) before. Three situations may happen. They correspond to the three rules of Fig. 5. and are:

• the called procedure *SP* has already been specialized into *V* with the same static data

Definitions

EnvSP $\triangleq$ *Called(SP).EnvCalled*

Caller → called propagation

StaticFormal $\triangleq$ *Corres* (*EnvSP.Formal*, *LParam*) ; *eval* (*State.SV*) $\rhd$ *Values*

StaticCom1 $\triangleq$ *GalCorres* (*EnvSP.ComDecl*$^{-1}$; *State.ComVal*)

Input $\triangleq$ (*State.SV* † *EnvSP. InitData*) $\cup$ (*StaticFormal* † *StaticCom1*)

ComInh' $\triangleq$ *dom* (*Env.ComDecl*) $\cup$ *ComInh* − *dom* (*EnvSP.ComDecl*)

State1 $\triangleq$ *mk-TState* (*Input*, *StaticCom1*)

Called → caller propagation

StaticActual$\triangleq$ *ident*$\lhd$ [*Corres* (*LParam*, *EnvSP.Formal*); *eval* (*State2.SV*)]$\rhd$*Values*

StaticCom2 $\triangleq$ *GalCorres* (*Env.ComDecl*$^{-1}$; *State2.ComVal*)

SV' $\triangleq$ *EnvSP.SavData* $\lhd$ (*State2.SV* † *StaticActual* † *StaticCom2*)

RemCom $\triangleq$ *dom* (*EnvSP.ComDecl*) $\cup$ *ComInh'* $\cup$ *EnvSP.SavData*

ComVal' $\triangleq$ *RemCom* $\lhd$ *State2.ComVal*

State' $\triangleq$ *mk-TState* (*SV'*, *ComVal'*)

Propagation rule

$$\frac{EnvSP.Env,\ State1,\ ComInh',\ Called\ \vdash\ EnvSP.Stmts : State2}{Env,\ State,\ ComInh,\ Called\ \vdash\ \mathbf{call}\ SP\ (LParam) : State'}$$

Fig. 4. Propagation of call-statements

(*Input*). Thus, *V* becomes the specialized procedure of the subject procedure *SP* (first rule).

• the procedure is not as specialized as wanted (second rule). For instance the last parameter of the procedure to specialize is static, but the last parameter of the most specialized version is dynamic. Then, one of the most specialized versions (*Version*) and its output static data (*Output*) are selected among the versions such that the cardinal of their input static data is the biggest. Note that its static variables and parameters are strictly included in those of the subject procedure, and its common blocks, that have also a scope in the subject procedure, are such that each of their static variables has the same value in both common blocks. Thus, *Version* and *Output* are:

$$\textit{Version, Output} \triangleq \text{any } V, O \text{ such that } (N, In, O, V) \in \textit{Called(SP).Versions} \wedge$$
$$\textit{Input} \subset \textit{In} \wedge (\nexists(N', J, O, V') \in \textit{Called(SP).Versions} \text{ such that } \textit{Input} \subset J \subseteq \textit{In})$$

The selected procedure is then specialized (as in the following situation) and the name of the specialized procedure (*NewName*) is selected among the set **NAME** of possible names, that have not been already selected as procedure names.

• the called procedure has not been specialized. A residual procedure is specialized from it, a new name is computed and the current state is modified by the results of the propagation (third rule).

5 Implementation

We have implemented our specification rules as such in our tool where B and VDM operators have been translated into Prolog. This process is the same as those followed in [5] and [13]. The tool is fully automated. A debugger enables apply the rules to be applied step by step. This is useful while trying to understand the behavior of a residual program. For instance, with such a debugger, the user may know at any program point the static values and then understand either why a then-branch of an alternative has been removed and not the else-branch, as he would have thought, or why a function does not yields the expected result.

We have used natural semantics to write our specification rules. Natural semantics gives a formalism to write inference rules, but it does not provide any formalism to describe (and decompose) the environments appearing in the rules. Without interprocedural analysis, such an environment was a single SV-like map. This map was thus a variable in the inference rules. But with an interprocedural analysis, given the environments we propagate, we can not afford to show to the user all the environment variables appearing in the rules.

Environments are pretty-printed in a user-friendly style (which is close to Fortran). Instead of showing directly the mathematical variables that are used in environments, only information that are relevant for the user (e.g. information related to common blocks) are visualized. For instance COMMON A / e = 2, f = u is pretty-printed instead of $\textit{Env.ComDecl} = \{A \rightarrow \{1 \rightarrow e, 2 \rightarrow f\}\}$, $\textit{State.ComVal} = \{ A \rightarrow \{ 1 \rightarrow 2 , 2 \rightarrow u \} \}$.

6 Conclusion

We have explained how to extend by interprocedural analysis our program specialization technique for program understanding. That analysis is especially difficult in Fortran, due to the lack of well defined interprocedural mechanisms and the complexity of visibility rules. Therefore we have designed these extensions very carefully, starting with a formal specification of the information to be computed. These

Definitions

Version, *Output* Δ any V, O such that $(N, In, O, V) \in Called(SP).Versions \wedge$

$Input \subset In \wedge (\nexists (N', J, O, V') \in Called(SP).Versions$ such that $Input \subset J \subset In)$

$SelectedNames \, \Delta \, \{N \mid (N, I, O, V) \in \textbf{ran}\,(Called).Versions\}$

Simplification rules

Rule 1: the called pocedure has already been specialized with the same static data

$$\frac{\begin{array}{c} (Name, State1.SV, Out, V) \in Called(SP).Versions \\ EnvSP, Output, ComInh', Called \vdash^{Propagation} V : State2 \end{array}}{Env, State, ComInh, Called \vdash \textbf{call}\ SP\ (LParam) \rightarrow \textbf{call}\ NewName\ (LParam)} \quad (1)$$

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

$$\frac{\begin{array}{c} \{(Name, In, Out, V) \in Called(SP).Versions \mid State1.SV \subseteq In\} \neq \emptyset \\ EnvSP, Output, ComInh', Called \vdash Version \rightarrow SP' \\ EnvSP, Output, ComInh', Called \vdash^{Propagation} SP' : State2 \\ \textbf{add}\ ((NewName, State.Variables, State2.Variables, SP'), Called(SP).Versions) \\ NewName \in \textbf{NAME} - SelectedNames \end{array}}{Env, State, ComInh, Called \vdash \textbf{call}\ SP\ (LParam) \rightarrow \textbf{call}\ NewName\ (LParam)} \quad (2)$$

Rule 3: the called procedure has not been specialized

$$\frac{\begin{array}{c} \{(Name, In, Out, V) \in Called(SP).Versions \mid State1.SV \subseteq In\} = \emptyset \\ EnvSP, State1, ComInh', Called \vdash EnvSP.Stmts \rightarrow SP' \\ EnvSP, State1, ComInh', Called \vdash^{Propagation} SP' : State2 \\ \textbf{add}\ ((NewName, State.Variables, State2.Variables, SP'), Called(SP).Versions) \\ NewName \in \textbf{NAME} - SelectedNames \end{array}}{Env, State, ComInh, Called \vdash \textbf{call}\ SP\ (LParam) \rightarrow \textbf{call}\ NewName\ (LParam)} \quad (3)$$

Fig. 5. Simplification of call-statements

extensions are now integrated in our tool and they allow us to specialize complex programs with much greater precision than previously. Information is propagated along procedure calls, and specialized versions of the called procedures are proposed to the maintainer.

The first experiments with these extensions have given very satisfactory results. Some improvements are now under consideration. For instance a good strategy for reuse of procedure specializations must be developed. Furthermore, the information we compute for interprocedural specialization are of great interest by themselves, in the program comprehension process, independently of its particular use in specialization. Therefore we develop ways to show it in a user-friendly shape.

References

1. J.R.Abrial *The B method* CNAM lecture notes, 1994
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 PEPM, Melbourne, 1994, 119-132.
4. S.Blazy, P.Facon *Partial evaluation as an aid to the comprehension of Fortran programs* 2nd IEEE Workshop on Program Comprehension, Capri, Italy, July 1993, 46-54.
5. S.Blazy, P.Facon *SFAC: a tool for program comprehension by specialization* 3rd IEEE Workshop on Program Comprehension, Washington D.C., November 1994, 162-167.
6. S.Blazy, P.Facon *Formal specification and prototyping of a program specializer* TAPSOFT'95, Aarhus, Denmark, May 1995, LNCS 915, 666-680.
7. Coen-Porisini, F.De Paoli, C.Ghezzi, D.Mandrioli *Software specializaton via symbolic execution* IEEE Trans. on Soft. Engineering, 17(9), 1991, 884-899.
8. *FORTRAN.* ANSI standard X3.9, 1978.
9. M.Haziza, J.F.Voidrot, E.Minor, L.Pofelski, S.Blazy *Software maintenance: an analysis of industrial needs and constraints* IEEE Conf. on Soft. Maintenance, Orlando, 11/1992.
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 1987.
13. H.Parisot, F.Paumier *Aide à la compréhension et à la maintenance du logiciel: calculs inter-procéduraux pour la maintenance de programmes* MSc. thesis, IIE-CNAM, 06/95.
14. S.Tilley, K.Wong, M.A.Storey, H.Müller *Programmable reverse engineering* Int. Journal of Soft.Eng.&Knowledge Eng., 4(4), December 1994, 501-520.
15. H.J.Van Zuylen *Understanding in reverse engineering.* REDO handbook, Wiley, 09/92.
16. G.A.Venkatesh *The semantic approach to program slicing* Proc. ACM SIGPLAN Conf. on Programming Languages Design and Implementation, Toronto, Canada, 1991, ACM SIGPLAN Notices, 26(6), 107-119.
17. A.Von Maryhauser, A.M.Vans *Dynamic code cognition behaviors for large scale code* 3rd IEEE Workshop on Program Comprehension, Washington D.C., November 1994, 74-81.
18. M.Weiser *Program slicing* IEEE Trans. on Soft. Engineering, 10(4), 352-357, 1984.
19. N.Wilde, M.C.Scully *Software reconnaissance: mapping program features to code* Software maintenance research and practice, vol.7, 1995, 49-62.