Specifying and automatically generating a specialization tool for Fortran 90

Sandrine Blazy

Specifying and automatically generating a specialization tool for Fortran 90

SANDRINE BLAZY blazy@iie.cnam.fr
CEDRIC IIE, 18 allee Jean Rostand, 91 025 Evry Cedex, France

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 a dynamic semantics of Fortran 90 and a partial evaluation process, both with various formalisms (inference rules with global definitions and set and relational operators) and how the partial evaluation has been manually derived from the dynamic semantics. The guidelines for proving the correctness of the partial evaluation with respect to the dynamic semantics are introduced in this paper. The tool implementing the specifications is also detailed. 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, etc.).

Keywords: program understanding, partial evaluation, dynamic semantics, formal specification, interprocedural analysis, alias analysis, proof of correctness

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.* shared mechanisms of common blocks 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.

Our tool aims at improving the understanding of scientific application programs. These application programs are difficult to maintain mainly because they were 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. This context is very well known by all the people belonging to the maintenance team; this is their minimum knowledge concerning data of their application program. This context is described by equalities between specific variables and values. Such variables and their values are very well known by the maintenance team [23].

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. In partial evaluation, the only constraints are the values of some input data. 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 [13]. Partial evaluation has been applied to generate compilers from interpreters (by partially evaluating the interpreter for a given program). In this context, previous work ([15, 1]) has primarily dealt with functional and declarative languages. Partial evaluation has also been applied to improve speedups of imperative programs ([11, 18, 16]). 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 ([11, 16]). Statement 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 and does not generate residual programs that are easier to understand. Thus, our residual programs are not as efficient as they could have been if we have used statement unfolding and other more sophisticated partial evaluation techniques (*e.g.* binding-time analysis). In like manner, our partial evaluator neither generates 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 [20] for details about our specialization strategy).

Fig. 1 briefly illustrates how initial code is specialized, with respect to constraints on input variables. The initial code that has been removed is striked out (e.g. in fig. 1, the first if statement is removed and replaced by its specialized then-branch). The initial code that is not striked out corresponds to the specialized code. In the specialized code, expressions are also simplified. This code is an executable code. In initial and specialized codes, simplified expressions are underwaved. This

```
SUBROUTINE INIG(X,DX,IDEC,DXL,ZMIN)
COMMON/GEO1/IM,JM,KM,KMM1,IMAT
COMMON/GEO2/INDX_I,INDX_J,INDX_K
IF(IREX.NE.0)THEN
      IF (DXL.EQ.0) THEN WRITE(NFIG12,1001)DXL
      ENDIF
      IF(KM.EQ.0)THEN READ(NFIG11,*,ERR=1102)ZMIN
      ELSE ZMIN=0. ENDIF
      X=ZMIN+FLOAT*DXL; KMM1=KM-1
ELSE READ(NFIG11,*,ERR=1103)X ENDIF
IF (IMAT.EQ.0.AND.KM.GE.10) THEN
      CALL VALMEN(MAT,KMM1,-1)
      IF(IREX.EQ.0)THEN WHAT='I' ELSE
            CALL VALMEN(MAT,KM,3)
            IF(IDESCREG.EQ.0)THEN
                  IREG=0
            ELSE IREG=1; IDEC=IDESCREG
            ENDIF
            IF(IDEC.EQ.0.AND.IREG.EQ.0)
                  THEN WHAT='I'; IDEC=3
            ELSE IF(INDX_J.NE.0)THEN
                  WHAT='K'
            ELSEIF(INDX_J.NE.0)THEN
                  WHAT='J'
            ELSEIF(INDX_K.NE.0)THEN
                  WHAT='K'
            ELSE CALL STOP('INIG')
            ENDIF
      ENDIF
      IF(IDEC.EQ.1)THEN
            IF(IREG.EQ.0)THEN IMIN=2; IMAX=IM
            ELSE IMIN=IM; IMAX=IM ENDIF
      ELSEIF(IDEC.EQ.2)THEN
            IF(IREG.EQ.0)THEN JMIN=2; JMAX=JM ENDIF
      ELSEIF(IDEC.EQ.3)THEN
            IF(IREG.EQ.0)THEN KMIN=2; KMAX=KM
            ELSE KMIN=KM; KMAX=KM+1 ENDIF
      ENDIF
ENDIF; END
```

Initial code

```
IREX=1
IDESCREG=3
INDX_I=2
DXL=0.5
KM=20
```

Constraints on input variables

```
SUBROUTINE INIG(X,DX,IDEC,DXL,ZMIN)
COMMON/GEO1/IM,JM,KM,KMM1,IMAT
COMMON/GEO2/INDX_I,INDX_J,INDX_K
ZMIN= 0.
X= FLOAT * 0.5
KMM1= 19
IF ( IMAT .EQ. 0) THEN
      CALL VALMEN_V1(MAT,19,-1)
C SPECIALIZED VERSION OF VALMEN WITH...
      CALL VALMEN_V2 (MAT,20,3)
C SPECIALIZED VERSION OF VALMEN WITH...
      IREG= 1; IDEC= 3
      WHAT= 'K'
      KMIN=20; KMAX= 21
ENDIF
END
```

Specialized code

Figure 1. An example of program specialization.

formatting of the program text is automatically done by our tool. Usually, a color is associated to each format. 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). They are not shown in fig. 1 to avoid overloading it.

At the very beginning, our tool implemented a simple intraprocedural analysis. It simplified assignments, alternatives and loops [21]. The formal specification consisted only of inference rules in natural semantics operating on abstract syntax trees [4]. These rules were quite easy to understand: they were made of sequents defining inductively a propagation relation called *propag* ($\mathsf{S1} \overset{propag}{\vdash} \mathsf{I} : \mathsf{S2}$ means that the execution of the statement I modifies the initial state $\mathsf{S1}$ into the final state $\mathsf{S2}$), a simplification relation called *simpl* ($\mathsf{S} \overset{simpl}{\vdash} \mathsf{I1} \hookrightarrow \mathsf{I2}$ means that given the state S, the statement $\mathsf{I1}$ simplifies into $\mathsf{I2}$), and the combination of both for defining a partial evaluation relation called PE ($\mathsf{S1} \overset{PE}{\vdash} \mathsf{I1} \mapsto \mathsf{I2}, \mathsf{S2}$ means that given the state $\mathsf{S1}$, the specialization of the statement $\mathsf{I1}$ yields a simplified statement $\mathsf{I2}$ and a new state $\mathsf{S2}$). These sequents consist of variables (*e.g.* S or $\mathsf{I1}$- usually they are capitalized), symbols representing relations (*e.g.* $\mapsto$, $\hookrightarrow$ and :) and nodes of abstract syntax trees (*e.g.* $\mathsf{I1}$ and $\mathsf{I2}$). These nodes will be written in bold in this paper.

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 (*i.e.* the hypotheses of the sequents) 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 [21].

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. We have also specified 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.

We have specified and implemented a context-sensitive and flow-sensitive general alias analysis. It is more sensitive than other alias analysis that are more efficient [17, 24, 12], but as our partial evaluation propagates only equalities between variables and values, less data are propagated during the interprocedural alias analysis. Our analysis does not support recursive calls (the application programs we have analyzed are not recursive) but it handles return constants. The framework of our

analysis is similar to the more general one described in [3]: for each procedure, information that describe the effects of that procedure are propagated through the call graph. This graph represents the structure of the analyzed program. In this paper, we will also specify analyzed programs.

Natural semantics rules are useful to show how semantic 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 composition links between these data by object diagrams. The diagrams show some variables of the rules and other variables that are defined outside the rules to avoid overloading them.

The aim of this paper is threefold: to show how we have specified, proven and implemented these extensions to our partial evaluator. Compared to our previous work ([20, 21, 22]):

- Our program analyses take into account an interprocedural alias analysis.
- We have implemented a graphical interface.
- Before specifying natural semantics rules, we have defined object diagrams for structuring modeled data.
- We have adapted our specifications to allow local definitions.

The rest of this paper is structured as follows. First, Section 2 recalls some concepts of Fortran 90 and explains our specialization strategy. Then, section 3 details how we specify programs and the information computed during their analyses. It gives a dynamic semantics of call statements and pointer variables, and specify the interprocedural pointer analysis of the partial evaluation. It shows how the propagation relation has been derived from the dynamic semantics relation. Section 4 presents the general framework for proving the correctness of the partial evaluation and gives the proof steps for an example of call statements. Section 5 is devoted to the implementation of our tool.

2. Background

This section introduces some concepts of Fortran 90: procedures, common blocks, structures, pointers and targets. Then, it explains how procedures are specialized and reused during the specialization process. Reused procedures are also modeled in an object diagram.

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. Although the names of the common blocks themselves are global, none of the variables within the common blocks are global. 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.

In Fortran 90 a structure consisting of a list of fields, each of some particular type, is a type. The field's 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 [17]. There is no pointer arithmetic in Fortran 90. A pointer can not point to a pointer. 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.* $*(p \rightarrow next)$ in fig. 2 and fig. 3).

Fig. 2 shows an example of a Fortran 90 code, where a new type called `node` is defined. It will be constructed from two values representing an `ident` and a pointer to the `next` field in a linked list. Once two variables p and q of type pointer to `node` have been declared, the values 3.4 and 6.2 are inserted in the list of `nodes` in that order.

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

Figure 2. An example of Fortran 90 code.

An 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 of a procedure 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

Abstract syntax

$VarName \subseteq Lhs$

$\mathbf{deref} : Lhs \rightarrow Lhs$

$\mathbf{field\_lhs} : Lhs \times VarName \rightarrow Lhs$

Examples

concrete syntax	abstract syntax
v	$VarName(v)$
$person\%address\%town$	$\mathbf{field\_lhs}(\mathbf{field\_lhs}(person, address), town)$
$*(p \rightarrow next)$	$\mathbf{deref}(\mathbf{field\_lhs}(\mathbf{deref}(p), next))$

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

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 (its input static 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).

The diagram of fig. 4 models specialized versions. A procedure SP belongs to the object called Procedure. SP represents the code of the whole procedure (its declarations and its statements). The specialized versions of SP belong to the object called Version. They are represented by the set *version*(SP). A specialized version v of SP consists of a name for this version (*name_version*(v)), input static data (*input*(v)), output static data (*output*(v)) and statements (*stmt_version*(v)). This is equivalent to saying that a version is represented by a quintuplet (name of original procedure, version name, input data, output data, statements). The version and its corresponding procedure have the same arities. Thus, the formal parameters of the version are those of the procedure.

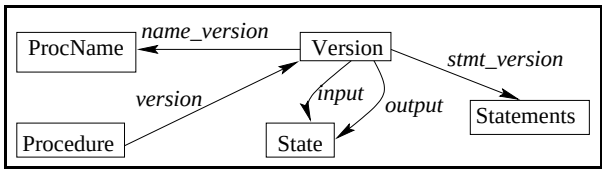


Figure 4. Object diagram modeling specialized versions.

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

- is the same as those of a previous call, then the corresponding version is directly reused,
- strictly includes those of a previous call, then the corresponding version is specialized yielding a new version that is 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 input static data,
  - shortest version among most specialized versions.

3. Formal Specification of the Partial Evaluation

This section shows in object diagrams data that are propagated during execution and partial evaluation. It also introduces set operators that apply on these data. Then, this section details the specification of the dynamic semantics of two kinds of statements: call statements and assignments between pointers. The dynamic semantics of assignments between targets and pointers is specified in a similar way. It has not been presented in this paper. Lastly, the propagation and simplification relations (introduced in section 1) are derived from the dynamic semantics relation.

3.1. Program Representation

We have specified with inference rules (as in [21]) both the dynamic semantics and the partial evaluation, but more data are propagated in the inference rules. In our specifications, a Fortran 90 procedure is represented by:

- its environment (called Env in fig. 5), that represents what does not vary during the analysis of the procedure (formal parameters, common blocks variables that have a scope in the current procedure, local variables and statements),
- its state (called State in fig. 5) modeling relations between variables and known values at the current program point. An object State (called S) consists of two objects: $val(\mathsf{S})$ and $comVal(\mathsf{S})$. $val(\mathsf{S})$ maps variables to values (if these values are known) and $comVal(\mathsf{S})$ maps common blocks to the known values of their variables. Since in common blocks, values are shared between two variables simply by the fact that they occupy corresponding positions within the same common block, these values are not modeled as values of other variables. In fig. 5 these values are represented by the object called "Value of common blocks variables" that is accessed from State through the $comVal$ function. An instance of this object denotes the values (and their corresponding positions) of variables belonging to common blocks.

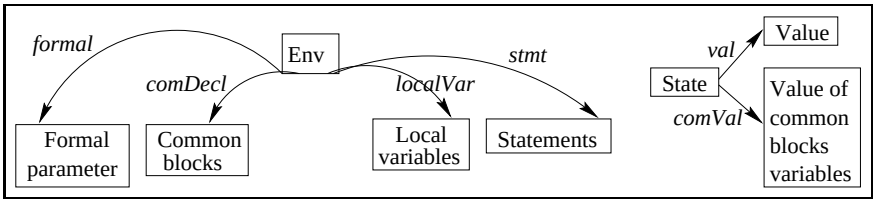


Figure 5. Some propagated data: the state and the environment of a procedure.

[22] gives examples of these data and fig. 6 models the whole data that represent procedures. The diagram of fig. 6 represents objects and access functions between them. It extends the diagram of fig. 5. The object called Procedure denotes a called procedure SP at a given program point. From SP, we have access to its environment *environment*(SP), its state *values*(SP), its inherited common blocks *inherits*(SP) and its name *name*(SP). This name belongs to ProcName, that denotes actual procedure names (including names of versions) that are used in the current application program.

An instance of the diagram of fig. 6 represents information for all procedures that are called from a main program (or from one of its called procedures). It shows the bindings between procedure names and their corresponding statements, and it results from a preliminary analysis of the program. Such an instance is an implicit parameter of the dynamic semantics and partial evaluation systems. The objects of the instance are never modified by the sequents. Thus, the instance is an implicit parameter, to avoid overloading the sequents.

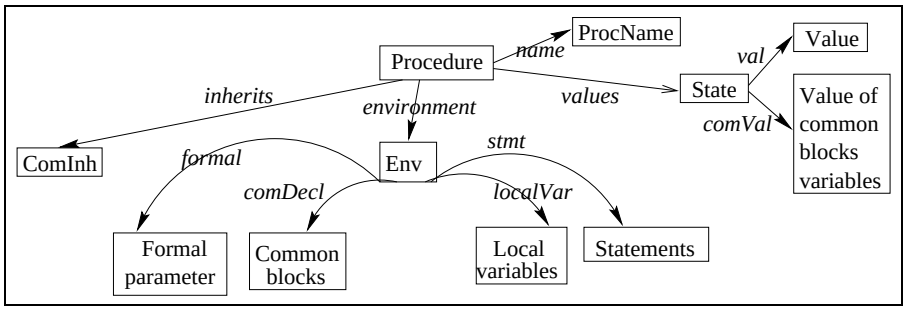


Figure 6. Object diagram showing data propagated during an interprocedural analysis.

3.2. Definitions

We define in this section some notations, especially set operators, that we use in our specifications. $\mathcal{PROCNAME}$ denotes the set of possible names of procedures, thus ProcName is a subset of the set $\mathcal{PROCNAME}$. $\mathcal{VALUE}$ 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 [20]. We introduce useful set operators, similar to those defined in the formal specification languages B [10] and VDM [2]: mainly inverse ($^{-1}$), domain (**dom**), range (**ran**), union ($\cup$), override ($\dagger$), restrictions ($\rhd$, $\lhd$ and $\ntriangleleft$), relation composition ($;$ ^{1}) and direct product ($\otimes$). In the following definitions s denotes a set, r, p and q denote binary relations, m and n denote maps (specific binary relations where each element is related to at most one element). A binary relation is a set of pairs. Thus, classical set operators such as union can also be applied to binary relations.

- $r^{-1} = \{x \mapsto y \mid y \mapsto x \in r\}$
- $\mathbf{dom}(r) = \{x \mid x \mapsto y \in r\}$
- $\mathbf{ran}(r) = \{y \mid x \mapsto y \in r\}$
- $r \dagger p = \{x \mapsto y \mid x \mapsto y \in p \bigvee (x \mapsto y \in r \bigwedge x \notin \mathbf{dom}(p))\}$
- $r \rhd s = \{x \mapsto y \in r \mid y \in s\}$
- $s \lhd r = \{x \mapsto y \in r \mid x \in s\}$
- $s \ntriangleleft r = \{x \mapsto y \in r \mid x \notin s\}$
- $r;p = \{x \mapsto z \mid \exists y \cdot x \mapsto y \in r \bigwedge y \mapsto z \in p\}$
- $r \otimes p = \{x \mapsto (y,z) \mid x \mapsto y \in r \bigwedge x \mapsto z \in p\}$

  When r and p are functions, their direct product is specially interesting when it is an injective function. In this case, when a pair of the form $x \mapsto (y,z)$ belongs to $r \otimes p$ then (y,z) determines x uniquely, and x may be written $(r \otimes p)^{-1}(y,z)$.
- $r \otimes p \otimes q = \{x \mapsto (y,z,t) \mid x \mapsto y \in r \bigwedge x \mapsto z \in p \bigwedge x \mapsto t \in q\}$

  In like manner, when $r \otimes p \otimes q$ is an injective function, x may be written $(r \otimes p \otimes q)^{-1}(y,z,t)$.
- Given s a set of pairs of maps, we define:

  $$\mathbf{Corres}(s) = \bigcup\{m^{-1};n \mid m \mapsto n \in s\}.$$

  We use $\mathbf{Corres}(s)$ to bind variable names of a common block to their corresponding values. 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, each pair $m \mapsto n$ of s corresponds to a common block. m and n map integers (the position in the list of declared variables of the common block) to respectively variable names and values. $m^{-1};n$ is then a map from variable names to values and $\mathbf{Corres}(s)$ is the union of all such maps.

  In the following example, the declaration of two common blocks `B` and `C` is represented by the map $ComDecl$. At the current program point, the values of variables belonging to common blocks are given by the map $ComVal$. $ComVal$ states that:

- the value of the first variable of `B` is 0 and the value of the other variable of `B` is unknown (this variable is not represented in the map $ComVal$),
- the value of the first (resp. third) variable of `C` is 0.5 (resp. 6.2), the value of the second variable of `C` is unknown.

In the current procedure, the variables of `B` (resp. `C`) are `t` and `u` (resp. `x`, `y` and `z`). Thus, at the current program point, the map from these variables to their values is $\mathbf{Corres}(s)$.

EXAMPLE:
```
COMMON B / t, u
COMMON C / x, y, z
```

$$ComDecl = \{B \mapsto \{1 \mapsto t, 2 \mapsto u\}, C \mapsto \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\}\}$$
$$ComVal = \{B \mapsto \{1 \mapsto 0\}, C \mapsto \{1 \mapsto 0.5, 3 \mapsto 6.2\}\}$$
$$\text{If } s = ComDecl^{-1}; ComVal \qquad \text{then}$$

$$s = \{\{1 \mapsto t, 2 \mapsto u\} \mapsto \{1 \mapsto 0\}, \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\} \mapsto \{1 \mapsto 0.5, 3 \mapsto 6.2\}\}$$
$$\text{and} \qquad \mathbf{Corres}(s) = \{t \mapsto 0, x \mapsto 0.5, z \mapsto 6.2\}$$

□

3.3. Dynamic Semantics of Call Statements

The dynamic semantics of Fortran 90 is deterministic. It is formalized by the *sem* system, that generates sequents of the form $\mathsf{E}, \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \mathsf{I} : \mathsf{S}'$, meaning that given an environment E, a state S and inherited common blocks CI, the execution of statement I leads to the values given by the state S'. E, S and CI are the hypotheses of the sequent.

Fig. 7 shows the dynamic semantics rule for a call statement `CALL SP(LParam)`, given an environment E, a state S and inherited common blocks CI. SP is the name of the called procedure and LParam is a map from positions (in the list of parameters) to the actual parameters of SP at the current program point. In the definitions part of the figure, some definitions are factorized. They introduce some useful local variables appearing in the dynamic semantics rule. These variables are EnvSP, $\mathsf{S1}$, CI' and S'. Definitions are here "macros" that are supposed to be applied to the rules containing the variables. They are given in the definition part of fig. 7, to avoid overloading the dynamic semantics rule.

The first definition introduces EnvSP, the environment of the procedure named SP. $name^{-1}$ yields an object Procedure from an object ProcName (see fig. 6). Thus, since SP is a procedure name, $name^{-1}(\mathsf{SP})$ represents the procedure whose name is SP and EnvSP is defined as the environment of this procedure.

The beginning of fig. 7 illustrates through an example which data are propagated during the execution of statements (with respect to the dynamic semantics rule

of the figure) and which are the corresponding program points (α, β, γ and δ in the figure). Classically, the execution of the call statement starts with a forward propagation through the call graph (from program point α to program point γ) followed by a backward propagation (from program point δ to program point β). The forward propagation aims at giving new values to formal parameters (referenced as VFormal in fig. 7) and common blocks variables of the called procedure SP (VComCalled). These values are $val(\mathsf{S})$ (values of variables) and $comVal(\mathsf{S})$ (values of common blocks).

3.3.1. Computation of VFormal (values of formal parameters at program point γ) Formal (resp. actual) parameters are represented by maps between positions and names (resp. values) of the parameters. In fig. 7, these maps are respectively $formal(\mathsf{EnvSP})$ and LParam. Thus:

- $(formal(\mathsf{EnvSP}))^{-1}$ maps the names of the formal parameters of SP to their positions,
- $(formal(\mathsf{EnvSP}))^{-1}; \mathsf{LParam}$ maps the names of the formal parameters of SP to the actual parameters of the current call statement,
- ^{2} $\mathsf{VFormal} \triangleq (formal(\mathsf{EnvSP}))^{-1}; \mathsf{LParam}; val(\mathsf{S})$ maps the names of the formal parameters to their initial values, that are the values of actual parameters at the program point α.

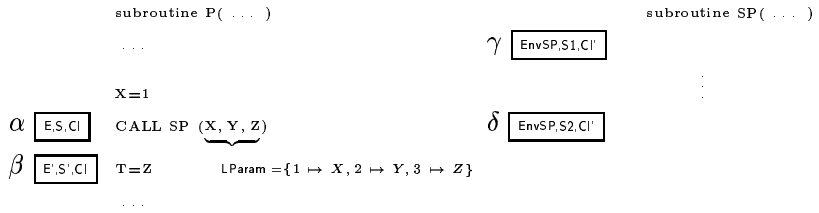
3.3.2. Computation of VComCalled (values of common blocks at program point γ) Variables belonging to a common block are also represented by maps between positions in the common block and variable names. The values of variables belonging to common blocks of the calling procedure are also transmitted to the corresponding variables (they share a same position in the common block). Thus, $\mathsf{VComCalled} \triangleq \mathbf{Corres}((comDecl(\mathsf{EnvSP}))^{-1}; comVal(\mathsf{S}))$ is a list of pairs (variable of a common block declared in SP, value) (see the example of section 3.2), and it belongs to the object called "Value of common blocks variables" in fig. 6.

3.3.3. Computation of InputVar (values of variables at program point γ) The variables of SP (and thier initial values) are:

- formal parameters and variables of common blocks that are declared in SP, that is VFormal $\dagger$ VComCalled: in SP, if a formal parameter is also declared in a common block, then its value is the value given by the common block.
- variables of the calling procedure P that have a scope in SP. These are the current static variables, but not the local variables of P (that are represented by a set): $localVar(\mathsf{E}) \lhd val(\mathsf{S})$.

The restriction between these two maps models scope rules between procedures. The resulting map InputVar belongs to the object called Value in fig. 6.

Example



Definitions

$$\mathsf{EnvSP} \triangleq environment(name^{-1}(\mathsf{SP}))$$

1. Caller $\hookrightarrow$ called propagation

$$\mathsf{VFormal} \triangleq (formal(\mathsf{EnvSP}))^{-1}; \mathsf{LParam}; val(\mathsf{S})$$
$$\mathsf{VComCalled} \triangleq \mathbf{Corres}((comDecl(\mathsf{EnvSP}))^{-1}; comVal(\mathsf{S}))$$
$$\mathsf{InputVar} \triangleq (localVar(\mathsf{E}) \lhd val(\mathsf{S})) \dagger (\mathsf{VFormal} \dagger \mathsf{VComCalled})$$
$$\mathsf{S1} \triangleq (val \otimes comVal)^{-1}(\mathsf{InputVar}, \mathsf{VComCalled})$$
$$\mathsf{Cl}' \triangleq (\mathbf{dom}(comDecl(\mathsf{E})) \cup \mathsf{Cl}) - \mathbf{dom}(comDecl(\mathsf{EnvSP}))$$

2. Called $\hookrightarrow$ caller propagation

$$\mathsf{VActual} \triangleq \mathsf{LParam}^{-1}; formal(\mathsf{EnvSP}); val(\mathsf{S2})$$
$$\mathsf{VComCaller} \triangleq \mathbf{Corres}((comDecl(\mathsf{E}))^{-1}; comVal(\mathsf{S2}))$$
$$\mathsf{OutputVar} \triangleq (localVar(\mathsf{EnvSP}) \lhd val(\mathsf{S2})) \dagger (\mathsf{VActual} \dagger \mathsf{VComCaller})$$
$$\mathsf{OutputComBlock} \triangleq [\mathbf{dom}(comDecl(\mathsf{EnvSP})) - (\mathbf{dom}(comDecl(\mathsf{E})) \cup \mathsf{Cl})] \lhd comVal(\mathsf{S2})$$
$$\mathsf{S}' \triangleq (val \otimes comVal)^{-1}(\mathsf{OutputVar}, \mathsf{OutputComBlock})$$

Dynamic semantics rule

$$\frac{\mathsf{EnvSP}, \mathsf{S1}, \mathsf{Cl}' \overset{sem}{\vdash} stmt(\mathsf{EnvSP}) : \mathsf{S2}}{\mathsf{E}, \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) : \mathsf{S}'}$$

Figure 7. Dynamic semantics of a call statement.

3.3.4. Computation of S1 (known values at program point γ) $val \otimes comVal$ is an injective function:

- Any state State represents at least one program point, where values of some variables (*i.e.* the static variables) are known. These values are precisely given by the pair of maps $(val \otimes comVal)(\mathsf{State})$. Thus, $val \otimes comVal$ is a total function.
- Given two states S1 and S2, $(val \otimes comVal)(\mathsf{S1}) = (val \otimes comVal)(\mathsf{S2})$ means by definition of a pair that $val(\mathsf{S1}) = val(\mathsf{S2})$ and $comVal(\mathsf{S1}) = comVal(\mathsf{S2})$. This means that all the values of static variables are the same, and that the static variables are also the same in S1 and S2. Thus, S1 and S2 denote a same state.

We can then write $(val \otimes comVal)^{-1}(\mathsf{InputVar}, \mathsf{VComCalled})$ to denote a State object for SP, as explained previously (see section 3.2). We call this state S1.

3.3.5. Computation of CI' (names of inherited common blocks at program point γ) These common blocks CI′ that are inherited by SP are those that have a scope in its caller except those that are also declared in SP (these are $\mathbf{dom}(comDecl(\mathsf{EnvSP}))$, where the domain of the map $comDecl(\mathsf{EnvSP})$ yields the names of common blocks). The common blocks that have a scope in the caller are its declared common blocks $\mathbf{dom}(comDecl(\mathsf{E}))$ and its inherited common blocks CI.

3.3.6. Backward propagation Once values have been transmitted to SP, the current program point is γ. Then, statements of SP are executed given the environment of SP (EnvSP), its state (S1) and inherited common blocks (CI′) (premise of the dynamic semantics rule), yielding a new State object S2 for SP. S2 represents the new values (for common blocks and parameters of SP) that need to be transmitted to the calling procedure.

The analysis has reached program point δ. The known values are then transmitted back to the calling procedure: the maps LParam and $comDecl(\mathsf{E})$ (from actual parameters and common blocks variables to their values) are updated (backward propagation, second part of definitions in fig. 7), yielding the final state S′ of the caller. Similarly to VFormal in the forward propagation,

$$\mathsf{VActual} \triangleq \mathsf{LParam}^{-1}; formal(\mathsf{EnvSP}); val(\mathsf{S2})$$

is a list of pairs (actual parameter, value of formal parameter). The definition of VComCaller (resp. OutputVar) is very close to the definition of VComCalled (resp. InputVar).

SP has inherited the values of common blocks from P. But, some values of common blocks of SP are not backward propagated in P. Thus, the definition of S1 differs slightly from the definition of S′. The values OutputComBlock of common blocks at program point β are the values of common blocks at program point δ (*i.e.* $comVal(\mathsf{S2})$), except for the common blocks declared in SP that do not have

a scope in P. The common blocks of SP that do not have a scope in P are those defined in P and those inherited by P (from one of its callers). These are:

$$\mathbf{dom}(comDecl(\mathsf{EnvSP})) - (\mathbf{dom}(comDecl(\mathsf{E})) \cup \mathsf{Cl}).$$

Thus, the values OutputComBlock of common blocks at program point β are:

$$[\mathbf{dom}(comDecl(\mathsf{EnvSP})) - (\mathbf{dom}(comDecl(\mathsf{E})) \cup \mathsf{Cl})] \lhd\!\!+ comVal(\mathsf{S2}).$$

3.4. Pointer Representation

For every pointer variable, we need to represent the set of objects it may point to. Here, an object is a location that can store information (for example, variables). In our specifications, we use stores to represent associations between variables and their values. The variables are represented by locations in stores [5]. In a procedure, the set of values (denoted by $Value$) includes integers and other values (such as locations denoted by $Location$). Thus, locations are specific values. The dynamic semantics of pointers is modeled by the following functions that are defined in fig. 8:

- $locOf$ maps (simple) identifiers to their locations.
- The map $locOfGen$ extends the $locOf$ map to left-hand sides and dereferences. The location of a pointed record is the value of its first field.
- 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 $locOf$ map. The store of a pointer is the location of its pointed object, if the pointer points to a target. If the pointer has the `NULL` value, it is not represented in the *store* function. Thus, *store* is a partial function.

$$\begin{array}{rl} locOf & \in VarName \to Location \\ locOfGen & \in Lhs \to Location \\ store & \in Location \nrightarrow Value \cup Location \quad \texttt{and} \quad Value \subseteq \mathcal{VALUE} \\ accessField & \in Location \times VarName \nrightarrow Location \end{array}$$

$$\begin{array}{r} For \\ i \in VarName \\ and\ l \in Lhs \end{array} \left\{ \begin{array}{rcl} locOfGen(i) & = & locOf(i) \\ locOfGen(deref(l)) & = & store(locOfGen(l) \\ locOfGen(field\_lhs(l,i)) & = & accessField(locOfGen(l),i) \end{array} \right.$$

Figure 8. Dynamic semantics of some variables.

- given the location of a record r and a field f, $accessField$ yields the location of $r.f$. This is a partial function since only record names with their corresponding fields may have a location.

The function mapping variables to their known values was previously called *val* in our object diagram. It is now defined as:

$$val \triangleq locOfGen; store \triangleright \mathcal{VALUE}.$$

The map *pointsTo* from pointers to their targets is defined as follows:

$$pointsTo \triangleq locOfGen; store; locOfGen^{-1}$$

EXAMPLE: $pointsTo = \{p \mapsto \mathbf{deref}(q), r \mapsto loc\}$ □

Fig. 9 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 and details the maps *val* and *pointsTo*. 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 *locOfGen*. The definition of this map is illustrated in the last part of fig. 9.

Fig. 10 models the whole data that are propagated during the execution of any statement. It extends fig. 6 by showing functions modeling pointer variables. A new object representing locations is accessed from State through the *locOfGen* function. Values of variables are given by the *store* function. Thus, the object called State may be considered as a triplet *state*(set of locations, set of stores, set of values for variables of common blocks), where *state* denotes a constructor of occurrences of this object.

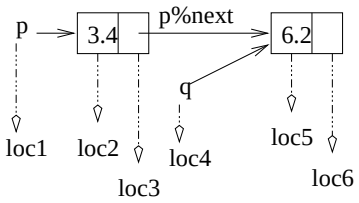
3.5. Dynamic Semantics of Assignments Between Pointers

Fig. 11 shows the dynamic semantics of a pointer assignment $p \Rightarrow q$, given two pointers p and q, an environment E, a state S and inherited common blocks Cl (same hypotheses as in the rule for a call statement). The execution of the assignment modifies only locations and stores. This means that it updates the current state S. The current locations and stores are respectively $\mathsf{L} \triangleq locOfGen(\mathsf{S})$ and $\mathsf{ST} \triangleq store(\mathsf{S})$. The dynamic semantics expresses that L and ST are updated by the alias introduced by that assignment.

q points to a target $*q$ if $q \in \mathbf{dom}(pointsTo)$ (rule 1). In this case, when p is affected by q:

- p points to $*q$. The store of the location of p (*i.e.* $\mathsf{ST}(\mathsf{L}(p))$) becomes the location of q (*i.e.*$\mathsf{L}(q)$), yielding a new map of locations $\mathsf{L1}$.
- the location of $*p$ becomes the location of $*q$, $\mathsf{ST}(\mathsf{L}(q))$, yielding a new map of stores $\mathsf{ST1}$.

Representation of a list



$$
\begin{aligned}
locOf &= \{p \mapsto loc_1, q \mapsto loc_4\} \\
locOfGen &= \{\mathbf{deref}(p) \mapsto loc_2, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto loc_5\} \cup locOf \\
store &= \{loc_1 \mapsto loc_2, loc_3 \mapsto loc_5, loc_4 \mapsto loc_5, loc_2 \mapsto 3.4, loc_5 \mapsto 6.2\} \\
accessField &= \{(loc_2, next) \mapsto loc_3, (loc_2, ident) \mapsto loc_2, (loc_5, ident) \mapsto loc_5, \\
&\quad (loc_5, next) \mapsto loc_6\}
\end{aligned}
$$

Location of the cell $p\%next\%ident$

The abstract syntax of $p\%next\%ident$ is:
$\mathbf{fieldLhs}(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)), ident)$.

$$
\begin{aligned}
&locOfGen(\mathbf{fieldLhs}(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)), ident)) = \\
&accessField(locOfGen(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next))), ident) = \\
&accessField(store[locOfGen(\mathbf{fieldLhs}(\mathbf{deref}(p), next))], ident) = \\
&accessField(store[accessField(locOfGen(\mathbf{deref}(p), next))], ident) = \\
&accessField(store[accessField(store(locOf(p)), next)], ident) \\
&accessField(store[accessField(store(loc_1), next)], ident) = \\
&accessField(store(loc_3), ident) = loc_5
\end{aligned}
$$

Maps val and $pointsTo$

$$
\begin{aligned}
locOfGen; store &= \{\mathbf{deref}(p) \mapsto 3.4, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto 6.2, \\
&\quad p \mapsto loc_2, q \mapsto loc_5\} \\
val &= \{\mathbf{deref}(p) \mapsto 3.4, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto 6.2\} \\
pointsTo &= \{p \mapsto \mathbf{deref}(p), q \mapsto \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next))\}
\end{aligned}
$$

Figure 9. An example of linked list.

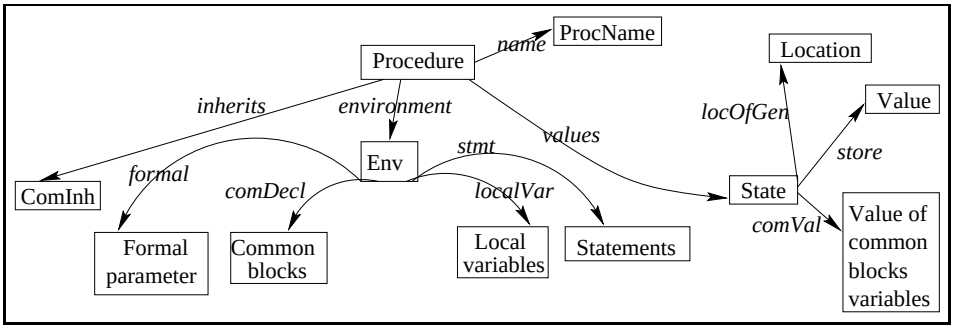


Figure 10. Object diagram showing data propagated during the execution of a procedure.

A new state S1 is then created from its three components L1, ST1 and $comVal(S)$. This creation is very close to the creation of the states S1 and S′ in section 3.3 (see fig. 7).

If q has the NULL value, $q \notin \mathbf{dom}(store)$ (see section 3.4) and then, by definition of $pointsTo$ $q \notin \mathbf{dom}(pointsTo)$. In this case, only the store is updated by the value NULL of the location of p (rule 2). A new state S2 is created from this new store and from its two other components that have not changed.

If there is no location pointed by q (rule 3) then, the maps of locations and stores are restricted:

- p does not point to any location. L becomes $\{\mathbf{deref}(p)\} \lhd\!\!\!- \mathsf{L}$.
- $*p$ has no location anymore. ST becomes $\{\mathsf{L}(p)\} \lhd\!\!\!- \mathsf{ST}$.

3.6. Interprocedural Partial Evaluation

The partial evaluation of a program is an analysis that propagates:

- the same data as the dynamic semantics (environment, inherited common blocks and state),
- specialized versions of already specialized procedures (see section 2.2).

Fig. 12 shows data that are propagated during the partial evaluation. It extends fig. 4 and fig. 10. As in fig. 6, an instance of the diagram of fig. 12 (without the object Version) is an implicit parameter of the sequents belonging to the partial evaluation system, to avoid overloading the sequents. Except for the object Version that is treated separately and appears in the sequents, the objects of the instance are never modified by the sequents.

The partial evaluation relation is the combination of the propagation and simplification relations (see section 1). For simple statements, the partial evaluation relation is directly defined, but for other statements such as call statements, the

Definitions

$$\mathsf{L} \triangleq locOfGen(\mathsf{S})$$
$$\mathsf{ST} \triangleq store(\mathsf{S})$$

$$rule\ 1 \quad \begin{cases} \mathsf{L1} \triangleq \mathsf{L} \dagger \{\mathbf{deref}(p) \mapsto \mathsf{ST}(\mathsf{L}(q))\} \\ \mathsf{ST1} \triangleq \mathsf{ST} \dagger \{\mathsf{L}(p) \mapsto \mathsf{L}(q)\} \\ \mathsf{S1} \triangleq (locOfGen \otimes store \otimes comVal)^{-1}(\mathsf{L1}, \mathsf{ST1}, comVal(\mathsf{S})) \end{cases}$$

$$rule\ 2 \quad \begin{cases} \mathsf{ST2} \triangleq \mathsf{ST} \dagger \{\mathsf{L}(p) \mapsto \texttt{NULL}\} \\ \mathsf{S2} \triangleq (locOfGen \otimes store \otimes comVal)^{-1}(\mathsf{L}, \mathsf{ST2}, comVal(\mathsf{S})) \end{cases}$$

$$rule\ 3 \quad \begin{cases} \mathsf{L3} \triangleq \{\mathbf{deref}(p)\} \lhd \mathsf{L} \\ \mathsf{ST3} \triangleq \{\mathsf{L}(p)\} \lhd \mathsf{ST} \\ \mathsf{S3} \triangleq (locOfGen \otimes store \otimes comVal)^{-1}(\mathsf{L3}, \mathsf{ST3}, comVal(\mathsf{S})) \end{cases}$$

Dynamic semantics rules

$$\frac{q \in \mathbf{dom}(pointsTo)}{\mathsf{S} \overset{sem}{\vdash} p \Rightarrow q : \mathsf{S1}} \quad \text{(rule 1)}$$

$$\mathsf{S} \overset{sem}{\vdash} p \Rightarrow \texttt{NULL} : \mathsf{S3} \quad \text{(rule 2)}$$

$$\frac{q \notin \mathbf{dom}(pointsTo) \qquad q \neq \texttt{NULL}}{\mathsf{S} \overset{sem}{\vdash} p \Rightarrow q : \mathsf{S2}} \quad \text{(rule 3)}$$

Figure 11. Dynamic semantics of a pointer assignment.

rules are more complex and we define separately the propagation and simplification relations. The dynamic semantics propagates all values of variables through statements. Our propagation relation is close to the dynamic semantics relation except that it propagates only known values of variables (with respect to the initial values given by the user). Thus, sequents of the dynamic semantics and propagation systems may share the same formalism. For that reason, in the *propag* system, we have overloaded the colon symbol representing the dynamic semantics relation instead of using a new symbol.

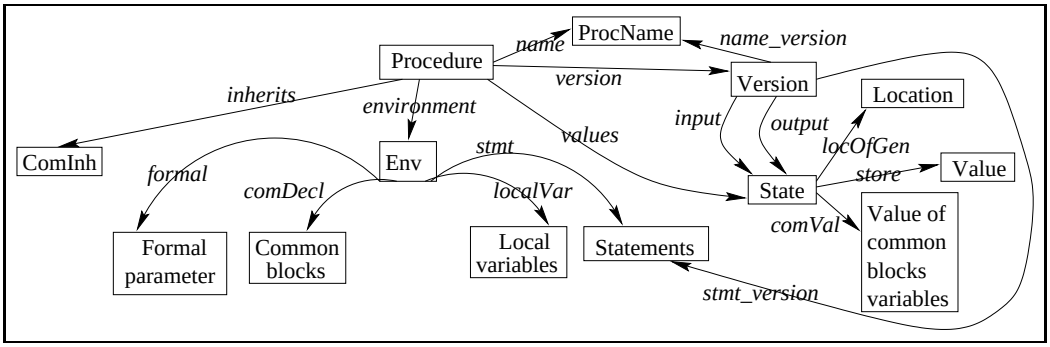


Figure 12. Object diagram showing data propagated during partial evaluation.

3.6.1. Propagation This section discusses how we have manually derived the propagation of a call statement from its dynamic semantics. This process is similar to the process described in [9], where program slicing algorithms have been automatically derived from semantic specifications, thanks to term rewriting systems.

Fig. 13 specifies the propagation of a call statement to a procedure SP with a list LParam of actual parameters, given an environment E, a state S and inherited common blocks CI. E, S and CI are propagated through the statement CALL SP(LParam), and the result from the propagation is a new state S′.

As the dynamic semantics, the propagation starts with a forward propagation followed by a backward propagation. Compared to the dynamic semantics, some parameters and variables have unknown values. Formal parameters of SP that have a known value are denoted by StaticFormal. As in fig. 7, $(formal(\mathsf{EnvSP}))^{-1}; \mathsf{LParam}$ maps the formal parameters of SP to the actual parameters of the current call statement. Actual parameters are also evaluated but here the evaluation yields expressions whose values may be unknown. As our partial evaluation propagates only equalities between variables and values, the resulting map is restricted to static formal parameters (*i.e.* formal parameters that have been totally evaluated).

In like manner, during the backward propagation process:

1. $\mathsf{LParam}^{-1}; formal(\mathsf{EnvSP})$ is a map of pairs (actual parameter, corresponding formal parameter).
2. Formal parameters are evaluated with respect to the known values at the end of the propagation in SP.
3. Dynamic actual parameters (they map to an expression whose value is unknown, *e.g.* $z = a - 2$ with a unknown) are removed from the map. The resulting map is StaticActual.

[22] gives examples of expressions propagated through called procedures.

3.6.2. Simplification Fig. 14 specifies the simplification of a call statement to a procedure SP. The simplification rule is a general rule that:

Definitions

$$\mathsf{EnvSP} \triangleq environment(name^{-1}(\mathsf{SP}))$$

1. Caller $\hookrightarrow$ called propagation

$$\begin{aligned}
\mathsf{StaticFormal} &\triangleq [(formal(\mathsf{EnvSP}))^{-1}; \mathsf{LParam}; \mathbf{eval}(val(\mathsf{S}))] \triangleright \mathcal{VALUE} \\
\mathsf{StaticComCalled} &\triangleq \mathbf{Corres}((comDecl(\mathsf{EnvSP}))^{-1}; comVal(\mathsf{S})) \\
\mathsf{InputVar} &\triangleq (localVar(\mathsf{E}) \lhd val(\mathsf{S})) \dagger (\mathsf{StaticFormal} \dagger \mathsf{StaticComCalled}) \\
\mathsf{S1} &\triangleq (val \otimes comVal)^{-1}(\mathsf{InputVar}, \mathsf{StaticComCalled}) \\
\mathsf{Cl}' &\triangleq \mathbf{dom}(comDecl(\mathsf{E})) \cup \mathsf{Cl} - \mathbf{dom}(comDecl(\mathsf{EnvSP}))
\end{aligned}$$

2. Called $\hookrightarrow$ caller propagation

$$\begin{aligned}
\mathsf{StaticActual} &\triangleq [\mathsf{LParam}^{-1}; formal(\mathsf{EnvSP}); \mathbf{eval}(val(\mathsf{S2}))] \triangleright \mathcal{VALUE} \\
\mathsf{StaticComCaller} &\triangleq \mathbf{Corres}((comDecl(\mathsf{E}))^{-1}; comVal(\mathsf{S2})) \\
\mathsf{OutputVar} &\triangleq (localVar(\mathsf{EnvSP}) \lhd val(\mathsf{S2})) \dagger (\mathsf{StaticActual} \dagger \mathsf{StaticComCaller}) \\
\mathsf{OutputComBlock} &\triangleq [\mathbf{dom}(comDecl(\mathsf{EnvSP})) - (\mathbf{dom}(comDecl(\mathsf{E})) \cup \mathsf{Cl})] \lhd comVal(\mathsf{S2}) \\
\mathsf{S}' &\triangleq (val \otimes comVal)^{-1}(\mathsf{OutputVar}, \mathsf{OutputComBlock})
\end{aligned}$$

Propagation rule

$$\frac{\mathsf{EnvSP}, \mathsf{S1}, \mathsf{Cl}' \overset{propag}{\vdash} stmt(\mathsf{EnvSP}) : \mathsf{S2}}{\mathsf{E}, \mathsf{S}, \mathsf{Cl} \overset{propag}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) : \mathsf{S}'}$$

Figure 13. Propagation of call statements.

- simplifies the statements of the called procedure (first premise of the rule),
- propagates data through these simplified statements (second premise),
- creates a new version of the called procedure (third premise).

As specialized versions of called procedures are maintained (with a strategy introduced in section 2.2), the implementation of this rule has been optimized ([22]) to take into account the cases when:

- the called procedure has already been specialized with respect to the same static data,

- the most specialized version of the called procedure is not as specialized as wanted.

Definitions

$\mathsf{EnvSP} \triangleq environment(name^{-1}(\mathsf{SP}))$ as in the previous figure
$\mathsf{S1} \triangleq \ldots$
$\mathsf{Cl}' \triangleq \ldots$

Simplification rule

$$\mathsf{EnvSP}, \mathsf{S1}, \mathsf{Cl}', \mathsf{V} \overset{simpl}{\vdash} stmt(\mathsf{EnvSP}) \hookrightarrow \mathsf{StmtV}', \mathsf{V}' \qquad \text{(premise 1)}$$

$$\mathsf{EnvSP}, \mathsf{S1}, \mathsf{Cl}' \overset{propag}{\vdash} \mathsf{StmtV}' : \mathsf{S2} \qquad \text{(premise 2)}$$

$$\mathsf{NewName} \in \mathcal{PROCNAME} - \mathrm{ProcName} \qquad \text{(premise 3)}$$

$$\mathsf{E}, \mathsf{S}, \mathsf{Cl}, \mathsf{V} \overset{simpl}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \hookrightarrow \texttt{CALL}\ \mathsf{NewName}(\mathsf{LParam}), \mathsf{V}' \cup \{\mathsf{NewVersion}\}$$

with $\mathsf{NewVersion} \triangleq (\mathsf{SP}, \mathsf{NewName}, \mathsf{S}, \mathsf{S2}, \mathsf{StmtV}')$

Figure 14. Simplification of call statements.

The simplification rule of fig. 14 refers to intermediate variables that are computed during the propagation. Thus, the definitions of these variables (S1 and Cl′) are those given in fig. 13 and they have not been repeated in fig. 14. As explained previously, the set of initial static variables of SP is S1. Thus, during the simplification process:

- The statements of SP are simplified into StmtV′, yielding an updated set of specialized versions V′ (first premise).
- Then data (EnvSP, S1 and Cl′) are propagated through these simplified statements StmtV′, yielding a new state S2 for SP (second premise).
- A new name (NewName) is created for StmtV′ (third premise). This new name is a possible name that is not already a procedure name:

$$\mathsf{NewName} \in \mathcal{PROCNAME} - \mathrm{ProcName}.$$

- The call to SP is replaced by the call to NewName with the same parameters (no unfolding). The new specialized version NewVersion is also added among specialized versions of SP (conclusion of the rule). This new version is the quintuplet (*name of original procedure* = SP, *version name* = NewName, *input data* = S, *output data* = S2, *statements* = StmtV′).

3.6.3. Partial Evaluation Given an environment E, a state S, inherited common blocks Cl, and specialized versions V, the partial evaluation of a call statement combines the propagation of E, S and Cl through this statement and the simplification of this statement. Fig. 15 shows how the partial evaluation of a call statement yields a new call statement (with the same actual parameters), a new state and a new set of specialized versions.

$$\frac{\mathsf{E},\mathsf{S},\mathsf{Cl} \overset{propag}{\vdash} \texttt{CALL}\ (\mathsf{LParam}) : \mathsf{S}' \qquad \mathsf{E},\mathsf{S},\mathsf{Cl},\mathsf{V} \overset{simpl}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \hookrightarrow \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}),\mathsf{V}'}{\mathsf{E},\mathsf{S},\mathsf{Cl},\mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}),\mathsf{S}',\mathsf{V}'}$$

Figure 15. Partial evaluation of call statements.

4. Correctness of the Partial Evaluation

The natural semantics rules that specify partial evaluation define inductively a formal system (*PE*), that groups the propagation (*propag*) and the simplification (*simpl*) systems. Propagated data and simplified statements are built-up in a unique way. To prove that the partial evaluation system is correct means proving that it is sound (each result of a residual program is correct with respect to the initial program) and complete (each correct result is computed by the residual program, too) with respect to the dynamic semantics of Fortran 90 given in section 3. The dynamic semantics rules are not proved: they are supposed to define *ex nihilo* the semantics of Fortran 90.

The general schema to prove the correctness of the partial evaluation is the following. Given an initial procedure P, its environment E represents its syntax, and its inherited common blocks Cl can be computed from its call graph. Intially, not a single specialized version has been created. Given some values S0 for known input data, the partial evaluation of P yields a residual procedure P'. Let a state S' and a set of specialized versions V' be such that the sequent $\mathsf{E},\mathsf{S0},\mathsf{Cl},[\,] \overset{PE}{\vdash} \mathsf{P} \mapsto \mathsf{P}',\mathsf{S}',\mathsf{V}'$ holds in the partial evaluation system (*PE*). In this sequent [ ] denotes the initial and empty list of specialized versions. Given this sequent, we want to prove that

the partial evaluation is sound and complete with respect to the dynamic semantics (*sem* system). This is expressed by the following property of partial evaluation, where the implication $\Rightarrow$ (resp. $\Leftarrow$) states completeness (resp. soundness).

$$\forall \mathsf{S}, \mathsf{S}' : (\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \mathsf{P} : \mathsf{S}' \Leftrightarrow \mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \mathsf{P}' : \mathsf{S}').$$

S denotes the values of dynamic input variables (initally, their values are unknown). Such variables are the remaining input data once S0 has been given by the user. Thus, S0 $\cup$ S denotes the values of the whole input variables. To prove in the *sem* system this property on programs, we prove it for any Fortran 90 statement we simplify (*i.e.* for any rule of the partial evaluation system). Such a proof proceeds by structural induction on the Fortran 90 abstract syntax: the proof that the partial evaluation of a compound statement such as `IF` c `THEN` $i1$ `ELSE` $i2$ is correct is done under the induction hypothesis that states it is correct for the substatements $i1$ and $i2$. The proof for some simple simplifications of statements has been given in [21].

Our approach to prove the correctness of partial evaluation is close to the approach of [8] to prove the correctness of translators: in that paper, dynamic semantics and translation are both given by formal systems and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules that are proved by induction on the length on the proof. However, to simplify our proofs, they do not deal with validity of inference rules in the union of several formal systems (expressing the dynamic semantics of two langages and the traduction from one langage to the other). Furthermore, we do not need rule induction for all our proofs, but only structural induction and sometimes induction on derivations (to handle loops).

As an example, we give here the guidelines for proving by structural induction the completeness of the partial evaluation of a call statement `CALL` SP(LParam). Figure 16 gives the completeness rule to prove. This rule is proved under the induction hypothesis that the statements of SP are complete.

$$\frac{\mathsf{E}, \mathsf{S0}, \mathsf{Cl}, \mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}), \mathsf{V}', \mathsf{S}' \qquad \mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) : \mathsf{S}''}{\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{Cl}, \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}) : \mathsf{S}''}$$

Figure 16. Completeness rule for a call statement.

Let $\mathsf{E}, \mathsf{S0}, \mathsf{Cl}, \mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}), \mathsf{V}', \mathsf{S}'$ be a sequent of the partial evaluation system. Let S and S' be such that the following sequent of the *sem* system holds: $\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) : \mathsf{S}''$. Now we have

to prove that given the same input values, the simplified call statement yields the same output values. This is expressed by the following sequent.

$$\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP'}(\mathsf{LParam}) : \mathsf{S'}$$

It will then prove the completeness of the partial evaluation of the call statement CALL SP(LParam) and of the specialized versions of SP. This is proved in two stages.

1. Firstly, no data declaration is modified during the partial evaluation. Thus, both procedures SP and SP' have the same formal parameters and the same data declarations. It follows that given input values S0∪S the data declarations of SP' yield the same values as the declarations of SP.

2. Secondly, we assume that the induction hypothesis on the simplified statements of SP holds, and deduce that the new version and the specialized statements are complete as required.

5. The Tool

We have implemented our partial evaluator on top of a kernel that has been generated by the Centaur system [6]. 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 90 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 90 procedures (provided by Centaur), a partial evaluator, together with an uniform graphical interface. Fig. 17 shows the architecture of our tool, its inputs and outputs. Given a file of constraints and an initial program represented by several Fortran 90 files, the tool generates the program specialized with respect to these constraints and the corresponding final constraints.

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 in order to develop succinct and efficient Typol rules [14]. Thus, the Typol rules operate on the abstract syntax and they are close to the formal specification rules as shown in [20]. The partial evaluation process transforms an initial abstract syntax tree (representing the initial Fortran procedures and constraints) into a residual abstract syntax tree (representing the specialized code and the final constraints).

The abstract syntax of Fortran 90 is general and close to the abstract syntax of any imperative language. For instance, to be more general, our specifications

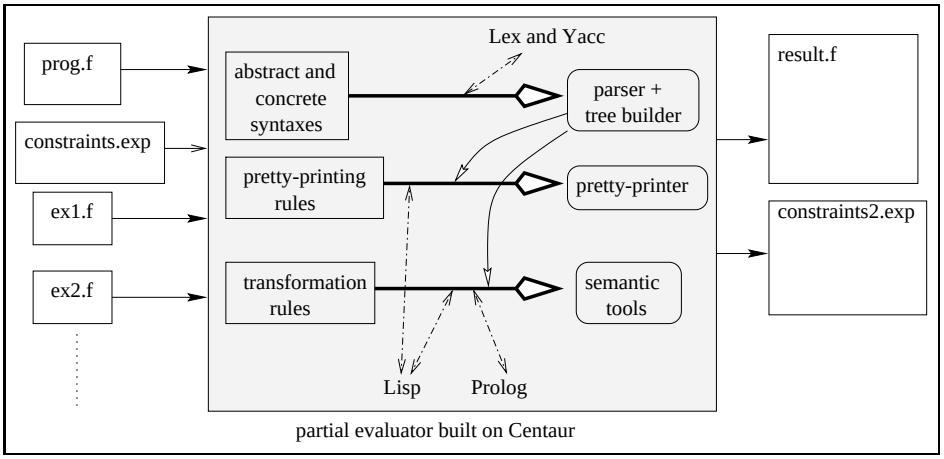


Figure 17. Architecture of the partial evaluator.

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 (without recursion).

We have implemented a graphical interface to facilitate the exploration of Fortran 90 application programs [19]. It has been written in Lisp, enhanced with structures for programming communication between graphical objects and processes. It is shown in fig. 18 and used as follows.

The user starts to define the application program to be specialized. For example, in fig. 18, the user has selected the files called `ex.f`, `exann.f` and `exannsb.f`. The constraints related to this application program are called through a popup menu button (in fig. 18 they are written in the file called `ex-ter.lgaux`). When the partial evaluation is triggered, two new windows are displayed. The first one (called "Initial programs" in fig. 18) displays the procedures to specialize. This is especially useful if some Fortran 90 files have not been already displayed. The second window displays (under the previous one in the figure) the specialized procedures.

Hyperlinks have been added to visualize with color:

- all occurrences in all displays with a special selection,
- specialized versions of a procedure (*e.g.* in the window called "Specialized versions of sp1" in fig. 18),
- propagated data (in a window similar to the "`ex-ter.lgaux`" window, not shown in Fig. 18),
- warning messages in a special message window (called "Centaur messages" in fig. 18) that will open automatically.

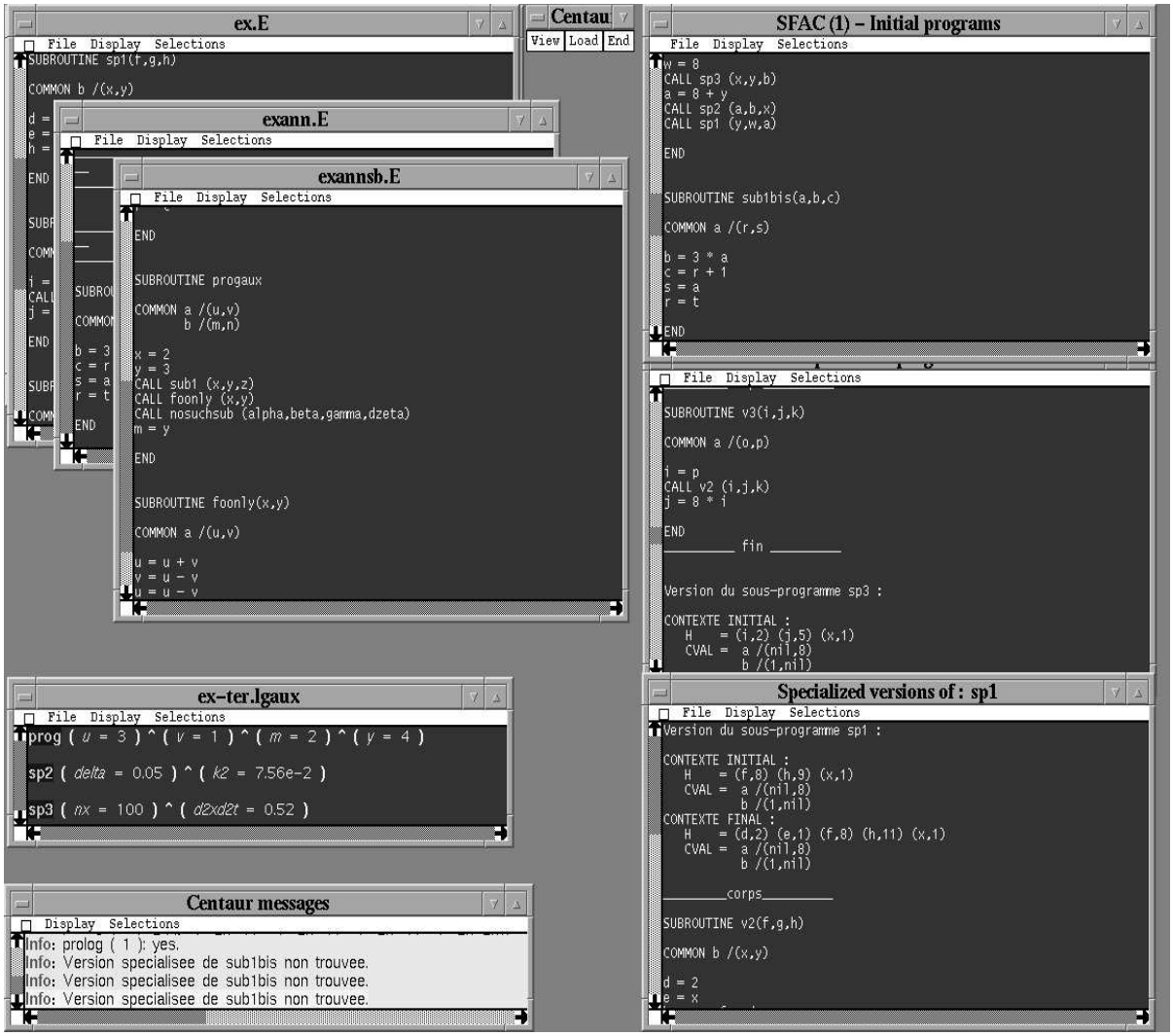


Figure 18. Partial evaluation of a Fortran 90 application program (with reuse of specialized versions).

In fig. 18, the display of the window "Specialized versions of sp1" has been triggered by a click on the call statement to SP1 in the "Initial programs" window. The user may trigger several instances of the tool together. Fig. 18 shows only an instance numbered SFAC(1) (this 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 is killed.

6. 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 and we have derived the propagation from the dynamic semantics, also expressed in natural semantics. 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 to avoid overloading the rules. Data structuration has been presented in object diagrams with access functions.

From the specifications, we have proven by induction the correctness of the partial evaluation with respect to a dynamic semantics of Fortran 90. This proof has been done by hand. We are currently investigating an automatic proof of the correctness of the partial evaluation.

A tool has been implemented from the specifications. 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 the EDF (the French national company that produces and distributes electricity), that provided us with scientific application programs [20]. 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 and to use metrics such as those defined in [17] to evaluate our interprocedural constant propagation. An other current focus is in improving the analysis by propagating general constraints between variables instead of only equalities between variables and values. To this end, we could adapt the rules described in [7].

Furthermore, partial evaluation is complementary to program slicing, another technique for extracting code when debugging a program. Program slicing aims at identifying the statements of the program which impact directly or indirectly on some variable values. We believe that merging partial evaluation (a forward walk through the call graph) and program slicing (a backward walk) would improve a lot the reduction of programs.

Notes

1. we could have used the other symbol for composition $\circ$ that is tantamount to ; since for any pair of binary relations r and p it is defined by $p \circ r = r; p$
2. $\triangleq$ means "by definition"

References

1. ACM. Symposium on partial evaluation, number 4 in ACM Computing Surveys, December 1998.
2. C.B.Jones. Systematic development using VDM. Prentice-Hall, 1990.
3. D.R.Chase and F.K.Zadeck. Analysis of pointers and structures. In Programming Languages Design and Implementation Conference Proceedings, pages 296–31, White Plains, June 1990. ACM SIGPLAN.
4. G.Kahn. Natural semantics. In STACS Proceedings, volume 247 of Lecture Notes in Computer Science, 1987.
5. H.R.Nielson and F.Nielson. Semantics with application - A formal introduction. John Wiley and Sons, 1992.
6. INRIA. Centaur 1.2 documentation, 1994.
7. J.A.Bergstra, T.B.Dinesh, and J.Heering. Toward a complete transformational toolkit for compilers. ACM Transactions on Programing Languages and Systems, (5):639–684, September 1997.
8. J.Despeyroux. Proof of translation in natural semantics. In Symp. on Logic in Computer Science Proceedings, Cambridge, USA, June 1986.
9. J.Field, G.Ramalingam, and F.Tip. Parametric program slicing. In Principles Of Progamming Languages Conference Proceedings, pages 379–392, San Francisco, USA, 1995.
10. J.R.Abrial. The B-Book assigning programs to meanings. Cambridge University Press, 1996.
11. L.O.Andersen. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.
12. M.Sagiv, T.Reps, and R.Wilhelm. Solving shape-analysis problems in languages with destructive updating. In Principles Of Progamming Languages Conference Proceedings, January 1997.
13. N.D.Jones, C.K.Gomard, and P.Sestoft. Partial evaluation and automatic program generation. Prentice-Hall, 1993.
14. N.Dubois and P.Sayarath. Aide à la compréhension et à la maintenance: pointeurs pour la spécialisation de programmes. Master's thesis, IIE-CNAM, June 1996. in French.
15. O.Danvy, R.Glück, and P.Thiemann, editors. International seminar on partial evaluation, volume 1110 of Lecture Notes in Computer Science, Dagstuhl castle, February 1996.
16. R.Baier, R.Glück, and R.Zöchling. Partial evaluation of numerical programs in fortran. In Partial Evaluation and semantics based Program Manipulation Workshop Proceedings, pages 119–132, Melbourne, 1994. ACM SIGPLAN.
17. R.Carini and M.Hind. Flow-sensitive interprocedural constant propagation. In Programming Languages Design and Implementation Conference Proceedings, pages 23–31, La Jolla, June 1995. ACM SIGPLAN.
18. R.Marlet, S.Thibault, and C.Consel. Mapping software architectures to efficient implementation via partial evaluation. In Automated Software Engineering Conference Proceedings, pages 183–192. IEEE, November 1997.
19. R.Vassallo. Ergonomie et évolution d'un outil de compréhension de programmes. Master's thesis, IIE-CNAM, June 1996. in French.
20. S.Blazy and P.Facon. Sfac, a tool for program comprehension by specialization. In Third Workshop on Program Comprehension Proceedings, pages 162–167, Washington D.C., November 1994. IEEE.
21. S.Blazy and P.Facon. Formal specification and prototyping of a program specializer. In TAPSOFT Conference Proceedings, volume 915 of Lecture Notes in Computer Science, pages 666–680, Aarhus, May 1995.
22. S.Blazy and P.Facon. An automatic interprocedural analysis for the understanding of scientific application programs. In O.Danvy et al. [15], pages 1–16.
23. S.Blazy and P.Facon. Partial evaluation for program understanding. ACM Computing Surveys - Symposium on partial evaluation, 4(4), December 1998.
24. W.Landi and B.G.Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In Programming Languages Design and Implementation Conference Proceedings. ACM SIGPLAN, June 1992.