SFAC, a tool for program comprehension by specialization

Sandrine Blazy, Philippe Facon

SFAC, a Tool for Program Comprehension by Specialization

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

Abstract

This paper describes a tool for facilitating the comprehension of general programs using automatic specialization. The goal of this approach was to assist in the maintenance of old programs, which have become very complex due to numerous extensions.

This paper explains why this approach was chosen, how the tool's architecture was set up, and how the correctness of the specialization has been proved. Then, we discuss the results obtained by using this tool, and the future evolutions.

1. Introduction

In [2], we have presented a new approach to assist in the comprehension of general programs, based on partial evaluation techniques to analyze the behaviour of source code. Here, after a brief recall of that work, we will first discuss the use of the tool, secondly we will show how it has been proved with respect to dynamic semantics, and thirdly we will present the evolutions we are working on.

It is well-known that program comprehension is a tedious and time consuming phase of software maintenance. This is particularly true when maintaining scientific application programs that have been written for decades, such as those developped at EDF. Electricité de France (EDF) is the national French company that produces, distributes and provides electricity to the whole country. As such, EDF has to deal with an extensible amount of computerized scientific applications. These application programs are mainly implemented in Fortran, which is an old-fashioned language, but which is still widely used in industry. The reduction of a program when some of its input variables are known was the main wish of EDF scientists and programmers. These people were faced with so general applicaton programs that existing market tools could not help them to understand their application programs. Thus, we have developped a technique for specializing programs by showing several views, a view representing a program functionality or a specific context.

This approach is complementary to classical techniques of representing programs at higher level of abstractions (reverse engineering techniques). Program specialization has been seldom used for program comprehension. Automatic program specialization has been used (as partial evaluation) in compilation to optimize programs or to generate compilers from interpreters (by partially evaluating the interpreter for a given program) [13].

In [2], we have described our technique for restricting the behaviour of a program to specific values of its input variables. We have shown an example of program reduction and we have formally specified the two main tasks of our specialization process by means of inference rules. The aim of this new paper is threefold: first to explain how our tool, SFAC (Specialization of Fortran programs as an Aid to their Comprehension) is used, and second to show how we have proved the completeness and soundness of the specialization rules we had previously presented. At least, we present the evolutions of our tool.

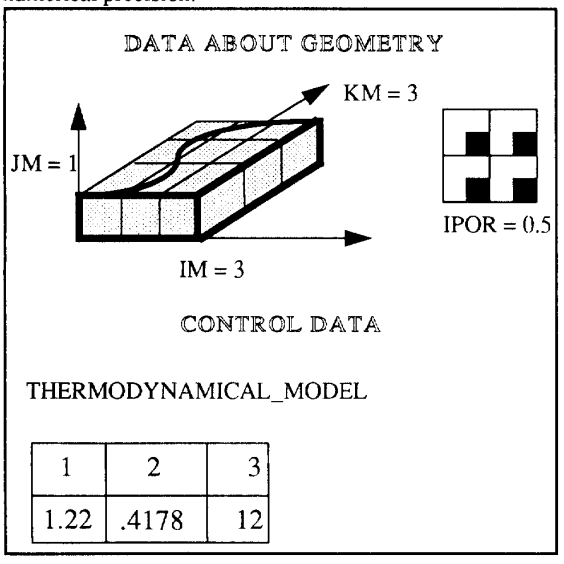
We focus on general purpose programs that are large and complex and whose application domain models encapsulate several models. This generality is implemented by Fortran input variables whose value does not vary in the context of the given application. We have distinguished two classes of such variables:

- *data about geometry*, which describe the modelled domain. They appear most frequently in assignment statements (equations that model the problem).
- *control data*. These are either *filters* necessary to switch the computation in terms of the context (modelled domain), or *tags* allowing to minimize risks due to precision error in the output values. Control data take a

0-8186-5647-6/94 $04.00

finite number of values; they appear in particular in conditions of alternatives or loops.

As an example illustrated in Figure 1, consider the modelization of a liquid flow along the surface of a nuclear power plant component. That volume 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. Thus IM, JM, KM and IPOR are data about geometry. Now we have a filter, THERMODYNAMICAL_MODEL, which is the name of the law that characterizes the liquid. We have also a tag, PRESSURE, with integer values that correspond, by a table, to real pressure values, each one with a specific numerical precision.



1	2	3
1.22	.4178	12

Fig. 1. Data about geometry and control data

Our approach aims at highlighting such input variables in old programs, that have become very complex due to extensive modifications. It led us to develop a technique for program specialization according to specific values of such variables (for instance, the specialization of a 3D-application into a 2D-one by fixing the value of a co-ordinate). This technique simplifies programs that are thus shorter and easier to understand.

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

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

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

The remainder of this paper is organized as follows. In section 2, we recall briefly how we have specified the specialization process. Then, we detail in section 3 how we have implemented our method for specializing Fortran programs. Section 4 explains how to prove the soundness and completeness of the specialization with respect to the dynamic semantics of Fortran. Examples of proofs for some Fortran statements are given in appendix. Section 5 presents conclusions and future work.

2. Specification of the Specialization

To specify the automatic program specialization, we have used inference rules operating on the Fortran abstract syntax and expressed in the natural semantics formalism, augmented with some VDM [9] operators. Natural semantics [11] has its origin in the work of G.Plotkin ([8], [14]). Under the name "structured operational semantics", he gives inference rules as a direct formalization of an intuitive operational semantics: his rules define inductively the transitions of an abstract interpreter. Natural semantics extends that work by applying the same idea (use of a formal system) to different kinds of semantic analysis (not only interpretation, but also typing, translation, etc.).

For every Fortran statement, we have written rules that define inductively the automatic program specialization [5]. Figure 2 gives two examples of these specialization

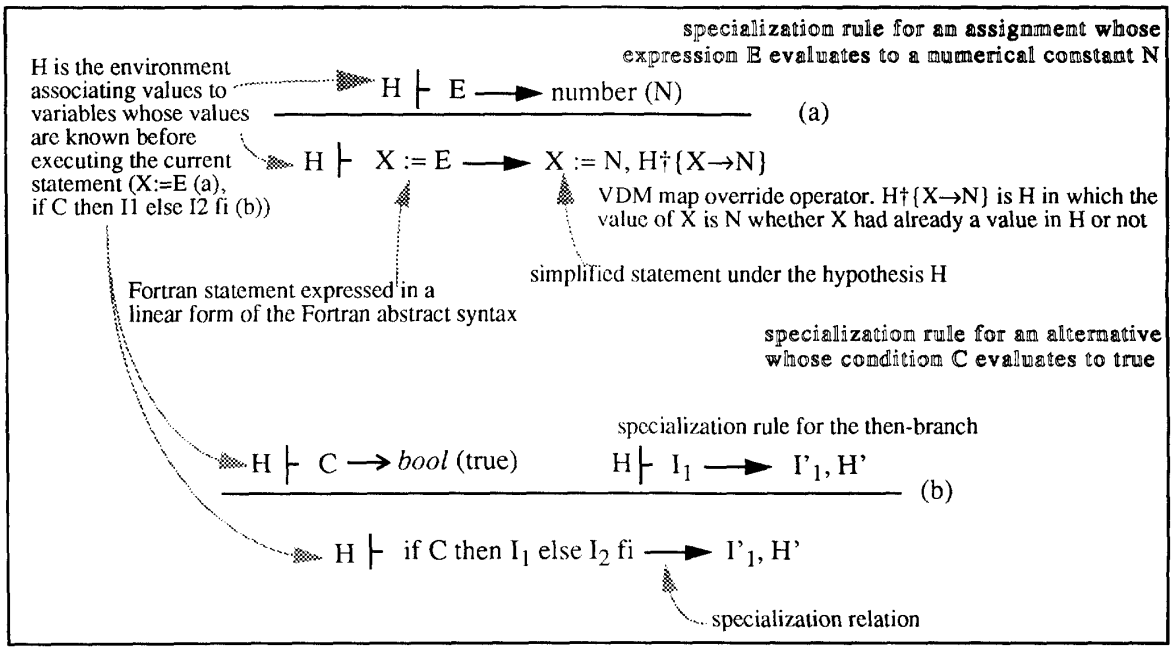


Fig. 2. Examples of specialization rules

rules. Rule (a) expresses the specialization of an assignment whose condition evaluates to a numerical constant and rule (b) expresses the specialization of an alternative whose condition evaluates to true.

3. The Specializer (SFAC)

We have implemented our method for automatic specialization of general programs by using the Centaur system [3], a generic programming environment parametrized by the syntax and semantics of programming languages. This section describes the overall architecture of our tool, SFAC. Then, it gives some quantitative results.

3.1. Architecture of SFAC

When provided with the description of a particular programming language, including its abstract and concrete syntaxes and semantics, Centaur produces an environment specific to that language. The resulting environment consists of a structured editor, an interpreter/debugger and other tools, together with an uniform graphical interface. In Centaur, program texts are represented by abstract syntax trees. The textual (or graphical) representation of abstract syntax trees nodes may be specified by pretty-printing rules. Centaur provides however a default representation.

We have used such a resulting environment, Centaur/Fortran, to build our automatic specializer. From Centaur/Fortran, we have implemented an environment for automatic specialization of general Fortran programs. Figure 3 shows the overall architecture of our system. In this Figure, Centaur/Fortran is represented by the grey part. It consists of :

- a Fortran parser and a tree builder, that have been generated by Centaur from a concrete syntax and an abstract syntax of Fortran
- a Fortran pretty-printer that displays Fortran abstract syntax trees as Fortran texts. Their layout may be customized.

A language for specifying the semantic aspects of languages called Typol is included in Centaur, so that the system is not restricted to manipulations that are based solely on syntax. Typol is an implementation of natural semantics. It can be used to specify and implement static semantics, dynamic semantics and translations. Typol specifications operate on abstract syntax trees; they are compiled into Prolog code. When executing these specifications, Prolog is used as the engine of the deductive system. A Typol program is roughly an unordered collection of axioms and inference rules. We have written Typol rules that implement the automatic specialization. These rules are very close to the ones we have formally specified.

Given a Fortran program or subroutine and some

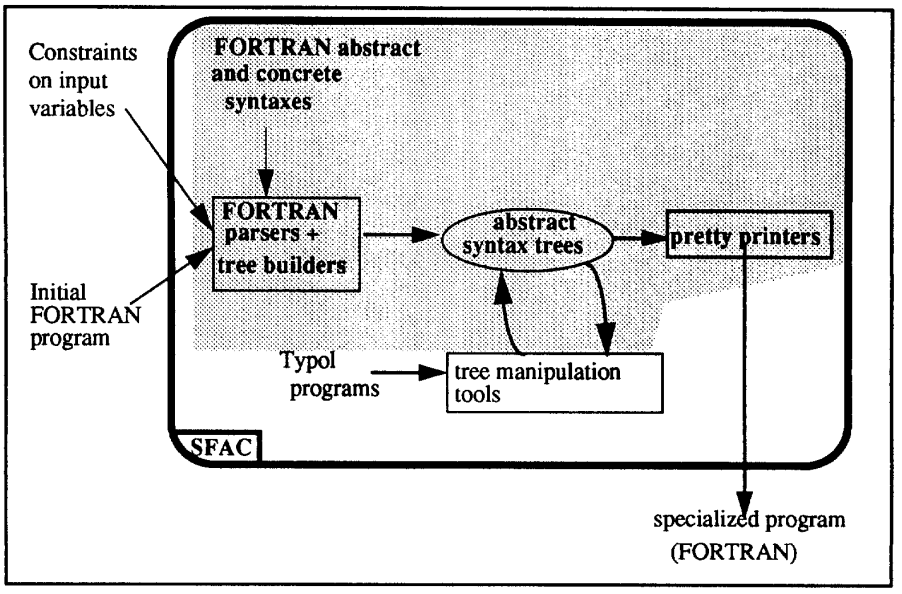


Fig. 3. The Centaur/FORTRAN environment

constraints on input variables (expressed as a list of equalities between variables and constants), the corresponding residual program is obtained by clicking in the window containing the Fortran code. The tool may be used in two ways: by visual display of the residual program as a part of the initial program (for documentation or for debugging) or by generating this residual program as an independent (compilable) program.

Note that some users found our tool interesting also for global program comprehension: in that special case they have not in advance known input variables, but they ask for interesting variables for specialization. Thus, we aim at providing such variables. A set of variables is interesting for specialization specially if their values at the program entry determines the value (true or false) of the conditions of somes alternatives. To detect such variables, we will perform backwards analysis, very close to those performed in program slicing [7]. We will also take into account the importance of the alternatives (number of statements in each branch).

3.2. Quantitative results

We have written about 200 Typol rules to implement our specializer. 10 rules express how to reach abstract syntax nodes representing simplifiable statements. 90 rules perform the normalization of expressions (this normalization allows to propagate constant values). Among the 100 rules for simplification, 60 rules implement the simplification of expressions. The 40 other rules implement the statements simplification. We have written about 25 Prolog predicates to implement the VDM operators we have used to specify the simplification. Thus, these operators are used in Typol rules as in the formal specification of the simplification.

The specializer may analyze any Fortran program, but it specializes only a subset of Fortran 77 (for large scientific applications at EDF, Fortran 77 is used exclusively to guarantee the portability of the applications programs between different machines). This subset is a recommended standard at EDF. For instance, it does not analyze any goto statement (they are not recommended at EDF), but only goto statements that implement specific control structures (e.g. a while-loop).

The average initial length of programs or subroutines we have analyzed is 100 lines of FORTRAN code, which is lengthier than the recommended length at EDF (60-70 lines). The reduction rate amounts from 25% to 80% of lines of code. This reduction is specially important when there is a large number of assignments and conditionals. This is the case for most subroutines implementing mathematical algorithms. For subroutines whose main purpose is editing results or calling other subroutines, the reduction is generally not so important.

4. Proof of Soundness and Completeness of the Specialization

For a tool like SFAC, it is essential to be sure about the correctnees of the specialization, specially if it is used for generating independent (compilable) programs. Our aim in this section is to show how to prove that the specialization we have specified is correct, with respect to the dynamic semantics of Fortran, given in the natural semantics formalism. Recall that partial evaluation of a program P with respect to input variables $x_1, ..., x_m, y_1, ..., y_n$ for the values $x_1 = c_1, ..., x_m = c_m$ must give a residual program P', whose input variables are $y_1, ..., y_n$ and the executions of $P(c_1, ..., c_m, y_1, ..., y_n)$ and $P'(y_1, ..., y_n)$ produce exactly the same results.

We will show that this is expressed by two inference rules, one expressing soundness (each result of P' is correct with respect to P) and one expressing completeness (each correct result is computed by P' too). As P and P' are deterministic, we could have only one rule using equality, but the demonstration of our two rules is not more complicated and is more general (being also applicable for non-deterministic programs).

To prove the specialization, we need a formal dynamic semantics of Fortran and we must prove the soundness and completeness of the specialization rules with respect to that dynamic semantics. To express the dynamic semantics of Fortran, we use the same formalism (natural semantics) as for specialization. Thus, the semantic rules we give have to generate theorems of the form

$$H \overset{sem}{\vdash} I : H'$$

meaning that in environment H, the execution of statement I leads to the environment H' (or the evaluation of expression I gives value H'). These rules are themselves not proved: they are supposed to define ex nihilo the semantics of Fortran, as G.Plotkin [14] and G.Kahn [11] did for other languages like ML.

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

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

$$Ho \vdash P \rightarrow P'$$

is a theorem of the specialization system.

Let us call H the environment containing the values of the remaining input variables. Thus, $H_0 \cup H$ is the environment containing the values of all input variables. With that initial environment, P' (respectively P) evaluates to H' if and if

$$Ho \cup H \overset{sem}{\vdash} P' : H' \quad \text{(respectively } Ho \cup H \overset{sem}{\vdash} P : H')$$

is a theorem of the dynamic semantics (***sem***) system.

Now, soundness of specialization with respect to dynamic semantics means that each result computed by the residual program is computed by the initial program. That is, for each P, P', H_0, H, H': if P is speciaized into P' under hypothesis H_0 and P' executes to H' under hypothesis $H_0 \cup H$, then P executes to H'under hypothesis $H_0 \cup H$. Thus soundness of specialization with respect to dynamic semantics is formally expressed by the first rule of Figure 4.

Completeness of specialization with respect to dynamic semantics means that each result computed by the initial program P is computed by the residual program P'. Thus, it is expressed by the second inference rule of Figure 4. In fact, our approach to prove specialization is very close to the approach of [5] to prove the correctness of translators: in that paper, dynamic semantics and translation are both given by formal systems and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules (that are proved by induction on the length of the proof; here we will use rule induction instead).

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

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

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

Fig. 4. Soundness and completeness of the program specialization

The proof that both rules hold is given in [1].

5. Conclusion

Specialization has given satisfactory results for facilitating program comprehension. Our tool, SFAC, has performed important reductions by specialization of Fortran programs and subroutines. That tool has been proved by rule induction given the dynamic semantics of Fortran.

An industrialization of our tool is planned in the framework of a cooperation between EDF, CEDRIC IIE and Simulog, a company that provided us with some basic tools including Centaur/Fortran. To obtain an insdustrial tool from SFAC, we have to extend the latter in four main ways:

- to perform interprocedural analysis,
- to take into account some new operators of Fortran 90,
- to accept as specialization criteria more general constraints than only equailities between variables and constant values. Thus, the tool will propagate relational expressions, such as y = z and a> 7*b+5.
- to suggest pertinent variables for specialization, as explained previously (page 4).

For performance reasons, the industrial tool will not be based on Prolog operating on anstract syntax trees, but in Lisp operating on graphs representing control and data flow information. Our specialization rules are taken as specification by the team that will do the industrial implementation.

References

[1] S.Blazy La spécialisation de programmes pour l'aide à la maintenance du logiciel, Ph.D. Thesis, CNAM, Paris, December 1993.

[2] S.Blazy, P.Facon, *Partial evaluation as an aid to the comprehension of Fortran programs* IEEE Workshop on Program Comprehension, Capri, July 1993, 46-54.

[3] *Centaur 1.1 documentation.* INRIA, January 1990.

[4] P.D.Coward, *Symbolic execution systems - a review.* Software Engineering Journal, November 1988, pp.229-239.

[5] J.Despeyroux, *Proof of translation in natural semantics.* Symposium on Logic in Computer Science, Cambridge USA, June 86.

[6] C.Consel, S.C.Khoo, *Parametrized partial evaluation.* ACM TOPLAS, 15(3), July 1993, pp.463-493.

[7] K.B.Gallagher, J.R.Lyle, *Using program slicing in software maintenance.* IEEE TOSE, 17(8), August 1991, pp.751-761.

[8] M.Hennessy, *The semantics of programming languages.* Wiley eds., 1990.

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

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

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

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

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

[14] G. Plotkin, *A structural approach to operational semantics.* Report DAIMI FN-19, University of Aarhus, 1981.