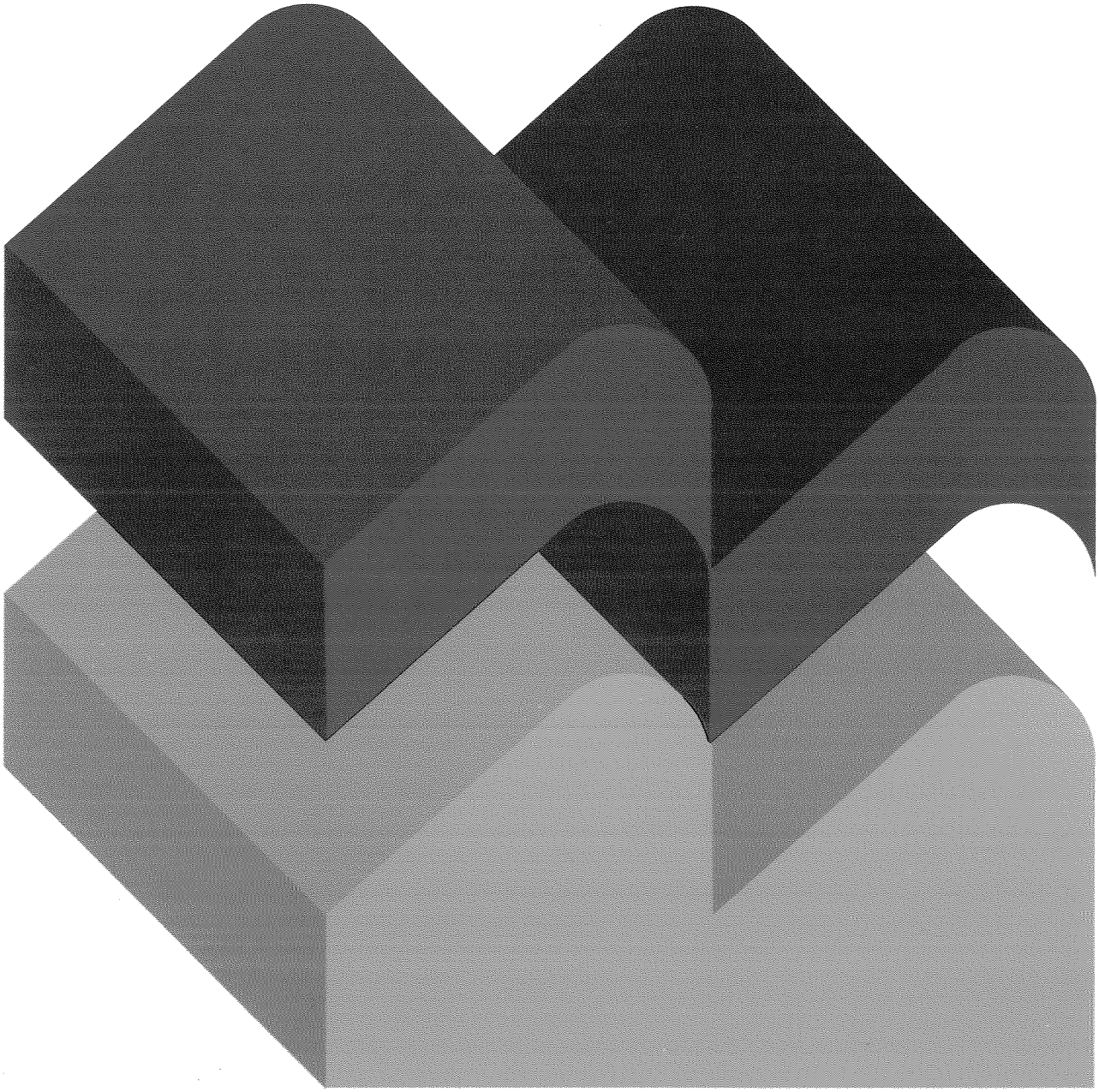
The Draft Formal Definition of Ada®

Commission of the European Communities: Multi-Annual Programme

Alessandro Fantechi, Stefania Gnesi, Paola Inverardi, Franco Mazzanti

Feasibility of a Mapping from the Ada Formal Definition to the NYU SETL Interpreter for Ada

January 1987



®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

THE DRAFT FORMAL DEFINITION OF ADA®

FEASIBILITY OF A MAPPING FROM THE ADA FORMAL DEFINITION TO THE NYU SETL INTERPRETER FOR ADA

Authors: *Alessandro Fantechi (IEI)*
Stefania Gnesi (IEI)
Paola Inverardi (IEI)
Franco Mazzanti (CRAI)

Date: *January 1987*

Workpackage: *Q*

Status: *Final*

Distribution: *Free*

Doc. no. *AdaFD / IEI / 30*

1 Introduction

This report refers to the workpackage Q whose purpose is "to study the extent to which the Ada FD of this project may be correlated to the existing SETL programmed interpreter for Ada as developed by the New York University". To this extent a description of both the projects is given which serves as a basis for the comparison of the two approaches.
In the following section, the architecture of the NYU interpreter [Dewar et al. 80, Dewar et al. 83] and that of the Ada FD are given thus allowing to point out which are, with respect to semantic issues, the most relevant components, namely the interpreter part of the NYU and the dynamic semantics part of the Ada FD. Section 3 analyzes the existing semantics differences proposing a relation which seems to us the most reasonable. A simple example is used for this purpose.

2 Description of the two approaches

The NYU project consisted in the design and implementation of a translator-interpreter for the Ada language. The NYU system has been written in SETL, a high level language [Swartz et al. 86] based on set theory, and it is composed of a compiler front-end and of an interpreter.
The compiler front-end takes as input an Ada program and gives as output an intermediate code program written in a suitable abstract language, AIX (Ada Interpreter eXecutable code), which represents an abstract syntax tree. AIX programs are a translation of the original Ada ones in which all names are disambiguated and all static semantics checks have been performed. Such programs are then the input of the interpreter.
The interpreter gives a description of the Ada run-time semantics and it relies on a semantic model in which tasks are independent from each other and dependencies among tasks are explicitly recorded in suitable global data structures, in this way each active tasks has its own stack of environments.

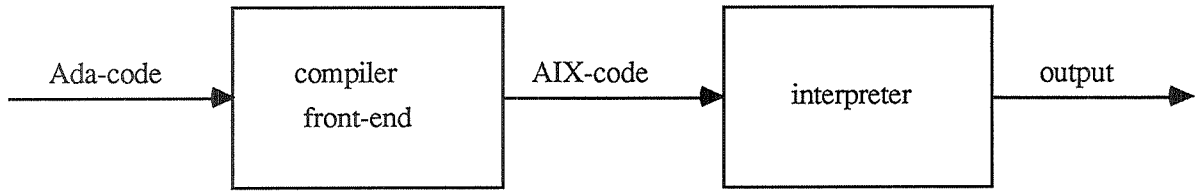


fig.1

The Ada FD project is structured in two main components: static semantics description and the dynamic semantics one. The first one is given using a classical denotational approach in VDM style, it starts from an abstract syntax AS1 which is derived from the Ada concrete syntax and it is modelled as a function which takes values in AS1 and returns a boolean value.

A different abstract syntax, AS2, defines the inputs to the dynamic semantics; AS2 is introduced in order to make the formulas simpler by not having to deal with static information like overloading. Hence an AS1-AS2 transformation is needed. The dynamic semantics is divided in two parts, a set of denotational clauses and the concurrent algebra. Output of the dynamic semantics is a term of the concurrent algebra. This algebra constitutes the run time model of execution of an Ada program, which represents tasks as a flat structure.

Both static semantics and dynamic semantics are written in the metalanguage of the formal definition which allows, in the semantics of the concurrent algebra, an explicit treatment of parallelism and non-determinism.

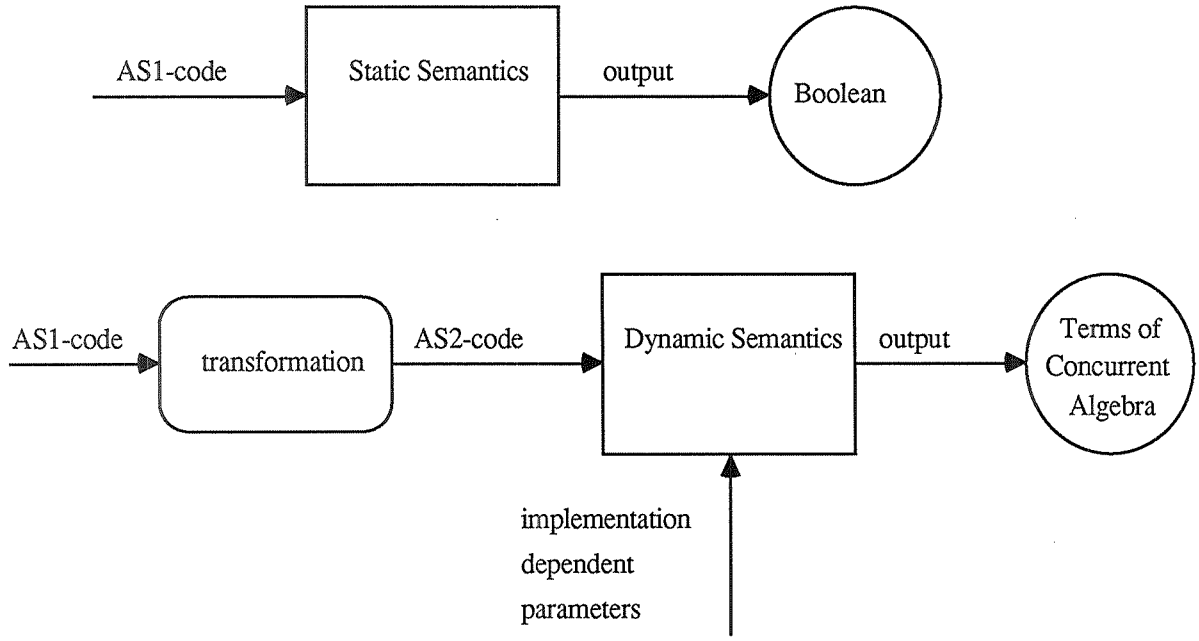


fig. 2

3 Comparison between NYU and Ada FD approach

Looking at the two above schemas it seems that the NYU front-end and the Ada Fd static semantics are not so different; in fact both, in some sense, *implement* a function which takes into account the static semantics aspects of an Ada program; their results are different, but if considering the AS1 to AS2 transformation as a part of the static semantics they differ only in the syntax of the intermediate code. To this respect the two approaches seem comparable in the sense that it is possible to foresee a formal way of checking the correctness of the NYU front-end with respect to the Ada FD static semantics.

The NYU interpreter and the Ada dynamic semantics resolve the dynamic part of an Ada program in a similar way; for example the execution of tasks is modelled in both the approaches as entities whose execution proceed in parallel and therefore they have independent representation.

The main differences between the interpreter and the dynamic semantics are:

i) the non determinism degree, in fact the Ada FD definition allows to explicitly model nondeterminism and parallelism by using the notion of Concurrent Algebra

, while the NYU interpreter, due to its executable nature, allows to model less nondeterminism than that possible in tha Ada language.

ii) the parameterization of Ada FD, as shown in fig.2, on values and functions representing implementation dependent characteristics such as the minimum and maximum integers or the conditions of memory availability; in the NYU interpreter these values and conditions are strictly connected with the particular implementation.

Now we give a very simple example that illustrates the execution of an assignment statement both with the NYU approach and with the Ada FD, to clarify the first consideration.

The treatment of an assignment statement :

```
name := expr
```

with the NYU Ada interpreter is the following:

```
[ ':=', name, expression ]
(':='):
  [ -,name,expression] := STM;
  exec ( [['oeval_', name],
          ['veval_', expression],
          ['assign_']        ]);
 ('assign_'):
  pop( rhs_ );
  pop( lhs_ );
  SETEVAL( lhs_,rhs_,om);
```

The order in which name and expression are evaluated is not defined by the language, but the interpreter choice is that of first evaluating the name and then the expression.

Using the Ada FD, the overall structure of the behaviour modelling the execution of the assignment: is the following:

```
choose
    def object-den = Eval-Name (name) (li)
    in      def  value = Eval-Expr (expr) (li)
       in Make-Assignment (object-den,value) (li)
       end def
    end def
  or
    def value = Eval-Expr (expr) (li)
    in      def  object-den = Eval-Name (name)  (li)
       in Make-Assignment (object-den,value) (li)
       end def
    end def
```

```
    end choose;
```

First the name and the expression are evaluated in a not specified order, then the assignment is executed. In this case, an explicit modelling of the Ada semantics is given. Any implementation can, consistently with the Ada FD, choose between the two alternatives.
This simple example illustrates the different abstraction level of the two approaches.

In other words, the NYU interpreter has to be considered as an implementation of the Ada language, though the use of a high abstract metalanguage and a careful design with respect to implementation bias like model memory etc., make it a high level implementation.

4 Conclusions

The aim of this report was to find a correlation between the Ada FD and the existing SETL programmed interpreter for Ada as developed by the New York University.
We think that it is not feasible to set a formal relation because of the different level of abstraction of the two approaches except than trying to show that the NYU interpreter conforms to the Ada FD in the same way another implementation should do it. We can perform this consistency test by executing a given program with the NYU interpreter and verifying if the resulting stream of input-output actions can be found , modulo internal actions, in one of the branches of the labelled tree which models the same Ada program following the AdaFD.
This test can be performed with the help of the execution enviroment proposed in [Fantechi et al. 87]; more precisely, for each visible action *act* performed by the NYU interpreter, the following goal has to be demonstrated:

```
nontaumoves(state1,act,State2).
```

where *state1* is the PCS state corresponding to the NYU interpreter state in which *act* has been performed, and *State2* returns a PCS state corresponding to the NYU interpreter state after having performed *act*.
The *nontaumoves* goal is defined as:

```
nontaumoves(State1,Action,State2):- moves(State1,Action,State2).
nontaumoves(State1,Action,State2):-
        moves(State1,tau,State3),
        nontaumoves(State3,Action,State2).
```

where *moves* is the predicate defined in [Fantechi et al. 87].

References

[Dewar et al. 80] Dewar,R.B.K., Fisher,G.A, Schomberg,E., Froehlich,R.M., Brynt, S., Clinton C.F., and Burke M. "The NYU Ada translator and interpreter", Proc. of the ACM-SIGPLAN Symposium on the Ada Programming Language, Boston, December 9-11,1980.

[Dewar et al. 83] Dewar,R.B.K., Froehlich,R.M., Fischer,G.A. and Kruchten,P. "An executable semantic model for Ada", Ada/Ed interpreter Ada Project, Courant Institute, NYU 1983.

[Fantechi et al.87] Fantechi,A., Gnesi S., Inverardi P., Montanari U., "On the feasibility of the execution of the Ada Formal Definition", Annex to deliverables 30 and 33 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[Schwartz et al. 86] Schwartz,J.T., Dewar,R.B.K., Dubinsky,E., Schomberg,E., "Programming with sets. An introduction to SETL", Springer-Verlag 1986.