Structuring the Formal Definition of Ada®

Kurt W. Hansen
Dansk Datamatik Center
Lundtoftevej 1C
DK-2800 Lyngby (Copenhagen)
Denmark

Abstract:

The structures of the formal definition of Ada are described in view of the work done so far in the project. At present, a 'difficult' subset of Ada has been defined and the experience gained so far by this work is reported on here.

Currently, the work continues towards the formal definition of the full Ada language.

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

This work has been partly supported by the CEC MAP project on 'The Draft Formal Definition of Ada'. Dansk Datamatik Center - Prime contractor, CRAI - contractor, CNR/IEI - subcontractor, consultants: University of Genoa (Dept. of Mathematics), Tech. University of Denmark (Dept of Comp. Science), and University of Pisa (Dept. of Informatics).

Introduction.

Since the final requirements of Ada (the STEELMAN document) and up to the present Reference Manual for the Ada Programming Language - ANSI/MIL-STD 1815A (RM) the language has been subject to a great deal of discussion, comments, suggestions, and shear critisism.

All of this evaluation has been done on the basis of natural language descriptions, since they are the only ones available. Natural language descriptions of a certain size have a tedency to be ambiguous and contradictory and the RM is no exception to that rule. This has caused some trouble to users, mainly compiler writers.

It is our belief, that having had a formal (mathematical) definition of the language developed together with the natural language description would to a large extent have had avoided these errors in the language design. Not only would it have helped in analysing the complexities of the language which may have altered the design, but it would also have provided an unambiguous definition.

As this was not done, the second best thing is to give a formal definition of the language as it now stands. The number of projects which have attempted this so far [ref INRIA 1982, Bjørner and Oest 1982] strengthen the belief that this work is important, and the fact that none has succeeded in formally defining full Ada also indicates that it is a very difficult task.

In order to gain confidence, and actually prove, that the project is able to formally define the full language Ada, the project has selected two sets of difficult aspects of Ada, in order to show that the expirience and the new methods used are adequate for the task. The reason for having two sets of aspects is, that Ada aspects which are statically difficult are not necessarily dynamically difficult, and vice versa so both modelling static and dynamic semantics were tried out.

At the present stage the project has succesfully finished the trial definition of the Ada subsets, and is now proceeding to formally define full Ada.

This presents the work done, and experience gained in the trial definition of the difficult Ada subsets.

The Overall Structure of the Formal Definition of Ada.

The draft formal definition of Ada has adopted the scheme for defining programming languages as found in VDM [ref Bjørner and Jones 1982]. This means dividing the semantics of the language into two parts: static semantics and dynamic semantics. This gives a good overview of the language features and in this case at the same time complies with the semantics of Ada. As described in the RM two types of rules are identified: rules which describe compile time checks to be performed, and rules describing the dynamic (run time) behaviour of an Ada program. Hence, the static semantics may be seen as the precondition for the dynamic semantics of Ada.

Both static and dynamic semantic definitions are written using the syntax directed approach in a compositional style. Compositional means, that the semantics of a construct is given as a function of the semantics of its subcomponents. Here semantics is understood as a homomorphism (function) from the algebra of syntax into some semantic algebra.

Not only does the compositional style make the writing of the formulae of the semantics of Ada easier as the semantics of each construct is defined in terms of the semantics of its subconstructs, but it also enhances readability as you do not have to remember the semantics of all preceeding constructs in order to understand the semantics of a given construct.

Of course for example in the static semantics you have to use the history to some extent, you have to know the names and types of defined variables in order to perform the type check, but this information is modelled in a separate abstract data type in order not to confuse the overall syntax directed approach.

One may consider the static semantics as the first part of the formal semantics of Ada. Static semantics takes as its input an algebra of syntax which is as ambiguous as the grammar found in the RM. Ambiguous means, that you cannot tell the meaning of a construct without taking into account the context in which it is found. An example is:

```
a := f(x);
```

This is obviously an assignment statement, but the expression f(x) may denote:

- an element of an array
- a function call with one positional parameter
- a type conversion of the expression 'x' to the type 'f'

The ambiguous grammar found in the RM, is translated directly into the algebra of syntax used in the static semantics. The idea is, that only essential information is retained. As an example, in the assignment statement the essential information is the fact that you have a left-hand side name and a right-hand side expression.

The syntactic construct of the assignment statement is therefore in our metalanguage written as:

```
Assignment_stmt :: Name × Expr
```

Static semantics now performs the compile time check on the syntactic constructs found. In the case of f(x), operations on the data type reflecting declarations are used to look up 'f' in order to disambiguate the term f(x). Next overloading is resolved, the static checks for the left-hand side and right-hand side are done, and at last the validity of the assignment statement is tested using the knowledge gained trying to statically check its components (compositionallity). The knowledge could be the fact, that for example the right-hand side is not well-formed at all, and therefore the static check of the whole construct must also fail.

In principle there is no reason why the dynamic semantic should not be able to perform its run time check of and execution on an Ada program on the same abstract syntax the one as used by the static semantics. However in practice this would impose on the dynamic semantics to do most of the work already done in the static semantics over again - like disambiguating syntactic constructs. This would complicate the dynamic semantics considerably, destroying the readability of the final formal definition of the dynamic semantics.

The approach taken in this project, is to impose a transformation on the algebra of syntax used in the static semantics (AS1). This transformation transforms AS1 into an equivalent algebra of syntax (AS2), where the static problems to a large extent have been resolved, and some statically available information is distributed more conveniently (e.g. an aggregate is always given a type).

Resolving the static problems of the syntax means, resolving of syntactic ambiguities, giving unique names to identifiers (apply visibility rules and resolve overloading), adding derived information (attach a type to an aggregate), and removing information not necessary for the dynamic semantics (e.g. the order in which compilation units appear).

The AS2 is then the starting point of the dynamic semantics. In order to improve readability, the AS2 is kept as close to the original Ada program as possible; a user should be able to recognize his program. Furthermore, if a user wants to know some facts about the run time behaviour of his program, he should be able to see the AS2 program

without having to first write an Ada program and then impose the AS1 to AS2 transformation. This of course implies, that the program given to the dynamic semantics must be statically correct, since the successfull application of the static semantics is a prerequisite for the dynamic semantics.

Human Aspects of Structuring.

The writing of formal definitions is still an exercise mostly done in the academic environment since the writing of formal definitions has not yet matured into an engineering practice.

As a reflection of this, most papers found on structuring of formal definitions are aimed at getting the right mathematical structuring, making sure that the whole formula system is correct and consistent. The issue of readability has not been addressed to any large extent. This is one of the facets of structuring that has been studied in this project.

It is our belief, that formally defining Ada is only a worthwhile task to perform, if a large group of people is able to use the definition.

Our good luck has been, that through the last years many more people have become familiar with the notion and uses of formal definitions. Some of the driving force has been the complex problems found in the development of large sofware systems and the users' needs for proven programs, as software move into more and more vital positions of our society. Formal methods provide a tool for analyzing and building such complex systems and some industrial expirience has already been reported on.

Therefore some of the studies laid down in the task of structuring the formal definition of Ada have been in the area of finding out how humans read the formal definition, and what may be done in order to make sure that the reader gets the easiest access to the definition.

In this work, many parameters have been looked into. Some of the parameters have been: what about the size of the reports? model oriented vs. axiomatic descriptions, direct semantics style vs. continuations.

The answer has not always been straightforward, but we believe that we have made the tradeoffs in such a way, that most people with a programming background and a little formal training added, should be able to read and understand the formal definition of Ada.

In the structuring of documents used in this project, each formula has been put into a fixed framework giving the auxiliary information needed in order to read that particular formula. This information includes:

- Identification which directly relates the formula to the RM thereby helping people to understand the formal definition in Ada terms.

- Short description of the objective of the formula.

- The formula itself given either axiomatically or model oriented. As model oriented is believed to be the most readable for computer programmers (it resembles a program) most of the definition is described in a functional style. If a number of concepts can be separated out into a selfcontained abstract data type, it has been done and in many cases the operations performed are described using axioms.

- Natural language explanations of how the formula is supposed to perform its task, and correlation of the formula to the concepts of the RM that the formula describes.

- An extensive cross referencing.

Examples of the above may be found in [ref DDC and CRAI 1986].

Structure of the Static Semantics of Ada.

The subset static semantics of Ada is a homomorphism from the algebra of syntax into the algebra of booleans since separate compilation and hence libraries are not part of the subset. This homomorphism makes heavy use of operations from abstract data types being able to extract information from the program text taken into account until the current point of interrest.

As a mean of breaking the static semantics into useable pieces, the foundation is a hierarchy of abstract data types each aimed at describing an essential Ada concept.

Splitting a definition into data types describing concepts which are carefully highlighted in the RM seems to give the definition two properties: one is that the definition gets broken into manageable size definitions which may be combined, and the other is that breaking the definition into data types which define Ada concepts will give the user who knows about programming languages (maybe even about Ada) a conceptual framework within which to understand the formal definition - facilitating familiarization with and enhancing readability of the definition.

The hierarchy of data types defined, has the following properties: at the bottom of the hierarchy: very basic data types describing integers, identifiers etc. Next level describes types and the strong typing concepts of Ada. This includes operations for the handling of derived types, subtypes, type matching etc. From this data type a new data type is built describing the properties of all entities in Ada which you may declare.

In the same fashion concepts like visibility, overloading, and generics are described in abstract data types in further levels of the hierarchy. The topmost data type is called SUR abbreviated from surroundings. This data type describes the 'static history' of the compilation unit so far, by combining all information from lower level data types. This is done, in order to assemble all static semantics information in one place.

The data types are used in the formation of the homomorphism from the algebras of syntax. This homomorphism is named the well-formed (wf) function(s).

In the subset the 'root construct' is the subprogram body. The type of the function is_wf_Subprogram_body is:

$$\text{Subprogram\_body} \rightarrow \text{SUR} \rightarrow \text{BOOL}$$

but often the check, that a given construct is well formed cannot be performed if the only fact known about the subconstructs is whether they are wellformed or not. Further retrieving of information about the subconstructs is necessary. As an example take the assignment statement: the left-hand side has to be well formed, the right-hand side has to be well formed, but on top of that, the types of the two sides have to be the same. As an is-wf function only returns BOOL, data type operations and auxiliary functions have to be used in order to retrieve the type information from both sides.

Structure of the Dynamic Semantics of Ada.

The dynamic semantics of Ada is modelled using the SMoLCS (Structured Monitored Linear Concurrent Systems) method as described in [Astesiano et al 1985].

Using the SMoLCS method already imposes some structuring on the formal definition of the dynamic semantics. SMoLCS is a layered approach to the description of concurrency. It consists of four layers. At the bottom describing the basic states possible in the system we find a labelled transition system similar to the ones found in for example CCS.

In order to describe the behaviour of the concurrent system, some constraints are applied to the transition system. These constraints fall into three types. First all actions which may result as synchronized operations of processes are identified, next all synchronized actions which may occur in parallel are identified, and the last step defines which actions are possible in the system as a whole.

The above levels constitute what we call step 2. Step 1 of the dynamic semantics, which is using a denotational style is the homomorphism from the algebra of syntax into the semantic algebra defined by step 2. As the metalanguage makes it possible to axiomatically define operations which closely match Ada concepts, the issue is what to define denotationally.

The problem has been solved by structuring the definition of dynamic semantics in such a way, that all concepts described in the RM are defined in denotational clauses, so that no concept of Ada is hidden in an abstract data type.

An argument for moving the concepts from the denotational part could be, that a definition may be written more abstractly by moving some Ada concept modelling out of the denotational part, but for the reason of understanding by the user, it seems more appropriate to split as described above.

A further advantage of the SMoLCS method is the high degree of parameterization. This is used to describe some of the features that previously have been very difficult to describe. These sorts of concepts include implementation dependent features. They may now be modelled by including the appropriate parameters in the definition. A further concept is context clauses. Also here the parameterization scheme helps [ref DDC and CRAI 1986].

Conclusion and Further Work.

The formal definition of the subsets mentioned has assured us, that the task of formally defining the language Ada as described in the RM is feasible and can be done.

During the work with the trial definition we have seen, that in the static semantics the abstract data types had a tendency to become rather large. The problem is overcome by splitting some of them into smaller data types. This is almost also a prerequisite for the second change: the axiomatic modelling of the data types. Currently they are defined by giving a specific model, but breaking the data types into smaller definitions makes an axiomatic definition feasible.

In the dynamic semantics the distinction between operations defined axiomatically and denotational formulae will be studied further. It seems as if the optimal solution (whatever this may be) has not been found yet.

Finally, for both sorts of semantics, some ways of modularizing formulae is needed in order to enhance the readability. The static semantics already to some extent is modularized, but more is needed and the dynamic semantics need more modularizing in step 1. Furthermore, the formal definition has to be updated w.r.t. the commentaries from the Language Maintenance Committee, a task which is timeconsuming and not always straightforward.

References.

Astesiano et al 1985
E. Astesiano, G. F. Mascari, G. Reggio, M. Wirsing
On the Parameterized Algebraic Specification of Concurrent Systems.
TAPSOFT Conf., Berlin
Springer Verlag
Lecture Notes in Computer Science, vol 185, 1985

Bjørner and Oest 1980
D. Bjørner, Ole N. Oest
Towards a Formal Description of Ada
Springer Verlag
Lecture Notes in Computer Science, vol. 98, 1980

Bjørner and Jones 1982
Dines Bjørner and Cliff B. Jones
Formal Specification and Software Development
Series in Computer Science, Prentice Hall 1982

DDC and CRAI 1986
E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, K. W. Hansen, P. Inverardi, E. W. Karlsen, F. Mazzanti, G. Reggio, J. Storbank Pedersen, E. Zucca
Static Semantics of a 'Difficult' Example Ada Subset, and
Dynamic Semantics of a 'Difficult' Example Ada Subset
1986

INRIA 1982
Honeywell inc., Cii Honeywell Bull, and INRIA
Formal Definition of the Ada Programming Language
1982