Annales Univ. Sci. Budapest., Sect. Comp. 10 (1989) 67-82

FORMAL SEMANTICS OF ANNA PACKAGES

VALÉRIA NOVITZKA

Institute of Computer Science
Technical University
040 01 Kosice, B.Nemcovej 3
Czechoslovakia

Abstract. In this paper we define the formal semantics of Anna package. It is shown that Ada construct package together with Anna annotations provide a good tool for describing abstract date types by algebraic specification methods. The semantic approach is based on the concept of heterogeneous algebras using the principles of denotational semantics.

1. Introduction

Anna /Annotated Ada/[1] is a language extension of Ada with two formal comments that are the virtual Ada text and the annotations. Anna has been designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs.

Annotations should be well suited for different possible applications during the life cycle of a program. Such applications include specification of program parts during the requirement analysis and program design. Annotations provide us to specify abstract data types by algebraic specification methods in the form of Anna packages.

We suppose that there may be a support environment mapping Anna specifications into Ada implementations similar as in CIP project [2]. We define the semantics of Anna packages as a mixture of denotational and predicative specifications using algebraic methods, and we can study the problem of consistency of implementation and specification.

2. Algebraic specification of abstract data types

A signature Σ is a pair $\Sigma = (S, 0)$ where S is a set of sorts and 0 is a set of function symbols $f : s_1 x \ldots x s_n \rightarrow s_{n+1}$, $s_i \in S$, $1 \leq i \leq n+1$. A specification of abstract data type T is a pair $T = (\Sigma, E)$ where E is a set of Σ-formulas /axioms/. If the axioms are the equations, we say about equational algebraic specification of abstract data types. [3]

A model for an algebraic specification $T = (\Sigma, E)$, where $\Sigma = (S, 0)$ is a signature, is a heterogeneous Σ-algebra A=(M,F), such that $M = (M_s)_{s \epsilon O}$, $F = (F_f)_{f \epsilon 0}$ and

1. every carrierset $M_s \in M$ is associated with a sort $s \in S$
2. every function $F_f \epsilon F$, $F_f : M_{s_1} x \ldots x M_{s_n} \rightarrow M_{s_{n+1}}$ is associated with a function symbol $f \epsilon 0$, $f : s_1 x \ldots x s_n \rightarrow s_{n+1}$.
3. every axiom $e \epsilon E$ is fulfilled after substituting every occurrence of function symbol from 0 by the corresponding functions from F and variables of sorts from S by the corresponding elements of carriersets from M.

A signature $\Sigma = (S, 0)$ is called hierarchical, if its subsignature $\Sigma' = (S', 0')$, $S' \subseteq S$, $0' \subseteq 0$ is designated to be primitive. A Σ-algebra A=(M,F) is called hierarchical if its signature Σ is hierarchical. A specification of hierarchical abstract data type $T = (\Sigma, E)$ consists of

1. a hierarchical signature Σ with a primitive subsignature Σ' and
2. a set of Σ-axioms E, where $E' \subseteq$ E is a subset of Σ'-axioms such that $T' = (\Sigma', E')$ forms an abstract data type. We call T' a primitive subtype of T.

The Ada construct package [4] together with Anna annotations provide the tool for algebraic specification of abstract data types. A package is provided in two parts: a package specification and a package body. The package specification consists of visible part, where we can define a sort as a private type declaration, its operations that are usable outside this package and axiomatic annotation defining the properties of operations on the private type. The hierarchical concept of private type can be specified by use clause. Hence the visible part of package specification contains the specification of abstract data type, i.e. it defines sorts, operations and axioms of the abstract data type. The visible part must incorporate all the necessary requirements of the user of the package. It must contain all these properties that the user will be able rely on since they are guaranteed to hold in every implementation. On the other hand, the visible part of a package must be loose enough so that the implement or still has freedom for design decision in its implementation.

Example:

```
package RATIO_PACK is

    use Boolean_Type, Integer_Type;

    type RATIONAL is private;

    function equal(x: RATIONAL; y: RATIONAL) return

                Boolean;

    function create (NUMERATOR: Integer;DENOMI-

                NATOR: Integer) return RATIONAL;

    function NUMERATOR (x: RATIONAL) return Integer;

    function DENOMINATOR (x: RATIONAL) return
```

```
                                                        Integer;
    --| axiom for all x: RATIONAL; y: RATIONAL; i: Integer;
    --|                  j:Integer ⇒
    --| equal (x,y) = (Integer_Type.equal (NUMERATOR (x),
    --|              NUMERATOR (y)) and Integer_Type.equal
    --|            DENOMINATOR (x), DENOMINATOR (y))),
    --| NUMERATOR (create (i,j)) = i,
    --| DENOMINATOR (create (i,j)) = j;

    private
    ---- abstract implementation
end RATIO_PACK;
```

The private part of package specification contains the definition of the structure of the private type. Because the structure of a private type is given by using other types we call the private part of a package specification the abstract implementation of the type. While the specification in the visible part is loose, i.e. it allows more than one model /polymorphic/, the private part will the implementator restrict to only one model /monomorphic/ up to isomorphism.

The package body contains the function bodies of operations on private type that was defined in the package specification. We call a package body the implementation part of a package.

Such package difinitions allow firstly to specify an abstract data type by visible part of a package specification. Then we can specify the structure of the type by the private part of this package and then we can specify how to obtain the result values of functions by function bodies in the package body.

3. Semantic concept

We based our semantic approach on the concept of heterogeneous algebras using the principles of denotational semantics [5,6,7].

First we define the universe of semantic objects. A semantic object can be a data object, a function, a carrierset or an algebra. We denote by DatObj a set of all possible data objects. This set contains a special object $\perp$ that represents the results of nonterminating computations. We denote by CarSet a set of carriersets associated with sorts. The elements of carriersets are data objects and every carrierset contains the element $\perp$.

$$\text{CarSet} = \{M : M \subseteq \text{DatObj} \wedge \perp \epsilon M\}$$

CarSet contains the special carrierset B that is associated with the sort Boolean of truth values

$$B = \{true, false, \perp\}$$

We denote by Fnct a set of strict functions between carriersets

$$Fnct = \big\{F : F \epsilon [M_1 \text{x} \ldots \text{x} M_n \rightarrow M_{n+1}]_{strict} \wedge \\ \wedge M_i \epsilon CarSet \wedge 1 \leq i \leq n+1\big\}$$

Let Sig be the set of all hierarchical signatures Σ. We denote by Alg the set of all Σ-algebras over DatObj

$$Alg = \left\{ A : A is \Sigma - \text{algebra} \wedge \Sigma \epsilon \text{Sig} \right\}$$

These algebras are computation structures associated with packages.

The universe of semantic objects is an union

$$\text{SemObj} = \text{DatObj} \cup \text{CarSet} \cup Fnct \cup Alg$$

and we assume that these sets are disjoint.

Every semantic object can be denoted by an identifier from the set Ident. The identifiers correspond to the elements of sorts, function symbols and package names. The set Ident contains two special identifiers:result -that denotes a result of a function and private - that denotes a private type.

We extend the definition of semantic objects and identifiers with attributes that contain semantic information. First we introduce category attribute cat ϵ CatAt that classifies to which of four categories /data, sort, function symbol, package/ is the semantic object associated and the identifier belongs. The set of category attributes is

$$\text{CatAt} = \left\{ data, func, sort, pack \right\}$$

We define the set of attributed identifiers as

$$AttId = CatAt \text{x} Ident$$

For semantic objects we introduce the second, hierarchical attribute that indicates a hierarchical structuring of semantic objects. Every semantic object is of form (v,cat,hat), where $v \epsilon$

$SemObj$, $cat \epsilon CatAt$ and hat is a hierarchical attribute explained bellow.

The set of attributed data objects is the set

$$\text{AttDatObj} = \left\{(d, data, s) : d\epsilon\text{DatObj} \wedge s\epsilon\text{Ident}\right\}$$

where s indicates the sort of the data item d.

The set of attributed carriersets is the set

$$\text{AttCarSet} = \left\{(M, sort, p) : M\epsilon\text{CarSet} \wedge p\epsilon\text{Ident}\right\}$$

where the identifier p indicates either a package in which the sort associated with M is defined or a sort such that M is a subset of the carrierset associated with p.

The set of attributed functions is defined as

$$\text{AttFnct} = \Big\{(F, func, < s_1 \ldots s_n s_{n+1} >) :$$

$$F\epsilon\text{Fnct} \wedge s_i\epsilon\text{Ident} \wedge 1 \leq i \leq n+1\Big\}$$

where $< s_1 \ldots s_n s_{n+1} >$ is the sequence of sorts such that F is associated with function symbol f, $f : s_1 \text{x} \ldots \text{x} s_n \rightarrow s_{n+1}$.

To define the set of attributed algebras we have to modify the definition of signature.

Definition:

A generalized hierarchical signature $\overline{\Sigma}$ is a sequence of attributed sorts, function symbols and packages.

The set of generalized hierarchical signatures is defined recursively

$$\mathrm{GSig} = \Big(\begin{array}{l} \{(s, sort, p) : s, p\epsilon \mathrm{Ident}\} \cup \\ \{(f, func, < s_1 \dots s_n s_{n+1} >) : s_i, f\epsilon \mathrm{Ident} \wedge \\ \wedge 1 \leq i \leq n+1\} \cup \\ \{(p, pack, \overline{\Sigma}) : \overline{\Sigma}\epsilon GSig\} \Big)^* \end{array}$$

Now we can define the set of attributed algebras as

$$\mathrm{AttAlg} = \Big\{(A, pack, \overline{\Sigma}) : A \text{ is } \overline{\Sigma} - \text{algebra} \wedge \overline{\Sigma}\epsilon \mathrm{GSig}\Big\}$$

The set of attributed semantic objects is the union

$$\mathrm{AttSemObj} = \mathrm{AttDatObj} \cup \mathrm{AttCarSet} \cup \mathrm{AttFnct} \cup \mathrm{AttAlg}$$

We extend the notion of environment defined in algebraic specifications as follows:

Definition:

The environment ξ is a total mapping

$$\xi : \mathrm{AttId} \rightarrow \mathrm{AttSemObj}$$

defined by

1. for every $(data, x)\epsilon AttId$

$$\xi(data, x) = (d, data, s)$$

where $d\epsilon DatObj$ of sort s;

2.for every $(sort, s) \epsilon AttId$

$$\xi(sort, s) = (M, sort, p)$$

where $M \epsilon CarSet$ and s is defined in package p or s is a subsort of sort p;

3. for every $(func, f) \epsilon AttId$

$$\xi(func, f) = (F, fun, < s_1 \ldots s_n s_{n+1} >)$$

where $F \epsilon Fnct$ and f is its associated function symbol $f : s_1 \mathrm{x} \ldots \mathrm{x} s_n \rightarrow s_{n+1}$;

4. for every $(pack, p) \epsilon AttId$

$$\xi(pack, p) = (A, pack, \overline{\Sigma})$$

where A is a $\overline{\Sigma}$-algebra.

The set of all environments is denoted by Env.

Definition:

Let $\xi \epsilon$ Env be an environment, $cat \epsilon CatAt$ a category attribute and hat a hierarchical attribute. If for $x \epsilon Ident$, $v \epsilon SemObj$

$$\xi(cat, x) = (v, cat, hat)$$

then we denote

$$v = \mathrm{Val}(\xi, cat, x)$$

$$\mathrm{hat} = \mathrm{Atr}(\xi, cat, x)$$

and say that the mapping Val returns the value of the identifier x in the environment ξ and the mapping Atr returns its hierarchical attribute.

4. Semantics of declarations

Every family of declarations specifies a hierarchical $\overline{\Sigma}$-algebra of some generalized hierarchical signature $\overline{\Sigma}\epsilon GSig$. Semantically we interpret every declaration as a predicate on an environment. Some declarations /functions, packages/ consist of two parts: the specification part and the implementation part /body/. We introduce two semantic functions for declarations:

$$\text{Dspec} : \text{declaration} \rightarrow \text{Env} \rightarrow B$$

$$\text{Dimpl} : \text{declaration} \rightarrow \text{Env} \rightarrow B$$

The semantic function Dspec considers only specification parts of declarations and the semantic function Dimp considers both the specification and the implementation parts. Introducing two semantic functions for declarations serves for formulating the criterions for consistency of programs and can serve for verification purposes.

To formulate the semantic equations for declarations we have to introduce several auxiliary functions.

Let $\xi\epsilon$ Env be an environment and $(C, X) \subseteq AttId$ a subset of attributed identifiers such that $C \subseteq CatAt$, $X \subseteq Ident$.
We define the function

$$\text{Accord} : \text{Env}\text{x}\mathcal{P}(\text{AttId}) \rightarrow \mathcal{P}(\text{Env})$$

that for given environment $\xi\epsilon Env$ and for the given set of attributed identifiers $(C, X) \subseteq AttId$ returns the set of all environments according with ξ for all identifiers from (C,X).

Formally

$$\mathrm{Accord}\Big(\xi, (\mathrm{C}, \mathrm{X})\Big) =$$

$$\Big\{\xi' \epsilon \mathrm{Env} : \forall(\mathrm{cat}, \mathrm{x}) \epsilon (\mathrm{C}, \mathrm{X}) \Rightarrow \xi(\mathrm{cat}, \mathrm{x}) = \xi'(\mathrm{cat}, \mathrm{x})\Big\}$$

The second auxiliary function Set

$$\mathrm{Set} : \mathrm{GSig} \rightarrow \mathcal{P}(\mathrm{AttId})$$

associates with given generalized signature the set of all its attributed identifiers and is defined by

$\mathrm{Set}(\varepsilon) = \emptyset$ where ε is the empty sequence

$Set\Big(< (s, sort, p) > \Big) = \Big\{(sort, s)\Big\}$

$Set\Big(< (f, func, < s_1 \ldots s_n s_{n+1} >) > \Big) = \Big\{(func, f)\Big\}$

$Set\Big(< (p, pack, \overline{\Sigma}') > \Big) = \Big\{(pack, p)\Big\}$

$Set(\overline{\Sigma}' o \overline{\Sigma}') = Set(\overline{\Sigma}) \cup Set(\overline{\Sigma}')$

The package names are declared twice: first by the specification part and then by its implementation part /body/. To remove the names declared twice we introduce the function

$$\mathrm{Rem} : \mathrm{GSig} \mathrm{x} \mathcal{P}(\mathrm{AttId}) \rightarrow \mathrm{GSig}$$

that is defined as follows: let $(C, X) \subseteq AttId$ be a subset of attributed identifiers and $\overline{\Sigma} \epsilon GSig$ a generalized hierarchical signature. Then

$$Rem(< (x, cat, hat) > o\overline{\Sigma}, (C, X)) =$$

$$= \begin{cases} \epsilon & if < (x, cat, hat) > o\overline{\Sigma} = \epsilon \\ Rem(\overline{\Sigma}, (C, X)) & if\ cat \in C, x \in X \\ < (x, cat, hat) > o\ Rem(\overline{\Sigma}, (C, X)) & \text{otherwise} \end{cases}$$

This function returns the generalized hierarchical signature where every name is declared explicitly once.

For every declaration we introduce the auxiliary semantic function

$$D : \text{declaration} \rightarrow \text{GSig}$$

that associated with every sequence of declarations the generalized hierarchical signature consisting of the sequence of the attributed sorts, function symbols and package names that are globally declared.

5. Semantic equations for package declaration

We denote by

d a declaration

p a package name

The package declaration consists of two parts: package specification and package body. The abstract syntax of both is

$$ps ::= \underline{\text{package}}\ p\ \underline{\text{is}}\ d\ \underline{\text{end}}$$
$$|\underline{\text{package}}\ p\ \underline{\text{is}}\ d_1\ \underline{\text{private}} d_2 \underline{\text{end}}$$
$$pb ::= \underline{\text{package body}} p \underline{\text{is}}\ d\ \underline{\text{end}}$$

An abstract data type specified by a package p is in the environment $\xi \epsilon$Env associted with a $\overline{\Sigma}$-algebra A such that $\overline{\Sigma}$ is the generalized hierarchical signature declared by the visible part and $(A, pack, \overline{\Sigma}) \epsilon$ AttAlg.

First we introduce the semantic equations for the auxiliary semantic function D:

$$D[\underline{\text{package}}\ p\ \underline{\text{is}}\ d\ \underline{\text{end}}] = < \left(p, pack, D[d]\right) >$$

$$D[\underline{\text{package}}\ p\ \underline{\text{is}}\ d_1\ \underline{\text{private}}\ d_2\ \underline{\text{end}}] = < \left(\text{p,pack,D}[d_1]\right) >$$

$$D[\underline{\text{package body}}\ p\ \underline{\text{is}}\ d\underline{\text{end}}] = \varepsilon$$

The package specification defines the generalized hierarchical signature consisting of attributed package $< \left(p, pack, D[d]\right) >$. The package bodies do not define a signature.

Let $\xi\epsilon$ Env be an environment. The semantic equations for the semantic function Dspec are as follows:

$$Dspec_\xi[\underline{\text{package}}\ p\ \underline{\text{is}}\ d\ \underline{\text{end}}] =$$

$$\begin{aligned}&(Atr(\xi, pack, p) = D[d])\wedge\\&(Val(\xi, pack, p) = A)\wedge\\&Dspec_\xi[d]\end{aligned}$$

where A is the D[d]-algebra associated with p in the environment ξ.

$$Dspec_\xi[\underline{\text{package}}\ p\ \underline{\text{is}}\ d_1\ \underline{\text{private}}\ d_2\ \underline{\text{end}}] =$$

$$\begin{aligned}&\left(Atr(\xi, pack, p) = D[d_1]\right)\wedge\\&\left(Val(\xi, pack, p) = A\right)\wedge\\&Dspec_\xi[d_1]\end{aligned}$$

The declarations in the private part do not extend the generalized hierarchical signature D$[d_1]$.

$$Dspec_\xi[\underline{\text{package body}}\ p\ \underline{\text{is}}\ d\ \underline{\text{end}}] = true$$

The package body does not contribute to the specification part.

The semantic equations for the semantic function Dimpl are as follows:

$$\begin{aligned} Dimpl_\xi[\underline{\text{package}}\ p\ \underline{\text{is}}\ d\ \underline{\text{end}}] = &\Big(Atr(\xi, pack, p) = D[d]\Big) \wedge \\ &\Big(Val(\xi, pack, p) = A\Big) \wedge \\ &Dimpl_\xi[d] \end{aligned}$$

$$\begin{aligned} &Dimpl_\xi[\underline{\text{package}}\ p\ \underline{\text{is}}\ d_1\ \underline{\text{private}}\ d_2\ \underline{\text{end}}] = \\ &= \Big(Atr(\xi, pack, p) = D[d_1]\Big) \wedge \\ &\Big(Val(\xi, pack, p) = A \wedge \Big) Dimpl_\xi[d_1] \wedge \\ &\Big(\exists \xi_1 \epsilon Accord(\xi, \mathrm{Set}(D[d_2]) \setminus \mathrm{Set}(D[d_1])) : \\ &Dimpl_{\xi_1}[d_2]\Big) \end{aligned}$$

Here we must take into account the declaration d_2 in the private part. ξ_1 is the environment according with ξ and all attributed identifiers declared in d_2 but not in d_1.

$$\begin{aligned} &Dimpl_\xi[\underline{\text{packagebody}}\ p\ \underline{\text{is}}\ d\ \underline{\text{end}}] = \\ &\Big(\exists \xi_1 \epsilon Accord(\xi, Set(D[d]) \setminus Atr(\xi, pack, p)) : \\ &Dimpl_{\xi_1}[d]\Big) \end{aligned}$$

where ξ_1 is the environment according to ξ and such identifiers declared by d that differ from identifiers in signature $Atr(\xi, pack, p)$.

References

[1] **Luckham, D.C.,et al:** Anna a language for annotating Ada programs; Technical Report 84-248, Stanford University,1984

[2] **Bauer, F.L.:** The Münich project CIP; LNCS 183,- Springer-Verlag, 1985

[3] **Ehrig, H., et al:** Fundamentals of algebraic specification 1; EATCS Monographs on Theoretical Computer Science; Springer-Verlag, 1985

[4] **The programming language Ada reference manual;** LNCS 155, Springer-Verlag,1983

[5] **Stoy, J.E.:**Denotational Semantics: The Scott-Strachey approach to programming language theory; MIT Press, 1977

[6] **Broy, M., et al:** Semantics; Technical Report, University Passau, 1987

[7] **Tennent, R.D.:** A practical guide to denotational semantic definitions; Technical Report, University of Oxford, 1978

Received: December 7, 1988.