The algebraic specification of annotated abstract syntax trees

M.G.J. van den Brand
Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ, Amsterdam, The Netherlands
email: markvdb@fwi.uva.nl

C. Groza
Department of Computer Science, Technical University of Timişoara
Bd. V. Pârvan 2, 1900 Timişoara, România
email: calin@utt.ro

Abstract

This paper presents an algebraic specification of annotated abstract syntax trees. The goal is to create a flexible mechanism for processing trees used by various tools in programming environments.

The major parts of the specification are: the definition of the abstract syntax of a language, the definition of the abstract data type tree and the specification of a systematic method to create, use, and delete information from the nodes of the tree.

1 Introduction

We consider a programming environment as a collection of independent tools, such as parser, editor, etc., each working on or using abstract syntax trees. The abstract syntax tree is an essential data structures of (generated) programming environments. It enables the user of the environment to perform edit actions in a syntax directed way. Tools in a programming environment may want to store specific information in the abstract syntax tree without knowing (and destroying) the information stored by other ones. We therefore extend abstract syntax trees with so-called *annotations* to store information, like position or information used by attribute evaluation mechanisms. A general mechanism to manipulate abstract syntax trees is presented together with their annotations.

The manipulation of abstract syntax trees used in a generated programming environment can be divided in two parts:

- a generic part for manipulating nodes and links, and
- a language-dependent part describing which trees are *well-formed* with respect to the underlying abstract syntax the so-called *signature* of the language for which the environment was generated.

The signature is derived during the generation of a programming environment. We will not discuss how to derive such a signature, but assume that it is available and can be used by the various tools. Furthermore we assume that the tools in a generated environments do not modify the signature.

We define a set of operations both on the signature (restricted to "read-only" operations), and on the tree. Signature operations are necessary to ensure the *well-formedness* of the abstract syntax trees. Trees operations are known by all tools and can be considered as an interface to abstract syntax trees.

The ASF+SDF Meta-environment [Kli93] is an example of a programming environment generator. It is a sophisticated interactive environment for the development of programming environments. It consists of a collection of cooperating syntax directed editors for the various parts of the language for which a programming environment must be generated. The ASF+SDF formalism permits syntactic and semantic specification of languages. It consists of two formalism ASF [BHK89] (Algebraic Specification Formalism) and SDF [HHKR89] (Syntax Definition Formalism). Furthermore it supports modularity.

2 Specification of signatures

The abstract syntax of a language is defined using a similar approach as in the algebraic specifications of abstract data types. The algebraic specification of the abstract data type BOOLEAN, for example, consists of a collection of sorts, functions, and equations. The sorts and functions form the signature of the specification:

```
sorts
  BOOL
functions
  TRUE:                          -> BOOL
  FALSE:                         -> BOOL
  AND: BOOL # BOOL               -> BOOL
  OR: BOOL # BOOL                -> BOOL
  NOT: BOOL                      -> BOOL
```

This specification can also be seen as a representation for the abstract syntax of the boolean language. It is possible to define the abstract syntax for any language in this way. From this perspective, we extend the use of the notion of signature. We will say that the abstract syntax of a language is represented by the signature of the language which is a collection of sorts and associated functions.

The formal specification of sorts, functions, and signatures is given below. We assume that the specification of basic entities: identifiers (`ID`), integers (`INT`), booleans (`BOOL`) and strings (`STRING`) are available. The constructor functions for `SORT`, `FUNC`, and `SIGN` are:

```
context-free syntax
  ID                                   -> SORT
  ID ":" { SORT "#"}* "->" SORT        -> FUNC
  "sorts"  SORT*
  "functions" FUNC*                    -> SIGN
```

In addition we define the access functions:

```
  name(SORT)                           -> ID
  name(FUNC)                           -> ID
  domain(FUNC)                         -> SORT-LIST
  range(FUNC)                          -> SORT
```

We use the following tables for storing sorts and functions:

```
  "(" SORT* ")"                        -> SORT-TABLE
  "(" FUNC* ")"                        -> FUNC-TABLE
  "<" SORT-TABLE ";" FUNC-TABLE ">"    -> SIGN
```

The names of sorts and functions are used as keys in these tables. The following access functions retrieve components of the signature by name:

```
  sort(SIGN, ID)                       -> SORT
  func(SIGN, ID)                       -> FUNC
```

Although the present specification is sufficient to describe the abstract syntax of a language, it still too limited.

1. All functions have a fixed arity. This results in an abstract syntax tree in which every node has a fixed number of children. However, in many languages there are constructs with variable number of items, for example, a compound statement contains a variable number of statements. Although a representation in a fixed arity tree is possible, in these cases, it is more convenient to have a representation in which the nodes can have a variable number of children.

2. In some cases unary functions can be left out. For example, in arithmetic expressions all the natural constants are expressions as well. In the specification of abstract syntax there must be a unary function to express this relation:

   ```
   nat2expr: NAT           -> EXPR
   ```

   and the abstract tree will therefore contain extra nodes. These nodes do not add structural information and can be left out.

These two features lead to an extension of the specification of signatures. First, it must be possible to define sorts as a list of other sorts. Functions therefore may have a variable number of arguments and as a consequence a node in an abstract syntax tree can have a variable number of children. Second, we introduce a binary relation between sorts. We say that S1 is a *subsort* of S2 if S1 can be used when S2 is required. This relation avoids the definition of functions from S1 to S2 and eliminates the corresponding nodes in the abstract syntax tree.

As an example we will consider the signature of a language for arithmetic expression:

```
sorts
  INT ID EXPR
subsorts
  INT < EXPR, ID < EXPR
functions
  plus: EXPR # EXPR                -> EXPR
  times: EXPR # EXPR               -> EXPR
  exp: EXPR # INT                  -> EXPR
  max: list(EXPR)                  -> EXPR
  min: list(EXPR)                  -> EXPR
```

The sort specification of the abstract syntax with the new features becomes:

```
  ID                               -> SORT %%constructors
  "SORT-ERROR"                     -> SORT
  list(SORT)                       -> SORT
  SORT "<" SORT                    -> SUBSORT-REL
  name(SORT)                       -> ID   %%access functions
```

The specification of functions:

```
  ID ":" { SORT "#" }* "->" SORT    -> FUNC %% constructors
  "<" ID ";" SORT-LIST ";" SORT ">" -> FUNC
  "FUNC-ERROR"                      -> FUNC
  name(FUNC)                        -> ID   %% access functions
  domain(FUNC)                      -> SORT-LIST
  range(FUNC)                       -> SORT
```

A more complete specification of signatures is:

```
  "sorts" SORT*
  "subsorts" { SUBSORT-REL ","}*
  "functions" FUNC*                   -> SIGN %% constructors
  "(" SORT* ")"                       -> SORT-TABLE
  "(" SUBSORT-REL* ")"                -> SUBSORT-REL-TABLE
  "(" FUNC* ")"                       -> FUNC-TABLE
  "<" SORT-TABLE ";" SUBSORT-REL-TABLE ";"
                     FUNC-TABLE ">"   -> SIGN
  sort(SIGN, ID)                      -> SORT %% access functions
  func(SIGN, ID)                      -> FUNC
```

3 Specification of general trees

We introduce a general notation for trees and a mechanism for storing arbitrary information in the nodes. Every node contains data called *annotations* and there is a systematic way to create, use, and delete it. The children of a node in the tree form an ordered set. The relative position among children is significant. We will call these trees *annotated ordered trees*.

In this section we focus on the structure of the tree and we will refer to the data stored in nodes as objects of sort ANNOT. The specification of ANNOT can be found in Section 4.

We use a recursive definition of trees: data represented as annotations forms a (single-node) tree and data extended with a list of trees forms a tree as well.

```
context-free syntax
  "[" ANNOT "]"                       -> TREE
  "[" ANNOT TREE-LIST "]"             -> TREE
  "(" TREE* ")"                       -> TREE-LIST
```

The specification contains functions to retrieve the structure information:

```
  children(TREE)                 -> TREE-LIST
  child(TREE, INT)               -> TREE


equations
  [tree-01] children([Annot]) = ()
  [tree-02] children([Annot TreeList]) = TreeList
  [tree-03] child(Tree, Int) = position(children(Tree), Int)
```

and functions to retrieve and store information in a node:

```
context-free syntax
  annot(TREE)                     -> ANNOT
  annot(TREE, ANNOT)              -> TREE


equations
  [tree-04] annot([Annot]) = Annot
  [tree-05] annot([Annot TreeList]) = Annot
  [tree-06] annot([Annot], Annot') = [Annot']
  [tree-07] annot([Annot TreeList], Annot') = [Annot' TreeList]
```

These functions offer the basic functionality for annotated ordered trees. Some applications, however, require the possibility to point to a node in the tree. For example, in a structure editor it must be possible to focus on a specific node.

In order to offer this facility, we extend our specification of trees with a new data type called *focus*. The focus is a list of integers which represents the path in the tree from the root to the "target" node. Trees and focus form the data type *focused trees*. The specification of focused trees contains the following constructors and functions:

```
context-free syntax
  ftree(TREE, FOCUS)           -> FTREE  %% constructors
  "<" TREE ";" FOCUS ">"       -> FTREE
  tree(FTREE)                  -> TREE   %% "get" the tree
  focus(FTREE)                 -> FOCUS  %% get focus
  focus(FTREE, FOCUS)          -> FTREE  %% set the focus
  up(FTREE)                    -> FTREE  %% go to the parent
  down(FTREE)                  -> FTREE  %% go to the first child
  next(FTREE)                  -> FTREE  %% go to next sibling
  prev(FTREE)                  -> FTREE  %% go to previous sibling
  root(FTREE)                  -> FTREE  %% go to root
```

Focused trees are organized as tuples with a tree field and a focus field. Tuples are often used in the specification to create new data structures. We present how a tuple in general is specified. These tuples are used at several places in the specification.
The syntax of tuples is:

```
context-free syntax
  tuple(FIELD1, FIELD2)        -> TUPLE   %% constructor
  "<" FIELD1 ";" FIELD2 ">"    -> TUPLE
  "TUPLE-ERROR"                -> TUPLE
  field1(TUPLE)                -> FIELD1  %% field1 - get
  field1(TUPLE, FIELD1)        -> TUPLE   %% field1 - set
  field2(TUPLE)                -> FIELD2  %% field2 - get
  field2(TUPLE, FIELD2)        -> TUPLE   %% field2 - set
```

The corresponding equations for tuples are:

```
equations
  [tuple-01] tuple(F1, F2)= < F1 ; F2 >
  [tuple-02] field1(< F1 ; F2 >)= F1
  [tuple-03] field1(< F1 ; F2 >, F1')= < F1' ; F2 >
  [tuple-04] field2(< F1 ; F2 >)= F2
  [tuple-05] field2(< F1 ; F2 >, F2')= < F1 ; F2' >
```

Note the form of the access functions `field1` and `field2`. The same form will be used in the specification of annotations later. Thus, we provide a uniform way to access the data of an object.

Data structures organized as tuples or tables are used several times in the specification. In order to increase the reusability of the code we have defined and used a tool which permits instantiation of generic modules, a facility which is not offered by the ASF+SDF Meta-environment. For example, all tables are generated using a generic module for tables which is instantiated for sorts, functions, etc. Another generic module was used for tuples. It was instantiated for signature, focused trees, etc.

4 Annotations

Annotations are a mechanism to store information in the nodes of a tree. Each *annotation*, consists of a *tag*, and a *value*. There are two operations to access annotations `set` and `get`. The structure and the values of annotations depend on the application. For example, the annotations can be used to store attribute values if the abstract syntax trees are used in attribute grammars. In structured editors the annotations can be used to store position information from the source text in the nodes of abstract syntax tree.

It should be possible to retrieve values from or store values in a node. But another important feature that should be provided is the possibility to add new annotations or to delete annotations from the set of annotations associated with a node. There are several reasons for having a dynamic set of annotations. If static annotations are used and a new tool, added to the programming environment, wants to store information in the tree, then the specification of trees as well as the specifications of all other tools must be changed. Tools are more independent of each other when dynamic annotation are used. It is also more efficient in the amount of memory required to store information: annotations which are no longer valid can be deleted. The disadvantage is some loss of execution speed.

ASF+SDF only provide a static type system. New sorts cannot be created dynamically. In order to permit dynamic definition of annotations a simple dynamic type system must be simulated. All dynamic annotations have the same type `DANNOT` in the specification, but may represent different types of information in the application.

A similar situation occurs in database systems. In the *relational data model* it is possible to create tuples but their structure cannot be changed during run-time. Another model, called the *functional data model* [Gra85] permits dynamic change of the structure. In this model the data is only visible through access functions. New access functions can be defined dynamically and can be used to store information of a new type in the database.

The annotation system requires a compromise between efficiency, in terms of access speed to the stored information, and flexibility expressed by the possibility to create new types of data. This is solved by choosing a mixed data model. Annotations can be either static or dynamic. The operations on static annotations are `set` and `get`. Dynamic annotations use the operations `set`, `get`, `add` and `remove`. The last two operations create and delete annotations in nodes.

Inspired by the access functions used in the specification of tuples we will use the following form for the *set* function:

```
tag ( Tree, Value )
```

which sets the annotation `tag` of the root of the tree `Tree` to the value `Value`. The form of the *get* function is:

```
tag ( Tree )
```

This function returns the value of the annotation `tag` of the root of `Tree`.

4.1 Specification of annotations

Annotations, in general, are defined as a tuple with one field for every static annotation plus a field, which is a table, for all dynamic annotations. If SA_i, $1 < i < n$ are static annotations and DA_j, $1 < j < m$ are dynamic annotations, then the data is stored in the node in the form:

$$\langle SA1;\ SA2;\ \ldots\ SA_n;\ (DA_1\ DA_2\ \ldots\ DA_m)\rangle$$

We will use the name *decor* for dynamic annotations. A decor is defined as a pair, *tag* and *value*. So, the dynamic annotation DA_j has the form:

$$\langle T_j; V_j\rangle$$

where T_j is the tag and V_j is the value of the decor. The tags are of sort `ID` and the values of sort `VALUE`. Conversion functions are defined and used to store data of a new type as decor.

In case of annotated trees the static annotations are: `func` – the function name in the abstract syntax corresponding to a node and `type` – the name of the sort which represents the type of the node. The constructor of annotations is:

```
context-free syntax
  "<" ID ";" ID ";" DANNOT-TABLE ">" -> ANNOT
```

The access functions for static annotations are:

```
func(ANNOT)          -> ID    %% retrieve func
func(ANNOT, ID)      -> ANNOT %% set  func
type(ANNOT)          -> ID    %% retrieve type
type(ANNOT, ID)      -> ANNOT %% set type
```

The access functions to retrieve and store the set of dynamic annotations are respectively:

```
dannot(ANNOT)                -> DANNOT-TABLE %% get
dannot(ANNOT, DANNOT-TABLE)  -> ANNOT        %% set
```

The semantics of these functions is similar to that of functions `field1` and `field2` presented for tuples in Section 3.

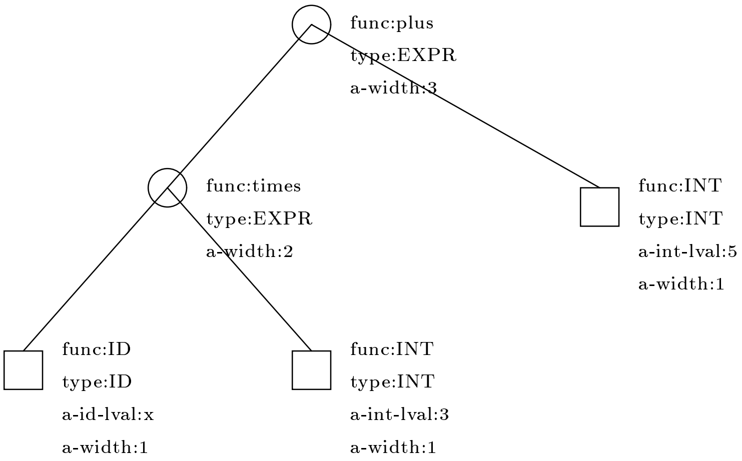

Figure 1: An example of annotated abstract syntax tree

4.2 Access functions for dynamic annotations

The obvious reasons for having access functions for the dynamic annotations is hiding the implementation and making a clear interface, but another important reason is hiding the "non-typed" specification of decors (all the decors have the same type $\langle ID; VALUE\rangle$). Well-defined access functions eliminate this drawback and assure a consistent use of dynamic annotations. We will present a set of functions to operate on all the decors and a method to define functions for specific decors.

A number of functions are defined for operations on decors. A new decor for a tree is created with `add-decor` and deleted with `remove-decor`. The functions `set-decor` and `get-decor` are used to modify the value of a decor or to retrieve the current value respectively.

```
context-free syntax
  add-decor(TREE, ID, VALUE)       -> TREE
  remove-decor(TREE, ID)           -> TREE
  set-decor(TREE, ID, VALUE)       -> TREE
  get-decor(TREE, ID)              -> VALUE
  exist-decor(TREE, ID)            -> BOOL
```

These functions affects only the root of the tree given as argument. The next two functions add and remove dynamic annotations from all the nodes of a tree:

```
add-decor-rec(TREE, ID, VALUE)-> TREE %% set recursively a decor
remove-decor-rec(TREE, ID)    -> TREE %% remove recursively a decor
```

Conversion functions are required when we want to put an integer, boolean, or whatever values type in the annotation. These functions have the syntax:

```
  cast-DECOR-TYPE(VALUE)-> DECOR-TYPE
  value(DECOR-TYPE)     -> VALUE
```

These functions are defined for the sorts `ID`, `INT` and `STRING` and must be defined for every new sort used for dynamic annotations.

Let us consider the annotated abstract syntax tree shown in Figure 1. The annotations of every node are written in the format `tag:value`. Tags of dynamic annotations start with "a-".

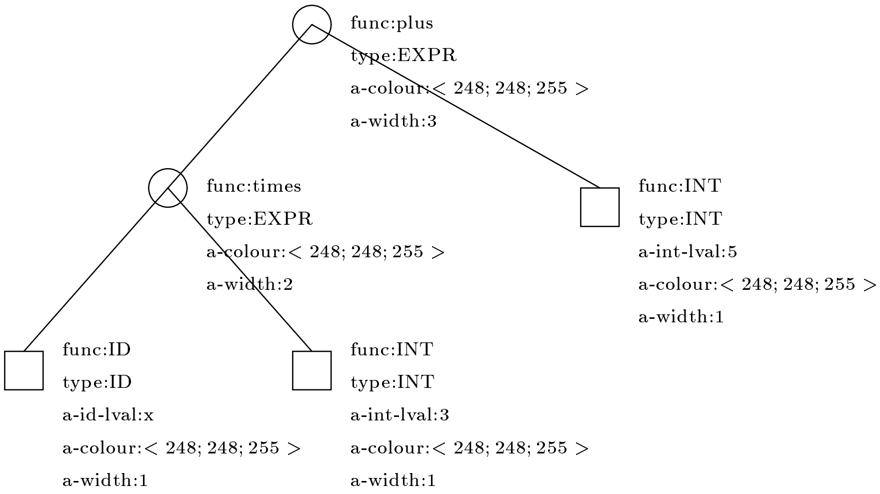

Figure 2: The annotated abstract syntax tree after adding the dynamic annotation "colour"

Let us consider a tool which uses a new decor called "colour" of sort COLOUR. The colours are defined by three natural numbers representing the red, green and blue components (this representations is used in the X Windows environment). First, we define conversion functions between the sorts COLOUR and VALUE.

```
sorts COLOUR
context-free syntax
  "<" INT ";"  INT ";" INT ">"    -> COLOUR %% constructor
  cast-COLOUR(VALUE)              -> COLOUR %% conv. functions
  value(COLOUR)                   -> VALUE
variables
  Colour                          -> COLOUR
```

Function `value` is a constructor. Function `cast-COLOUR` is defined by the following equation:

```
equations
  [colour-01] cast-COLOUR(value(Colour )) = Colour
```

Using these functions, it is possible to add the decor "colour" with the value $< 248; 248; 255 >$ (ghost white) in the nodes of the abstract syntax tree by a call to the function `add-decor-rec`:

```
add-decor-rec(Tree, a-colour, value(<248;248;255>))
```

The result of this operation is a new tree which has a new decor called "colour" in every node. It is shown in Figure 2. The value of this decor in the first child of the root can be retrieved using the function `get-decor`:

```
cast-COLOUR(get-decor(child(Tree,1), a-colour))
```

The functions for operations on dynamic annotations presented so far offer the functionality required to process annotations but have two disadvantages:

- the user must use often the conversion functions to pass parameters and to retrieve results. This makes the syntax complex;

- there is no checking of the correspondence between the tag of a decor and its value. For example, it is syntactically correct a call like:

  ```
  add-decor-rec(Tree, a-colour, value("ABRACADABRA"))
  ```

  but this is not semantically correct.

In order to avoid these drawbacks we introduce for the new decor a set of functions on top of the existing ones:

```
context-free syntax
  colour(TREE, COLOUR)     -> TREE   %% add/set the decor colour
  colour-rec(TREE, COLOUR)-> TREE   %% add/set recursively
  colour(TREE )            -> COLOUR %% retrieve colour
```

The definitions of these functions uses the "low-level" functions (`add-decor`, `add-decor-rec` and `get-decor`) defined for operations on decors.

```
equations
  [c-11] colour(Tree, Colour) = add-decor(Tree,
                       a-colour, value(Colour))
  [c-12] colour-rec(Tree, Colour) =add-decor-rec(Tree,
                       a-colour, value(Colour))
  [c-13] colour(Tree) = cast-COLOUR(get-decor(Tree, a-colour))
```

The decor "colour" can now be added to all the nodes using:

```
colour-rec(Tree, <248;248;255>)
```

and the decor of the first child of the root can be obtained by the call:

```
colour(child(Tree, 1))
```

Besides the fact that these new functions eliminates the disadvantages of the "low-level" functions a nice feature is that they have the same syntax as the access functions to static annotations. Thus, once the user has defined these functions, at the user interface level there is no distinction between static and dynamic annotations.

The discussion presented so far for the annotation *colour* can be generalized in a similar way as for tuples in Section 3. For every new decor used by a tool, a set of specific access functions is defined. In the following specification `decor` is the name of the new decor and `DECOR-TYPE` is its type. The first function adds a decor or, changes the value of an existing one in the root of the tree. The second function traverses the tree and adds or updates the decor in all the nodes of the tree.

```
context-free syntax
  decor(TREE, DECOR-TYPE)        -> TREE %% set
  decor-rec(TREE, DECOR-TYPE)    -> TREE %% set recursively
```

The "get" function returns the decor called `decor` from the root of the tree given as argument:

```
  decor(TREE)                    -> DECOR-TYPE
```

Similar functions are defined for operations on annotations. They are used in tree traversal to operate on the data stored in nodes. The first function adds or updates a decor in the annotation:

```
  decor(ANNOT, DECOR-TYPE)       -> ANNOT      %% store
```

A second function is used to retrieve a decor from an annotation:

```
  decor(ANNOT)                   -> DECOR-TYPE %% retrieve
```

Note that function names are overloaded. The same name is used for functions which operate on trees as for functions which operates on annotations.

4.3 Tree traversal and updating of annotations

The values of new or existing decors are computed in different ways, depending on what they represent. A general function to update recursively annotations cannot cover the diversity of ways to compute the values for annotations. This operation is specified by combining tree traversal, computations and the setting of annotations.

There are two situation which can occur:

- all nodes of a tree must be updated;
- a specific collection of tree nodes must be updated.

We give an example of first case. The new information stored in every node is the `width` of the subtree. The inductive definition for `width` is: every leaf has the width 1 and every non-leaf node has the width equal to the sum of the width of its children. This annotation is used in Section 5.2 to generate a graphical representation for trees.

The specification of the decor `width` has two parts. The first one defines the functions associated with the new decor using the method presented in the previous section. The tag is “width” and has the type INT .

```
context-free syntax
  width(TREE)                         -> INT
  width(TREE, INT)                    -> TREE
  width(ANNOT)                        -> INT
  width(ANNOT, INT)                   -> ANNOT

equations
  [wd-01] width(Tree) = cast-INT(get-decor(Tree, a-width))
  [wd-02] width(Tree, INT-Val) = set-decor(Tree, a-width,
                                 value(INT-Val))
  [wd-03] width(Annot) = cast-INT(get-decor(Annot, a-width))
  [wd-04] width(Annot, INT-Val) = set-decor(Annot, a-width,
                                 value(INT-Val))
```

The second part contains the definition of function `set-width-rec` which traverses the tree and updates the decor conform the definition for `width`.

```
context-free syntax
  set-width-rec(TREE)                  -> TREE
  set-width-rec(TREE-LIST)             -> TREE-LIST
```

Function `compute-width` computes the width of a list of trees:

```
  compute-width(TREE-LIST)             -> INT
```

The following equations show the traversal and the update of the tree annotations:
The width of leaves is 1:

```
equations
  [wd-05] set-width-rec([ Annot ])= width([ Annot ], 1)
```

The width of an internal nodes is the sum of the width of the children:

```
[wd-06]  Tree = [ Annot TreeList ],
      set-width-rec(TreeList) = TreeList',
      compute-width(TreeList') = Int
      ==============================
      set-width-rec(Tree) = [ width(Annot, Int) TreeList' ]
```

The definition of `set-width-rec` for lists of trees is:

```
[wd-07]  set-width-rec(())= ()
[wd-08]  set-width-rec((Trees)) =(Trees')
       ===================================
       set-width-rec((Tree Trees)) = (set-width-rec(Tree) Trees')
```

The equations for `compute-width` are:

```
[wd-09] compute-width(()) = 0
[wd-10] compute-width((Tree Trees)) =
              width(Tree) + compute-width((Trees))
```

5 Annotated abstract syntax trees

The specification of abstract syntax trees uses the specification of signatures and (annotated) focused trees. The sort for abstract syntax trees is `AST`. It is defined as a tuple formed by a signature and a focused tree:

```
context-free syntax
  astree(SIGN, FTREE)              -> AST  %% constructors
  "<" SIGN ";" FTREE ">"           -> AST
  "AST-ERROR"                      -> AST
  sign(AST)                        -> SIGN %% access functions
  sign(AST, SIGN)                  -> AST
  ftree(AST)                       -> FTREE
  ftree(AST, FTREE)                -> AST
```

The equations are similar to those presented for tuples in Section 3.

5.1 Well-formed trees

The signature ensures the *well-formedness* of the annotated abstract syntax trees. The functions defined in the signature impose constraints on the structure of the tree. Informally, the constraints can be summarized as follows:

- every node must have a corresponding function in the signature;
- the children of a node must have the type required by the domain of the function corresponding to the node.

Let us consider a node N in the tree. The first condition to have a correct tree is the existence of a function F in the signature, such as $func(N) = F$. The types of the children of N form a list of sorts $(S_1\ S_2\ \ldots\ S_m)$ and the domain of the function F is a list of sorts $(Q_1\ Q_2\ \ldots\ Q_n)$. The second condition given above can be expressed as follows:

$$(m = n) \wedge (\forall i, 1 < i < m : S_i = Q_i).$$

The possibility to have subsorts and sorts which are lists of other sorts makes a formal specification of the *well-formedness* more complex. We introduce first the operator "`<<`" which compares sorts and lists of sorts. For sorts, the operator defines the transitive closure (called *is_subsort_of**) of the subsort relation introduced in Section 2.

$$S << Q \ \equiv\ S\ is\_subsort\_of^*\ Q.$$

For list of sorts, if the right side does not contain sorts which are lists, then the operator checks a one-to-one correspondence between the two lists:

$$(S_1\ S_2\ \ldots\ S_m) << (Q_1\ Q_2\ \ldots\ Q_n) \equiv (m = n) \wedge (\forall i, 1 < i < m : S_i << Q_i).$$

For the case when the right side contains a `list` we introduce the following rule:

$$(S_1\ S_2\ \ldots\ S_n) << (list(S)) \equiv (\forall i, 1 < i < n : S_i << S).$$

The operator "<<" and two boolean functions are used to define the `well-formedness` of a tree with respect to a signature:

```
context-free syntax
  SORT "<<" SORT wrt SIGN                     -> BOOL
  SORT-LIST "<<" SORT-LIST wrt SIGN           -> BOOL
  well-formed  TREE  wrt SIGN                 -> BOOL
  well-formed  TREE-LIST  wrt SIGN            -> BOOL
```

For leaves the equation is:

```
equations
  [wf-01]  Tree = [ Annot ],
      func(Tree)= Id,
      func(Sign, Id)= Func
      ====================
      well-formed  Tree wrt Sign  = empty-SORT-LIST(domain(Func))
```

For internal nodes:

```
  [wf-02]  Tree = [ Annot TreeList ],
      func(Tree)= Id,
      func(Sign, Id)= Func
      ====================
      well-formed Tree wrt Sign  = well-formed TreeList wrt Sign  &
              sorts(children(Tree))  <<  domain(Func) wrt Sign
```

For list of trees:

```
  [wf-03] well-formed(Tree Trees) wrt Sign =
              well-formed Tree wrt Sign & well-formed (Trees) wrt Sign
  [wf-04] well-formed () wrt Sign = true
```

5.2 Example

We will now illustrate the use of annotated abstract syntax trees for a simple arithmetic expression language. The syntax is a modified form of the abstract syntax presented as example in Section 2. In this example the arithmetic operators are defined as functions on lists of expressions:

```
  sorts
      INT ID EXPR
  subsorts
      INT < EXPR, ID < EXPR
  functions
      plus: list(EXPR)        -> EXPR
      times: list(EXPR)       -> EXPR
      exp: EXPR # INT         -> EXPR
```

In order to create a graphical representation for the trees we define a function `print` which takes the tree as input and returns a sequence of graphical commands from which the image is created.

Figure 3 shows the abstract syntax tree for the expression: $x * 3 * 5 + 2 + y^1$. The leaf nodes are represented by squares and the internal nodes by circles. On the right side of every node are the annotations of that node. They are written in the format: `tag:value`. The tags of dynamic annotations start with "`a-`".

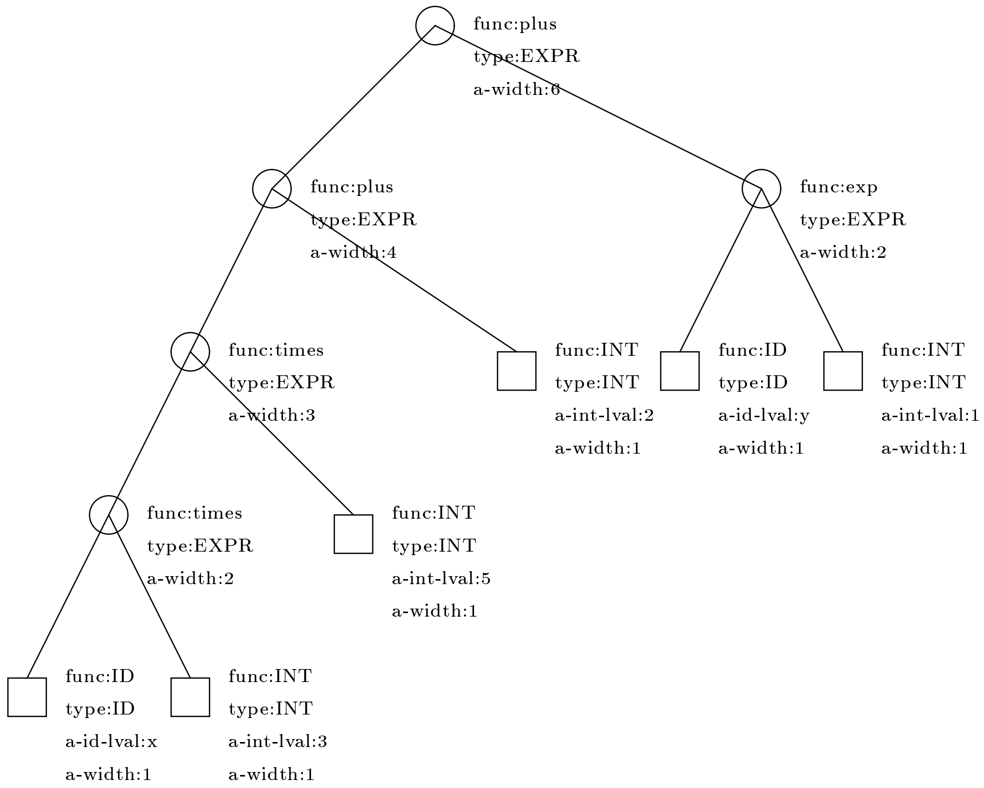

Figure 3: The abstract syntax tree for the expression $x * 3 * 5 + 2 + y^1$

.

The function `print` works in two phases. First, it computes the `width` of every node and stores the value as a dynamic annotation. This operations is done in a postfix traversal of the tree: the width of a node is computed based on the width of the children (see function `set-width-rec` in Section 4.3). Second, in a prefix traversal the function generates for every node a sequence of graphical commands to draw the image of the node. The arguments of these commands are based on the width information stored in nodes. The commands are used by a graphical interpreter to draw the image.

Two other functions are defined to illustrate the functionality of tree operations. One, called `flatten`, converts the abstract syntax tree with binary operators in a tree which contains nodes with variable number of children. This transformation uses the associativity information of the function which is stored in every node as a dynamic annotation. The tree shown in Figure 4 is the result of applying `flatten` to the abstract syntax tree for the expression $x * 3 * 5 + 2 + y^1$.

Another function, called `eval` does a kind of simplification: whenever it is possible to evaluate a part of the expression this is done. For example, operations between integers constants, multiplication by 0 or 1, etc. So, from the original expression we will obtain: $x * 15 + y + 2$. The result is shown in Figure 5.

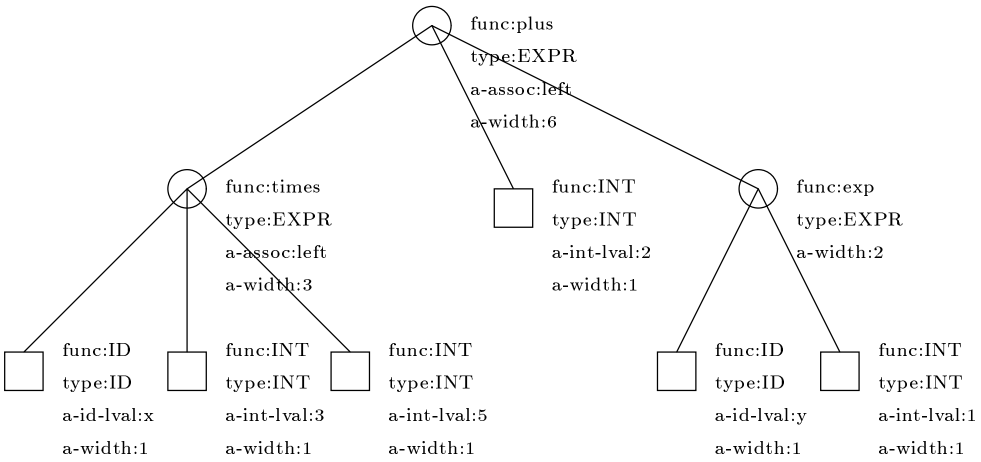

Figure 4: The abstract syntax tree for the expression $x * 3 * 5 + 2 + y^1$ after `flatten`

6 Related work

Other algebraic specifications for abstract syntax trees are, in general, given in papers which describe tools in an algebraic specification environment. Our work was mainly influenced by [Koo94]. It contains a specification of generic structured editors. For the abstract syntax we used a similar approach as in *Syntax Definition Formalism (SDF)* described in [HHKR89]. The "functional data model" is presented in [Gra85] in the context of database management systems.

The Virtual Tree Formalism (VTP) is a formalism used in the ASF+SDF Meta-environment for operation on abstract syntax trees. It offers interesting features like the possibility to add new data to a node and a form of pattern matching in trees using *metavariables*. A disadvantage of VTP is that it is implemented in LISP, and the main motivation for current study was to create a complete algebraic specification of a VTP like system. Ultimately, this can be used in a bootstrap of the ASF+SDF Meta-environment [Kli93].

A language for processing abstract syntax trees (LTA) is presented in [Hat89]. LTA uses the terms *phyla* and *operators* to define the abstract syntax of a language. These terms are equivalent to *sorts* and *functions* in our specification. In LTA it is possible to have subtrees of different languages, allows the access subtrees through pointers and has simple control structures.

7 Conclusions

When we started the specification of annotated abstract syntax trees we had two purposes:

- to develop an algebraic specification of abstract syntax trees. (This was important in a larger perspective of bootstrapping the ASF+SDF Meta-environment.)
- to create a module which is flexible and has a simple interface with other tools which are using trees.

Part of these goals were achieved in the current specification. The specification offers the basic functionality for operations on abstract syntax trees. Certainly this functionality can be extended.

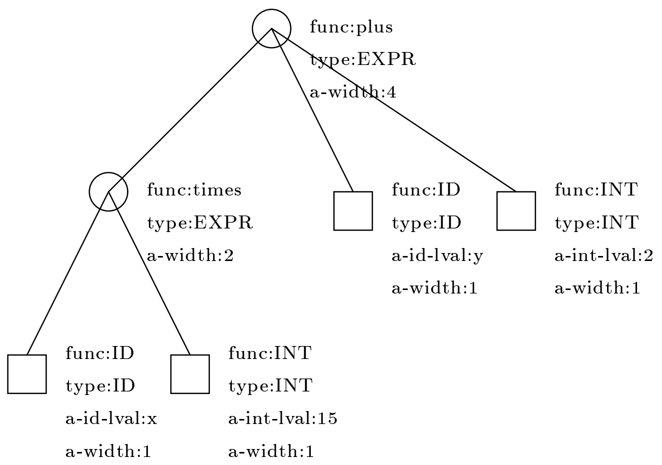

Figure 5: The abstract syntax tree for the expression $x * 15 + y + 2$. This is the result of application of `eval` to the expression $x * 3 * 5 + 2 + y^1$
.

What we have specified forms a minimum set of operations that can be used to create more complex functions.

We think that the use of a functional data model for the data stored in the nodes of a tree increase the flexibility. This is an important difference between our specification and other implementations of trees. In general, other specifications have a restricted set of data that can be stored in the nodes.

This flexibility is necessary in order to develop new tools which require new kinds of information to be stored in the nodes of a tree. A static structure for the data stored in the nodes is a serious obstacle to make a simple, flexible, specification of a tool which process the tree. The possibility to define dynamically the annotation of a node makes the implementation of various algorithms easier and does not require modification of other parts of the environment.

In general, the flexibility of a module means also a more complex interface. In the present specification the interface remains simple and easy to use. After the definition of a new decor, it can be used in the same way as the static annotations of the node.

The functional data model can be used in other cases when the structure of information must be dynamic. The solution is to extend the notion of annotation from nodes to other objects.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[Gra85] P.M.D. Gray. *Logic, algebra and databases.* Horwood, 1985.

[Hat89] D. Hattab. Lta: Un language de traitement d'arbres. Publication interne nr. 489, Institut de Rechereche en Informatique et Systèmes Aléatoires, Rennes, 1989.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Koo94] J.W.C. Koorn. *Generating uniform user-interfaces for interactive programming environments*. PhD thesis, University of Amsterdam, 1994.