Executable Specifications based on Dynamic Algebras

Angelica Maria Kappel

Universität Bonn, Institut für Informatik III
Römerstrasse 164, D-5300 Bonn 1

Abstract. In 1988, Y. Gurevich proposed an approach to operational semantics, which is based on finite, dynamic algebras. Dynamic algebras are comprehensible, precise and universally applicable. E. Börger recently presented a Dynamic Algebra Specification of full Prolog.
The main purpose of our work is a general concept for the implementation of dynamic algebras. We present a concrete language for Dynamic Algebra Specifications and the design of an abstract target machine specially tailored for Dynamic Algebra computations. Finally we explain some principles of code generation leading to highly efficient target programs.
Though the given concepts are independent from an implementation language, we show that they may very naturally be expressed within the framework of Logic Programming. A prototype of the compiler has been realized in Prolog. Starting with Börger's transition system for full Prolog we derived a Prolog interpreter in our Dynamic Algebra Specification Language.

1 Algebraic Operational Semantics

There are several approaches to operational semantics of programming languages. The essence is to explain the language by an abstract machine which takes terms of the language as input. Mathematical structures for the description of an abstract machine are traditionally static and infinite. All states of computation of the machine are given at once in only one infinite structure.

In 1988, Gurevich proposed an approach to operational semantics [Gur88], which is based on finite, dynamic algebras. What is new about the dynamic algebra is the way they are used: we are not interested in the set of all algebras, but we consider always only the actual algebra modelling the actual configuration at a given time. Hence, we describe a temporal logic. We represent the stepwise execution of a computation by an evolving structure, which develops with respect to a transition system. An algebra at a chosen point of computation is always finite. One of the main arguments to use dynamic algebras is their simplicity: they are comprehensible, precise and universally applicable.

Dynamic algebras have already been applied to give a semantics of several programming languages, especially Prolog, see [Bör90a, Bör90b, Bör90c, BD91].

A *Dynamic Algebra* is a tuple (Σ, I, F, T), consisting of a signature Σ, a finite, many-sorted, partial algebra I of signature Σ (initial state), a finite set

F of finite, many-sorted, partial algebras (final states) and a finite set T of transition rules.

We will use the abbreviation DA for dynamic algebras. I is called the *initial state* of the dynamic algebra. For every sort of Σ, there is a universe (or carrier set) in I. For every function symbol of Σ, I contains a partial function.

The initial state contains the universes and functions which are necessary to start a computation, e. g. a program and its input. Further, the initial state is equipped with the predefined universes BOOL, NUM and CHAR. F may be empty, meaning that a dynamic algebra may realize a perpetual computation. Starting from the initial state, the algebra changes with respect to the transition system.

The universes of the dynamic algebra evolve over time. The operations to change the algebra are called updates. Elements may be added to the universes or deleted. Deletion of an element may affect the functions, too. For every universe U of the dynamic algebra DA, equality $=_U$ is a function of DA. It is automatically updated with a universe contraction or extension, such that it always represents the identity relation.

The functions are evolving with the universes on which they are defined. They are given *in extensio* by enumeration. The contraction of a function comes implicitly with a universe contraction. Functional assignments are an explicit update of the dynamic algebra, see the description below.

There are three kinds of updates in a dynamic algebra.

1. **Function updates**
   Let $f(t_1, ..., t_n)$ and t be terms of the given signature, possibly containing variables introduced by a universe extension. The functional assignment
   $$f(t_1, ..., t_n) := t$$
   is called a function update. Its meaning is: First compute $t_1, ..., t_n$ and t in the actual state S. Let the elements $a_1, ..., a_n$ and a be the results. Then a will be the function value of f at argument position $(a_1, ..., a_n)$ in the next state. The values of f for all other argument tuples are left unchanged.
2. **Extension of universes**
   A universe extension is of the form
   $$\text{let } e = \text{ new}(U) \text{ in } F_1 ... F_n \text{ endlet}$$
   where the F_i are function updates containing e. Its meaning is: add a new element to the universe U and give this element temporarily the name e. Then perform the function updates $F_1 ... F_n$. The scope of e is delimited by the brackets `let` and `endlet`.
3. **Contraction of universes**
   Let t be a term over the given signature Σ.
   $$\text{dispose}(t)$$
   is a universe contraction with the following meaning: compute t in the actual state S and let a be the result. Delete a from the universe to which it belongs.

At the same time, the instances of functions having a in their domain or range become undefined.

A transition is an elementary step in the computation:

$$\text{if } C \text{ then } \mathrm{Up}_1, \ldots, \mathrm{Up}_n \text{ endif}$$

C is a closed boolean formula and $\mathrm{Up}_1, \ldots, \mathrm{Up}_n$ is a list of updates performed simultaneously. The meaning of a transition is: Evaluate the formula C in the actual state S. If C is true in S, the algebra S is changed into the algebra S' according to $\mathrm{Up}_1, \ldots, \mathrm{Up}_n$. If C is false in S, the transition has no effect.

The list of updates must be consistent. $\mathrm{Up}_1, \ldots, \mathrm{Up}_n$ is called *consistent* if the function updates in $\mathrm{Up}_1, \ldots, \mathrm{Up}_n$ are not identical on the left-hand sides. the updates $f(a) := b$ and $f(a) := c$ for example are not consistent[1]. A simultaneous execution of updates allows to perform a couple of related actions in one step, leading to a single algebra of interest. The context of term evaluation is always the actual state.

Motivated by the applications above, we investigate only deterministic *DA*s. A deterministic *DA*-computation is a finite or infinite sequence of states. For an arbitrary actual state S not in F, there is exactly one successor S'. For a given initial state I, there is at most one final state $E \in F$.

2 DASL: The Dynamic Algebra Specification Language

In the sequel, we define a concrete language for *DA*-specifications. The main elements of this language and their semantics are those of the previous section. In addition we define explicit constructs for aspects like input/output, initial and final states. Moreover, we want to facilitate programming with dynamic algebras by the use of syntactic abbreviations (macros and statics).

2.1 Declarations

Generally, a signature in DASL specifications may be divided into two parts: one part is marked as static and the other as dynamic. The *DA*-signature consists of the dynamic signature together with possibly static signatures and the predefined types `num, bool` and `string`. We write just signature instead of *DA*-signature and main signature for the dynamic part.

The main signature is defined by the declaration of sorts and functions in the following way:

```
sort Sortlist
func Funclist
```

[1] If $b \neq c$ in the actual state S, this definition agrees with the usual notion of consistency. In the case that the updates are logically equivalent in S (b and c evaluate to the same element in S), they are at the same time superfluous, so that we are motivated to forbid such constructions.

Sortlist and Funclist are sequences of type identifiers or function declarations.For each declared sort there is a carrier set with this name. Each declared function of the main signature will be represented by a function table and may be used dynamically. The standard functions of the predefined universe `num`, `bool` and `string` are predefined and therefore static. The declaration of a main signature, specially sorts, is optional.

In the known applications, *DA*s are used to specify one concrete process of special interest, less important aspects are thought to be given. Sometimes it is too troublesome to specify everything by dynamic algebras, e. g. lists. So the user needs a possibility to specify details of the description only needed for executability in a shorter, though less intuitive way. For this purpose, a DASL specification is divided into two parts. The evolving, finite part are the universes and functions belonging to the main signature. The second part consists of predefined and static types and leads to an infinite contribution to the algebra of discourse.

The static part is an equational specification (Σ, E) consisting of a signature and a set of equations. Its semantics is the initial algebra in the model class Mod(E) of E wrt. Σ. This initial model is assumed to be part of the *DA*. The syntactic definition of the static part is:

> `staticsort` Sortlist
> `con` Conslist
> `staticfunc` Funclist
> `eqs` Equations

Sortlist is a sequence of possibly parameterized polymorphic sorts and Conslist is a sequence of constructors. Funclist is a sequence of function declarations. Argument and result types must be predefined or static and must have been declared before.

A sequence of conditional or unconditional equations are defining the declared functions. Always the first matching equation will be applied for reduction of a term during the computation. In this way, the equational specification constitutes a confluent rewrite system. For details of the static part see [Kap90]

2.2 Operations

$$\text{Var} <- \texttt{new}(\text{S})$$

introduces a new element in the universe of the sort S, which must be a sort of the main signature. No predefined or static type is allowed here.

$$\texttt{dispose}(\text{t})$$

is the deletion of an element. t must be a term of the main signature that has no predefined or static type as result type. The universe that is contracted by this operation is determined by inferring the type of t. All function tables containing the deleted element at some argument or result position are contracted by deleting that entry.

$$f(t_1, \ldots, t_n) := t$$

is a function update. f is a function symbol of the main signature, $t_1, \ldots, t_n$ and t are arbitrary terms of the signature or identifiers declared by a preceding new-operation. The function table of f will be changed by this operation in the way that a possibly existing entry at position $t_1, \ldots, t_n$ will be deleted. Afterwards a new entry will be added at this position with the result of t as function value.

2.3 Transitions

`transition` Name `if` Condition `then` Updates

Name is the identifier of the transition and must be unique in the specification file. It will be used for compile time and run time information, e.g. tracing the execution. A Condition is a conjunction of equations $t_1 = t_2$ or disequations $t_1 <> t_2$. & is used as conjunction symbol.

Updates are a list of a *DA*-operations, macros or output operations separated by commas. Updates may occur in arbitrary order, as long as new elements are declared before they are used. No order is imposed on the execution of the updates; they are supposed to be performed simultaneously.

2.4 Initial and Final States

An important issue of the implementation of dynamic algebras is the specification of the initial state. In DASL, the starting (initial) state contains at least the initial model of predefined and static types. Concerning the main signature, we add an empty universe for each sort and an empty function table for each function. All universes and functions that are supposed to be nonempty in the initial state must be updated initially before starting a computation. We allow this by the use of an explicit start rule:

`start` Updates

Updates are a sequence of universe extensions or function updates separated by commas.

Final states are given implicitly by the transitions together with an initial state, because they are those states in which no transition is applicable. However, in DASL we want to distinguish between error situations and regular termination. So, the set of regular final states must be known. In this way, we are able to define error-free transition systems.

DASL final states are specified in the following way:

`result` Name `if` Conditions `then` Finalactions

Name is a unique identifier, as above. Conditions are the same as in transitions. These conditions are essentially the characterization of the final state.

In the final state, no further alterations of the algebra are allowed. Therefore, *Finalactions* are only operations for user information, i.e. output operations.

2.5 Other Language Constructs

Macros are available in many programming systems. It is a well-known mechanism for textual replacements that allows abbreviations for repeated statements. In DASL they may be used either in conditions or in updates of rules. The definition of a macro has the form:

`macro` Name => Replacement.
or `macro` Name (Parameters) $\rightarrow$ Replacement.

A *DA*-computation starts in the initial state and stops in a final state. At an abstract level, the input and output of a computation is the entire state. But in practice, the user is only interested in a part of these states e.g. the dynamic functions representing explicitely the input and output values. For this purpose, DASL has an output operation.

output(Function)

prints the function table for the denoted function. *Function* must be a function of the main signature. The output operation may occur in the update part of any transition rule, but not in the start rule.

2.6 Example

The use of static types is shown in the following little program, which generates a list of decreasing numbers. It starts with the empty list.

```
staticsort list(S).

con  nil: list(S),
     infix ::(S,list(S)): list(S).

staticfunc head(list(S)): S,
           tail(list(S)): list(S),
           app(list(S),list(S)): list(S).

eqs  head(A::B) = A,
     tail(A::B) = B,
     app(nil,L) = L,
     app(X::R,L) = (X::app(R,L)).

func actlist:list(num),
     endelem:num.

start  actlist := nil,
       endelem := 42.

result testresult
if head(actlist) = endelem
```

```
then output(actlist).

transition test_nil
if actlist = nil
then actlist := 1::actlist.

transition test_cons
if head(actlist) <> endelem
then actlist := (head(actlist)+1)::actlist.
```

3 ALMA: An Algebraic Target Machine

In this section we introduce a small and very simple abstract machine working on algebras. ALMA is single-sorted. The storage consists of a dynamic set of relations and a static set of Horn clause specifications. ALMA provides three kinds of control statements: A simple statement and two conditional statements: an *if*- and a *case*-statement. A simple statement is a sequence of basic operations, which are defined analogously to the DASL operations. An ALMA program is given by a decision tree. The decision tree contains *if*- and *case*-nodes. The leaves are either *update*- or *error*-nodes. A path in this tree represents one step in a *DA*-computation. An ALMA-computation is the execution of a simple statement (initialization) followed by a recursive walk through the decision tree.

From the implementational point of view, ALMA is a Prolog program. The basic operations are implemented by Prolog predicates with the same name. A simple statement corresponds to a Prolog literal containing calls of basic predicates. The *if*-statement is simply implemented by the use of Prolog's if-then-else-construct. Each *case*-statement is implemented by a different predicate with individual identifier. The decision tree is a compound control statement of nested if-then-elses and calls of case-predicates. In our implementation, only a few components of ALMA are fixed. Most work is done by a compiler from DASL to Prolog.

The following example shows a DASL program, that switches a traffic light from green over yellow to red with a delay between the changes.

```
staticsort lights.
con red, green, yellow: lights.

sort counter=num.
func a: counter,
     b: lights.

start  b := green,
       a := 10.

result switch_off
if  a = 0 &
    b = red
```

```
then output(b).

transition count_down
if   a <> 0
then a := a - 1.

transition switch_yellow
if   a = 0 &
     b = green
then b := yellow,
     a := 10.

transition switch_red
if   a = 0 &
     b = yellow
then b := red,
     a := 10.
```

The transition system may be transformed into a decision tree with *if*- and *case*-nodes. Following, you find the target code generated by our compiler. **`start`** represents the simple statement. It serves to initialize the dynamic part of the storage. **`step`** is the root of the decision tree. An ALMA program is called by

```
                run :- start, step.
```

```
start update b(green),
      update a(10).

step :- a(X) ->
        (X = 0 -> step0 ; step1(X))
        ;
        step2.

step0 :- b(Y) ->
         case0(Y)
         ;
         step5.

case0(green) :- update b(yellow),
                update a(10),
                step.

case0(yellow) :- update b(red),
                 update a(10),
                 step.

case0(red) :- show(b,1).

step1(X) :- Y is X - 1,
            update a(Y),
            step.

step2 :- write('error situation:'),
         write(a = 0),
         write(' undefined.').

step5 :- write('error situation:'),
         write(b),
         write(' undefined.').
```

Note how Prolog's first argument indexing makes the *case*-statement highly efficient.

4 Compiling Techniques

In order to compile a DASL program into an ALMA program, a many-sorted DASL signature is transformed into a one-sorted ALMA signature. Different elements of the source program are represented by different elements in the target code. In particular DASL sorts are interpreted by disjoint subsets of the unique carrier set of ALMA. Dynamic DASL functions are mapped to ALMA predicates. Functional expressions of the DASL source program are compiled to a conjunction of atoms as target code. The technique that we use for the transformation of functions to relations is flattening.

Moreover, we use a memo-table to guarantee that every compilation of a term results in the same compiled term. This memorizing serves to augment efficiency. Without the use of this technique, we obtain n different variables and n flat formulas only differing in their variables if we compile the same expression n times. At runtime, these different variables would be bound to the same value by evaluating the flat formulas in the current interpretation.

An equational specification of a DASL program is translated into the static part of an ALMA database given by a Horn Clause specification.

The type information present in DASL is used for efficient compilation of universe contractions in the following way: Let Σ = (S,F) be a DASL main signature, i.e. F a set of function symbols for dynamic functions. Below, δ denotes the flattening operation. Derive a family of sets $(\mathrm{C}_s)_{s \in \mathrm{S}}$ of pairs of predicates and natural numbers,

$$\mathrm{C}_s = \{(\delta(\mathrm{f}^{s_1 \ldots s_n, s_{n+1}}), i) | s = s_i \text{ or } s \text{ is a type parameter in } s_i, 1 \leq i \leq n+1\}$$

The natural number i in a pair(f, i) denotes the position of sort s in $\delta(f)$. Then extend ALMA's dispose operation *dispose(E)* by adding a second argument to *dispose(E,C_s)*. The meaning of this new *dispose*-operation is to delete the components of predicates in C_s that have element E at the argument position indicated by the number. This technique also increases the run time efficiency, because elements to be deleted are only searched at reasonable positions in the storage.

DASL updates are defined to be performed simultaneously, whereas the basic operations of a simple ALMA statement are executed sequentially. But as the basic ALMA operations (apart from the new-operation, which corresponds to the instantiation of a variable) have no free variables when they are executed, the sequential execution is equivalent to a simultaneous execution. Syntactical restriction of the source language guarantee, that new elements are introduced before they are used, so this is also done in the target code. All terms in a list of updates must be evaluated in the same state. This in guaranteed, as the compiled terms are instantiated before basic operations are executed.

Transforming a Transition System into a Decision Tree

We compile a given transition system into a decision tree consisting of *if*- and *case*-nodes and two kinds of leaves, called *update*- and *error*-nodes. The root of a decision tree is either an *if*- or a *case*-node. Every path in this tree from the root to an *update*-node corresponds to a transition or result rule.

As the conjunction in the conditional part of the rules is symmetric, there are many decision trees for an ALMA program. For every finite set of rules exists a finite set of decision trees. These trees differ in depth and width. For the sake of efficiency, we are only interested in a minimal decision tree. A *minimal* tree in a set of decision trees is a tree with the least depth. In case there are several trees with the same minimal depth, minimal trees are further restricted to be those with the least number of nodes.

A rule must sometimes be extended during the transformation process, what means that a condition, which originally did not belong to its conditional part will be added. In this case, we obtain two new rules, one with a positive occurrence of the added condition and one where the new condition occurs negatively, see [Kap90] for details.

We are able to detect determinism or possible nondeterminism, or to develop integrity constraints for determinism during the transformation of DASL rules into a decision tree. The transformation algorithm selects recursively an equation out of all conditions of the given set of transition rules until all rules are transformed. A transition system may have several corresponding decision trees, depending on the choice of the equation in the transformation process. We are mainly interested in the minimal decision tree. For this purpose, the following ideas are essential:

1. Obviously, it is convenient to get to the update part of a rule as soon as possible. An *update*-node will be built then, and for reason of determinism, all other rules still remaining in the actual set R can be omitted. This leads to a tree with less depth and less nodes. So we look for the equation to be selected in the C_i with the least number of equations.
2. We want to keep the extension process a s small as possible. There are two reasons for it: the extension procedure takes less time and, what is more important, we have fewer rules in R and consequently fewer nodes in the tree. Every rule not containing the selected equation in the conditional part will

appear in all subtrees of an *if*-node. So it will be considered several times in the resulting decision tree. In the worst case a tree can have identical subtrees. We try to avoid this by choosing an equation that has the maximal number of occurrences in the C_i of R.

3. Finally, it is better to choose an equation that appears only positively or only negated. One subtree of the node will be either true then or contains only rules that have been extended with eq. This leads also to a tree with less nodes.

With theses heuristics, our transformation procedure leads to very efficient decisions trees. It always results in a minimal tree as long as the rules must not necessarily be extended, see [Kap90] for details.

Example We now consider rules as logical formulas

$$r_i \equiv C_i \Rightarrow U_i, \quad i = 1 \ldots n$$

C_i is a conjunction of propositional constants denoting equations and U_i is a variable. Let R be the following rules:

$$\begin{aligned} &(a \wedge b) \Rightarrow U_1 && (1)\\ \wedge\ &(a \wedge \neg b \wedge c) \Rightarrow U_2 && (2)\\ \wedge\ &(a \wedge \neg b \wedge \neg c) \Rightarrow U_3 && (3) \end{aligned}$$

These rules are trivially deterministic. We show now three possible decision trees for R, differing in the choice of the root.

$$\begin{aligned} &(c \Rightarrow ((a \Rightarrow ((b \Rightarrow U_1) \wedge (\neg b \Rightarrow U_2))) \wedge (\neg a \Rightarrow \text{true})))\\ \wedge\ &(\neg c \Rightarrow ((\neg a \Rightarrow \text{true}) \wedge (a \Rightarrow ((b \Rightarrow U_1) \wedge (\neg b \Rightarrow U_3))))) \end{aligned}$$

The rule (1) is represented twice in this tree, because it has been extend wrt. c and occurs afterwards in both subtrees. This can be avoided by the choice of a or b as root.

$$\begin{aligned} &(b \Rightarrow ((a \Rightarrow U_1) \wedge (\neg a \Rightarrow \text{true})))\\ \wedge\ &(\neg b \Rightarrow ((a \Rightarrow ((c \Rightarrow U_2) \wedge (\neg c \Rightarrow U_3)) \wedge (\neg a \Rightarrow \text{true}))) \end{aligned}$$

Obviously, this tree has less nodes than the first one. Each rule is only represented once in the tree, because there has been no extension of R. But there is still an equation that occurs twice in the tree. This can also be avoided.

$$\begin{aligned} &(a \Rightarrow ((b \Rightarrow U_1) \wedge (\neg b \Rightarrow ((c \Rightarrow U_2) \wedge (\neg c \Rightarrow U_3)))))\\ \wedge\ &(\neg a \Rightarrow \text{true}) \end{aligned}$$

This tree is a minimal tree and is produced by our algorithm.

5 Conclusions

Dynamic algebras are a valuable tool of specification. Dynamic algebras have been used to define operational semantics of several programming languages, especially Prolog, see [Bör90a, Bör90b, Bör90c, BD91]. The cited publications have already been used by the ISO commitee for the Prolog standardization process.

In order to make dynamic algebra specifications executable, we have defined and implemented the language DASL. DASL is an extension of the dynamic algebras formalism. It includes polymorphic types and equational specifications. Equational specifications are used with advantage for those parts of a specification which would become too big and complicated using dynamic algebras.

As an application, we have used DASL to implement Börger's Prolog specification. Prolog's terms and unification are specified by equational specifications. The control part of the Prolog engine as well as the dynamic database, however, are described most elegantly in the dynamic algebra style.

DASL is both implemented in and compiled to Prolog. Using decision trees, the target code produced by our compiler is highly efficient. Prolog's own compilation and optimization techniques contribute to a reasonable execution speed.

References

[BD91] E. Börger and B. Demoen. A Framework to Specify Database Update Views for Prolog. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP)*, number 528, 1991.

[BKBR89] E. Börger, H. Kleine Büning, and M. Richter, editors. *3rd Workshop on Computer Science Logic*, 1989.

[Bör90a] Egon Börger. A Logical Operational Semantics of Full Prolog, part I: Selection Core and Control. In Börger et al. [BKBR89].

[Bör90b] Egon Börger. A Logical Operational Semantics of Full Prolog, part II: Built-in Predicates for Database Manipulations. Technical Report IWBS 115, IBM Deutschland GmbH, 1990.

[Bör90c] Egon Börger. A Logical Operational Semantics of Full Prolog, part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output. In Y Moschovakis, editor, *Workshop on Logic from Computer Science*. Springer Verlag, 1990.

[Gur88] Yuri Gurevich. Logic and the Challenge of Computer Science. In Egon Börger, editor, *Current Trends in Theoretical Computer Science*. Computer Science Press, 1988.

[Kap90] A. M. Kappel. Implementation of Dynamic Algebras with an Application to Prolog. Diploma thesis, Universität Dortmund, Germany, 1990.

This article was processed using the LaTeX macro package with LLNCS style