leanEA: A Lean Evolving Algebra Compiler*

Bernhard Beckert Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, 76128 Karsruhe, Germany
Email: {beckert,posegga}@ira.uka.de
WWW: http://i12www.ira.uka.de

Abstract. The Prolog program

```
"term_expansion((define C as A with B), (C=>A:-B,!)).
 term_expansion((transition E if C then D),
                ((transition E):-C,!,B,A,(transition _))) :-
         serialize(D,B,A).
 serialize((E,F),(C,D),(A,B)) :- serialize(E,C,B), serialize(F,D,A).
 serialize(F:=G, ([G]=>*[E],F=..[C|D],D=>*B,A=..[C|B]), asserta(A=>E)).
 [G|H]=>*[E|F] :- (G=\E; G=..[C|D],D=>*B,A=..[C|B],A=>E), !,H=>*F.
 []=>*[].
 A=?B :- [A,B]=>*[D,C], D==C."
```

implements a virtual machine for evolving algebras. It offers an efficient and very flexible framework for their simulation.

1 Introduction

Evolving algebras (EAs) (Gurevich, 1991; Gurevich, 1994) are abstract machines used mainly for formal specification of algorithms. The main advantage of EAs over classical formalisms for specifying operational semantics, like Turing machines for instance, is that they have been designed to be usable by human beings: whilst the concrete appearance of a Turing machine has a solely mathematical motivation, EAs try to provide a user friendly and natural—though rigorous—specification tool. The number of specifications using EAs is rapidly growing;[1] examples are specifications of the languages ANSI C (Gurevich & Huggins, 1993) and ISO Prolog (Börger & Rosenzweig, 1994), and of the virtual architecture APE (Börger *et al.*, 1994b). EA specifications have also been used to validate language implementations (e.g., Occam (Börger *et al.*, 1994a)) and distributed protocols (Gurevich & Mani, 1994).

There is little sense in implementing a Turing machine (besides for pedagogical reasons); however, an implementation of a machine for executing EAs can help a person

*A long version of this paper is available from the authors, that includes a formal treatment of the semantics of leanEA programs and describes possible extensions.

[1]There is a collection of papers on EAs and their application on the *World Wide Web* at http://www.engin.umich.edu/~huggins/EA.

working with this formalism a lot, as the level of abstraction in an EA specifications is problem-oriented.

This observation is of course not new and implementations of abstract machines for EAs already exist: Angelica Kappel describes a Prolog-based implementation in (Kappel, 1993), and Jim Huggins reports an implementation in C. Both implementations are quite sophisticated and offer a convenient language for specifying EAs.

In this paper, we describe an approach to implementing an abstract machine for EAs which is different, in that it emphasizes on simplicity and elegance of the implementation, rather than on sophistication. We present a simple, Prolog-based approach for executing EAs. The underlying idea is to map EA specifications into Prolog programs. Rather than programming a machine explicitly, we turn the Prolog system itself into a virtual machine for EA specifications: this is achieved by changing the Prolog reader, such that the transformation of EAs into Prolog code takes place whenever the Prolog system reads input. As a result, evolving algebra specifications can be treated like ordinary Prolog programs.

The main advantage of our approach, which we call leanEA, is that it is very flexible: the Prolog program we discuss in the sequel can easily be understood and extended to the needs of concrete specification tasks (non-determinism, special handling of undefined functions, etc.). Furthermore, its flexibility allows to easily embed it into, or interface it with other systems.

The paper is organized as follows: in Section 2, we start with explaining how a deterministic, untyped EA can be programmed in leanEA; this section is written pragmatically, in the sense that we do not present a mathematical treatment, but explain what a user has to do in order to use EAs with leanEA. The implementation of leanEA is explained in parallel. An extended example of using leanEA is given in Section 3. In Section 4 we make some remarks regardings semantics of leanEA specifications. We draw conclusions from our research in Section 5.

2 Programming EAs in leanEA

2.1 The Basics of leanEA

An algebra can be understood as a formalism for describing static relations between things: there is a universe consisting of the objects we are talking about, and a set of functions mapping members of the universe to other members. *Evolving* algebras offer a formalism for describing changes as well: an algebra can be "moved" from one state to another, in that the functions can be changed.

leanEA is a programming language that allows to program this behaviour. From a declarative point of view, a leanEA program is a specification of an EA. Here, however, we will not argue declaratively, but operationally by describing how statements of leanEA set up an EA and move it from one state to another.

leanEA is an extension of standard Prolog[2], thus a leanEA program can be treated like any other Prolog program, i.e., it can simply be loaded (or compiled) into the underlying Prolog system (provided leanEA itself has been loaded before).

[2] We assume the reader to be familiar with Prolog. An introduction can be found in (O'Keefe, 1990).

leanEA has two syntactical constructs for programming an EA: the first are *function definitions* of the form

`define` *Location* `as` *Value* `with` *Goal*.

which specify the initial state of an EA.

The second construct are *transition definitions* which define the EA's evolving, i.e., the mapping from one state to the next:

`transition` *Name* `if` *Condition* `then` *Updates*.

The signature of EAs is in our approach the set of all ground Prolog terms. The (single) universe, that is not sorted, consists of ground Prolog terms, too; it is not specified explicitly.

Also, the final state(s) of the EA are not given explicitly in leanEA. Instead, a state S is defined to be final if no transition is applicable in S or if a transition fires that uses undefined functions in its updates.

The computation of the specified evolving algebra is started by calling the Prolog goal "`transition _`".

2.2 Representation of States in leanEA

Before explaining how function definitions set up the initial state of an EA, we take a look at the leanEA internals for representing states: A state is given by the mapping of locations to their values, i.e., elements of the universe. A location $f(u_1, \ldots, u_n)$, $n \geq 0$, consists of a functor f and arguments $u_1, \ldots, u_n$ that are members of the universe.

Example 1 Assume, for instance, that there is a partial function denoted by `f` that maps a pair of members of the universe to a single element, and that `2` and `3` are members of the universe. The application of `f` to `2` and `3` is denoted by the Prolog term `f(2,3)`. This location can either have a value in the current state, or it can be undefined.

A state in leanEA is represented by the values of all defined locations. Technically, this is achieved by defining a Prolog predicate `=>/2`,[3] that behaves as follows: The goal "`Loc => Val`" succeeds if `Loc` is bound to a ground Prolog term that is a location in the algebra, and if a value is defined for this location; then `Val` is bound to that value. The goal fails if no value is defined for `Loc` in the current state of the algebra.

To evaluate a function call like `f(f(2,3),3)`, leanEA uses the evaluation predicate `=>*/2`: the relation t `=>*` v holds for ground Prolog terms t and v if the value of t—where t is interpreted as a function call—is v (in the current state of the algebra).

In general, the arguments of a function call are not elements of the universe (contrary to the arguments of a location). They are recursively evaluated. To make it possible to use members of the universe in function calls explicitly, they can be denoted by preceding them with a backslash "`\`": this disables the evaluation of whatever Prolog term comes after the backslash. We will refer to this as *quoting* in the sequel.

[3] Note, that `=>/2` is defined to be dynamic such that it can be changed by transitions (Fig. 1, Line 8).

```
:- op(1199,fy,(transition)),  op(1180,xfx,(if)),
   op(1192,fy,(define)),      op(1185,xfy,(with)),
   op(1190,xfy,(as)),         op(1170,xfx,(then)),
   op(900,xfx,(=>)),          op(900,xfx,(=>*)),
   op(900,xfx,(:=)),          op(900,xfx,(=?)),
   op(100,fx,(\)).

:- dynamic (=>)/2.

term_expansion((define Term as Res with Code),
               ((Term => Res) :- Code,!)).

term_expansion((transition Name if Cond then Update),
               (transition(Name) :-
                  (Cond,!,FrontCode,BackCode,transition(_)))) :-
        serialize(Update,FrontCode,BackCode).

serialize((A,B),(FrontA,FrontB),(BackB,BackA)) :-
        serialize(A,FrontA,BackA),
        serialize(B,FrontB,BackB).

serialize((LocTerm := Expr),
          ([Expr] =>* [Val], LocTerm =.. [Func|Args],
           Args =>* ArgVals, Loc =..[Func|ArgVals]),
          asserta(Loc => Val)).

([H|T] =>* [HVal|TVal]) :-
       (   H = \HVal
       ;   H  =.. [Func|Args], Args =>* ArgVals,
           H1 =.. [Func|ArgVals], H1 => HVal
       ),!,
       T =>* TVal.

[] =>* [].

(A =? B) :- ([A,B] =>* [Val1,Val2]), Val1 == Val2.
```

Figure 1: lean*EA*: the program

For economical reasons, the predicate `=>*/2` actually maps a list of function calls to a list of values. Figure 1, Lines 27–34, shows the Prolog code for `=>*`, which is more or less straightforward: if the term to be evaluated is preceded with a backslash, the term itself is the result of the evaluation; otherwise, all arguments are recursively evaluated and the value of the term is looked up with the predicate `=>/2`. Easing the evaluation of the arguments of terms is the reason for implementing `=>*` over lists. The base step of the recursion is the identity of the empty list (Line 34). `=>*` fails, if the value of the function call is undefined in the current state.

Example 2 As an example, consider again the binary function `f`, and assume it behaves like addition in the current state of the algebra. Then both the goals "`[f(\1,\2)] =>* [X]`" and "`[f(f(\0,\1),\2)] =>* [X]`" succeed with binding `X` to `3`. The goal "`[f(\f(0,1)},\2)] =>* [X]`", however, will fail since addition is undefined on the term `f(0,1)`, which is not an integer.

After exploring the lean*EA* internals for evaluating expressions, we come back to programming in lean*EA*. The rest of this section will explain the purpose of function and transition definitions, and how they affect the internal predicates just explained.

2.3 Function Definitions

The initial state of an EA is specified by a sequence of function definitions. They define the inital values of locations by giving Prolog code to compute these values. A construct of the form

`define` *Location* `as` *Value* `with` *Goal*.

gives a procedure for computing the value of a location that matches the Prolog term *Location*: if *Goal* succeeds, then *Value* is taken as the value of this location. Function definitions set up the predicate `=>` (and thus `=>*`) in the initial state. One function definition can specify values for more than one functor of the algebra. It is possible in principle, although quite inconvenient, to define all functors within a single function definition. The value computed for a location may depend on the additional Prolog code in a lean*EA*-program (code besides function and transition definitions), since *Goal* may call predicates from the additional code. If several function definitions define values for a single location, the (textually) first is chosen.

A function definition is translated into the Prolog clause

(*Location* `=>` *Value*) `:-` *Goal*`,!.`

Since each definition is mapped into one such clause, *Goal* must not contain a cut "`!`"; otherwise, the cut might prevent Prolog from considering subsequent `=>` clauses that match a certain location.

The translation of a define statement to a `=>` clause is implemented by modifying the Prolog reader as shown in Figure 1, Lines 10–11.[4]

Example 3 Examples for function definitions are:

[4] In most Prolog dialects (e.g., SICStus Prolog and Quintus Prolog) this is done by adding clauses for the `term_expansion/2` predicate. If a term t is read, and `term_expansion(`t`,S)` succeeds and binds the variable `S` to a term s, then the Prolog reader replaces t by s.

`define register1 as 1 with true.` Assigns the value `1` to the constant (0-ary location) `register1`.

`define X as X with (X=[]; X=[H|T]).` This defines that all lists evaluate to themselves; thus, it is not necessary to quote lists in function calls with a backslash. Similarly,

`define X as X with integer(X).` defines that Prolog integers are in the universe and evaluate to themselves.

`define X+Y as Z with Z is X+Y.` This definition shows how Prolog predicates can be used for calculating the value of (external) functions within an EA.

The user is responsible that the Prolog goals for calculation values meet certain conditions: the computed values have to be *ground* Prolog terms, and the goals must either fail or succeed (i.e., terminate) for all possible instantiations that might appear.[5] In addition, the goals must not change the Prolog data base or have any other side effects;[6] and they must not call the `lean`*EA* internal predicates `transition/1`, `=>*/2`, and `=>/2`.

2.4 Transition Definitions

A transition, if applicable, maps one state of an EA to a new state by changing the value of certain locations. Transitions have the following syntax:

`transition` *Name* `if` *Condition* `then` *Updates*.

where

Name is an arbitrary Prolog term (usually an atom).

Condition is a Prolog goal containing calls to the predicate `=?/2` (see below), or combinations thereof that are built using the logical Prolog operators "`,`" (conjunction), "`;`" (disjunction), "`->`" (implication), and "`\+`" (negation).

Updates is a comma-separated sequence of updates of the form

$$\begin{aligned} f_1(r_{11},\ldots,r_{1n_1}) &:= v_1\,, \\ &\vdots \\ f_k(r_{k1},\ldots,r_{kn_k}) &:= v_k \end{aligned}$$

An update $f_i(r_{i1},\ldots,r_{in_i})$ `:=` v_i $(1 \le i \le k)$ changes the value of the location that consists of (a) the functor f_i and (b) the elements of the universe that are the values of the function calls $r_{i1},\ldots,r_{in_i}$; the new value of this location is determined by evaluating the function call v_i. All function calls in the updates are evaluated

[5] Prolog exceptions that terminate execution have to be avoided as well. Thus, it is advisable to formulate the definition of `+` as:
`define X+Y as Z with integer(X),integer(Y),Z is X+Y.`

[6] Side effects that do not influence other computations are harmless and often useful; an example are the definitions for input and output of the EA in Section 3.

simultaneously (i.e., in the old state). If one of the function calls is undefined, the assignment fails.

If the left-hand side of an update is quoted by a preceding backslash, the update will have no effect besides that the right-hand side is evaluated; the meaning of the backslash cannot be changed.

A transition is applicable (fires) in a state, if *Condition* succeeds. For calculating the successor state, the (textually) first applicable transition is selected. Then the *Updates* of selected transition are executed. If no transition fires or if one of the updates of the first firing transition fails, the the new state cannot be computed. In that case, the evolving algebra terminates, i.e., the current state is final. Else the computation continues iteratively with calculating further states of the algebra.

A transition is transformed into the clause

```
transition(Name) :-
        Condition, !,
        UpdateCode,
        transition(_).
```

This is done by modifying the Prolog reader as shown in Figure 1, Lines 13–16. Since updates must be executed simultaneously, all function calls are evaluated before the first assignment takes place. The auxiliary predicate `serialize/3` (Lines 18–25) serves this purpose: it splits all updates into evaluation code, that uses the predicate `=>*/2`, and into code for storing the new values by asserting an appropriate `=>/2` clause.

Besides logical operators, lean*EA* allows in the condition of transitions the predefined predicate `=?/2` (Fig. 1, Line 36) implementing the equality relation: the goal "s `=?` t" succeeds if the function calls s and t evaluate (in the current state) to the same element of the universe. It fails, if one of the calls is undefined or if they evaluate to different elements.

It is possible to implement similar relations using the lean*EA* internal predicate `=>*` to evaluate the arguments of the relation: A predicate $p(t_1, \ldots, t_n)$ $(n \geq 0)$ is implemented by adding the code

```
p(t_1,...,t_n) :-
        [t_1,...,t_n] =>* [x_1,...,x_n],
        Code.
```

to lean*EA*.[7] Then the goal "$p(t_1, \ldots, t_n)$" can be used in conditions of transitions instead of "$p'(t_1, \ldots, t_n)$ `=? true`", where p' is defined by the function definition

```
define p'(x_1,...,x_n) as true with Code.
```

(which is the standard way of implementing relations using function definitions). Note, that p fails, if one of the function calls $t_1, \ldots, t_n$ is undefined in the current state.

Example 4 The is-not-equal relation is implemented by adding the clause

```
(A <> B) :- ([A,B] =>* [Val1,Val2], Val1 \== Val2).
```

for the predicate `<>/2` to lean*EA*.

[7] $x_1, \ldots, x_n$ must be n distinct Prolog variables and must not be instantiated when `=>*` is called. Thus, "`(A =? B) :- ([A,B] =>* [V,V]).`" must not be used to implement `=?`, but "`(A =? B) :- ([A,B] =>* [V1,V2]), V1 == V2.`".

2.5 leanEA's Operators

To make it possible to use the syntax for function and transition definitions as described in the previous sections, a couple of Prolog operators have to be defined with appropriate preferences; they are shown in Figure 1, Lines 1–6.

Note, that the preferences of operators (those pre-defined by leanEA as well as others used in a leanEA program) can influence the semantics of Prolog terms and thus of function calls.

3 An Example Algebra

The following program specifies an EA for computing $n!$:

```
define state as initial with true.
define readint as X with read(X), integer(X).
define write(X) as X with write(X).
define X as X with integer(X).
define X-Y as R with integer(X),integer(Y),R is X-Y.
define X*Y as R with integer(X),integer(Y),R is X*Y.

transition step
     if   state =? \running, \+(reg1 =? 1)
     then reg1 := reg1-1,
          reg2 := (reg2*reg1).

transition start
     if   state =? \initial
     then reg1  := readint,
          reg2  := 1,
          state := \running.

transition result
     if   state =? \running, reg1 =? 1
     then reg2  := write(reg2),
          state := \final.
```

The constant `state` is used for controlling the firing of transitions: in the initial state, only the transition `start` fires and reads an integer; it assigns the input value to `reg1`. The transition `step` iteratively computes the faculty of `reg1`'s value by decrementing `reg1` and storing the intermediate results in `reg2`. If the value of `reg1` is `0`, the computation is complete, and the only applicable transition `result` prints `reg2`. After this, the algebra halts since no further transition fires and a final state is reached.

4 Some Remarks Regarding Semantics

Relations There are no special pre-defined elements denoting true and false in the universe. The value of the relation `=?` (and similar pre-defined relations) is represented by succeeding (resp. failing) of the corresponding predicate.

Undefined Functions Calls Similarly, there is no pre-defined element `undef` in the universe, but evaluation fails if no value is defined. This, however, can be changed by adding

```
define _ as undef with true.
```

as the last function definition.

Internal and External Functions In leanEA there is no formal distinction between internal and external functions. Function definitions can be seen as giving default values to functions; if the default values of a function remain unchanged, then it can be regarded external (pre-defined). If no default value is defined for a certain function, it is classically internal. If the default value of a location is changed, this is what is called an external location in (Gurevich, 1994). The relation `=?` (and similar predicates) are static.

Since there is no real distinction, it is possible to mix internal and external functions in function calls.

Importing and Discarding Elements leanEA does not have constructs for importing or discarding elements. The latter is not needed anyway. If the former useful for an application, the user can simulate "`import v`" by the "`v := import`", where `import` is defined by the function definition

```
define import as X with gensym(f,X).
```
[8]

Local Nondeterminism If the updates of a firing transition are inconsitent, i.e., several updates define a new value for the same location, the first value is chosen (this is called local nondeterminism in (Gurevich, 1994)).

5 Conclusion

We presented leanEA, an approach to implementing an abstract machine for evolving algebras. The underlying idea is to modifying the Prolog reader, such that loading a specification of an evolving algebra means compiling it into Prolog clauses. Thus, the Prolog system itself is turned into an abstract machine for running EAs. The contribution of our work is twofold:

Firstly, leanEA offers an efficient and very flexible framework for simulating EAs. leanEA is open, in the sense that it is easily interfaced with other applications, embedded into other systems, or adapted to concrete needs. We believe that this is a very important feature that is often underestimated: if a specification system is supposed to be used in practice, then it must be embedded in an appropriate system for program development. leanEA, as presented in this paper, is surely more a starting point than a solution for this, but it demonstrates clearly one way for proceeding.

Second, leanEA demonstrates that little effort is needed to implement a simulator for EAs. This supports the claim that EAs are a practically relevant tool, and it shows a clear advantage of EAs over other specification formalisms: these are often

[8] The Prolog predicate `gensym` generates a new atom every time it is called.

hard to understand, and difficult to deal with when implementing them. EAs, on the other hand, are easily understood and easily used. Thus, leanEA shows that one of the major goals of EAs, namely to "bridge the gap between computation models and specification methods" (following Gurevich (1994)), was achieved.

References

Börger, E., Durdanovic, I., & Rosenzweig, D. 1994a. Occam: Specification and Compiler Correctness. *Pages 489–508 of:* Montanari, U., & Olderog, E.-R. (eds), *Proceedings, IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*. North-Holland.

Börger, E., Del Castillo, G., Glavan, P., & Rosenzweig, D. 1994b. Towards a Mathematical Specification of the APE100 Architecture: The APESE Model. *Pages 396–401 of:* Pehrson, B., & Simon, I. (eds), *Proceedings, IFIP 13th World Computer Congress*, vol. 1. Amsterdam: Elsevier.

Börger, Egon, & Rosenzweig, Dean. 1994. A Mathematical Definition of Full Prolog. *Science of Computer Programming.*

Gurevich, Yuri. 1991. Evolving Algebras. A Tutorial Introduction. *Bulletin of the EATCS*, **43**, 264–284.

Gurevich, Yuri. 1994. Evolving Algebras 1993: Lipari Guide. *In:* Börger, E. (ed), *Specification and Validation Methods.* Oxford University Press.

Gurevich, Yuri, & Huggins, Jim. 1993. The Semantics of the C Programming Language. *Pages 273–309 of: Proceedings, Computer Science Logic (CSL).* LNCS 702. Springer.

Gurevich, Yuri, & Mani, Raghu. 1994. Group Membership Protocol: Specification and Verification. *In:* Börger, E. (ed), *Specification and Validation Methods.* Oxford University Press.

Kappel, Angelica M. 1993. Executable Specifications based on Dynamic Algebras. *Pages 229–240 of: Proceedings, 4th International Conference on Logic Programming and Automated Reasoning (LPAR), St. Petersburg, Russia.* LNCS 698. Springer.

O'Keefe, Richard A. 1990. *The Craft of Prolog.* MIT Press.