leanEA: A Lean Evolving Algebra Compiler

Bernhard Beckert[1] and Joachim Posegga[2]

[1] University of Karlsruhe, Institute for Logic, Complexity and Deduction Systems, 76128 Karlsruhe, Germany; beckert@ira.uka.de
[2] Deutsche Telekom AG, Research Centre, 64276 Darmstadt, Germany; posegga@fz.telekom.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 _))) :-
         rearrange(D,B,A).
 rearrange((E,F),(C,D),(A,B)) :-
          rearrange(E,C,B), rearrange(F,D,A).
 rearrange(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 an efficient and flexible simulator for evolving algebra specifications.

1 Introduction

Evolving algebras (EAs) (Gurevich, 1991; Gurevich, 1995) 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; 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, 1995).

When working with EAs, it is very handy to have a simulator at hand for running the specified algebras. 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 written in C. Both implementations are quite sophisticated and offer a convenient language for specifying EAs.

In this paper, we describe a new approach to implementing a simulator for evolving algebras; we focus on deterministic, sequential EAs. Our implementation differs from previous approaches in that it emphasizes on simplicity, flexibility and elegance of the implementation, rather than on sophistication. We present a simple, Prolog-based approach for executing EAs. The underlying idea is to compile 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 its flexibility: the Prolog program we discuss in the sequel[3] 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: Section 2 briefly explains how a deterministic, untyped EA can be programmed in leanEA; no mathematical treatment is given in this section, but it is explained what a user has to do in order to use EAs with leanEA. The implementation of the basic version of leanEA is explained in parallel. This basic version is characterized mathematically in Section 4 by giving the semantics of leanEA programs; Subsection 4.6 summarizes the differences to the standard semantics of EAs, as defined in the Lipari Guide (Gurevich, 1995).

In Section 5, a number of purely syntactical extensions are added for the sake of programming convenience, and more semantical extensions like including typed algebras, or implementing non-deterministic evolving algebras are discussed. Section 6 introduces modularized EAs, where Prolog's concept of modules is used to structure the specified algebras. Finally, we draw conclusions from our research in Section 7.

Through the paper we assume the reader to be familiar with the basic ideas behind evolving algebras, and with the basics of Prolog (see e.g. (O'Keefe, 1990)).

2 leanEA: the Program and its Use

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 evolving algebra "moves" from one state to another, while functions are changed. leanEA is a programming language that allows to program this behavior. From a declarative point of view, a leanEA program is a specification of an EA. In this section, we explain the implementation of leanEA, and how it is used to specify EAs.

[3] The program is available on the *World Wide Web*. The URL for the leanEA home page is `http://i12www.ira.uka.de/leanea`.

2.1 Overview

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

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

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

that specify the initial state of an EA, and *transition definitions*

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

defining the EA's evolving, i.e., the mapping from one state to another.

The syntax of these syntactical constructs is implemented by a couple of Prolog operators as shown in Figure 1, Lines 1–6.[4]

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.

Furthermore, the final state(s) of the EA are not given explicitly in leanEA; 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.

A specified evolving algebra is started by calling the Prolog goal

```
transition _
```

leanEA then recursively searches for applicable transitions and executes them until no more transitions are applicable, or an undefined term is evaluated.

2.2 Representation of States in leanEA

Before explaining how function definitions set up the initial state of an EA, we consider 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 that `f` denotes a partial function mapping 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 either has a value in the current state, or it is undefined.

Technically, this mapping of locations to values is implemented with a dynamic Prolog predicate `=>/2`, (cf. Fig. 1, Line 7) that behaves as follows: The goal "`Loc => Val`" succeeds if `Loc` is bound to a ground Prolog term that is a

[4] Note, that the precedences of operators (those pre-defined by leanEA as well as others used in a leanEA program) can influence the semantics of Prolog goals included in leanEA programs.

```
:- 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,(\)).

:- multifile (=>)/2.
:- dynamic (=>)/2.

term_expansion((define Location as Value with Goal),
               ((Location => Value) :- Goal,!)).

term_expansion((transition Name if Condition then Updates),
               (transition(Name) :-
                  (Condition,!,
                   FrontCode,BackCode,transition(_)))) :-
        rearrange(Updates,FrontCode,BackCode).

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

rearrange((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.

[] =>* [].

(S =? T) :- ([S,T] =>* [Val1,Val2]), Val1 == Val2.
```

Fig. 1. lean*EA*: the Program

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, for example, `f(f(2,3),3)`, `lean`EA uses `=>*/2` as an evaluation predicate: 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 necessarily elements of the universe (contrary to the arguments of a location), but are expressions that are recursively evaluated. For the sake of convenience, one can refer to members of the universe in function calls explicitly: these are denoted by preceding them with a backslash "\"; no evaluation of whatever Prolog term comes after a backslash is performed. We will refer to this as *quoting* in the sequel.

For economical reasons, the predicate `=>*/2` actually maps a *list* of function calls to a list of values. Figure 1, Lines 23–28, shows the Prolog code for `=>*`: if the term to be evaluated (bound to the first argument of the predicate) 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 29). `=>*` fails if the value of the function call is undefined in the current state.

Example 2. 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 goals

```
[f(\f(0,1),\2)] =>* [X]  and  [f(f(0,1),\2)] =>* [X]
```

will fail since, in the first case, the term `f(0,1)` is not an integer but a location, and, in the second case, `0` and `1` are undefined constants (0-ary functions).

2.3 Function Definitions

The initial state of an EA is specified by a sequence of function definitions. They define the initial values of locations by providing 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 leanEA-program (code besides function and transition definitions), since *Goal* may call any Prolog predicate. If several function definitions define values for a single location, the (textually) first definition is chosen.

leanEA translates a function definition into a 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.

Technically, the translation of define statements to a `=>` clauses is implemented by modifying the Prolog reader as shown in Figure 1, Lines 9–10.[5]

Examples for Function Definitions

Constants The following definition assigns the value `1` to the constant `reg1`:

```
define reg1 as 1 with true.
```

Prolog Data Types Prolog Data Types are easily imported into the algebra. Here is how to introduce lists:

```
define X as X with X=[]; X=[H|T].
```

This causes all lists to evaluate to themselves; thus a list in a transition refers to the same list in the universe and needs not to be quoted. Similarly,

```
define X as X with integer(X).
```

introduces Prolog's integers.

Evaluating Functions by Calling Prolog Predicates The following example interfaces Prolog predicates with an evolving algebra:

```
define X+Y as Z with Z is X+Y.
define append(X,Y) as Result with append(X,Y,Result).
```

Input and Output Useful definitions for input and output are

```
define read as X with read(X).
define output(X) as X with write(X).
```

Whilst the purpose of `read` should be immediate, the returning of the argument of `output` might not be clear: the idea is that the returned value can be used in expressions. That is, an expression of the form `f(\1,output(\2))` will print `2` while it is evaluated.

[5] In most Prolog dialects (e.g., SICStus Prolog and Quintus Prolog) the Prolog reader is changed 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.

Necessary Conditions for Function Definitions The design of `lean`EA constrains function definitions in several ways; it is important to understand these restrictions, since they are not checked by `lean`EA. Thus, the programmer of an EA has to ensure that:

1. All computed values are *ground* Prolog terms, and the goals for computing them either fail or succeed (i.e.: terminate) for all possible instantiations that might appear. Prolog exceptions that terminate execution must be avoided as well.[6]
2. The goals do not change the Prolog data base or have any side effects affecting other computations.
3. The goals do not (syntactically) contain a cut "`!`".
4. The goals do not call the `lean`EA internal predicates `transition/1`, `=>*/2`, and `=>/2`.

Violating these rules does not necessarily mean that `lean`EA will not function properly; however, unless one is very well aware of what he/she is doing, we strongly recommend against it.

2.4 Transition Definitions

Transitions specify the evolving of an EA. An applicable transition maps one state of an EA to a new state by changing the value of locations. Transitions are specified as:[7]

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

where

Name is an arbitrary Prolog term (usually an atom).
Condition is a Prolog goal that determines when the transition is applicable. It usually contains calls to the predicate `=?/2` (see Section 2.4 below), and often uses the logical Prolog operators "`,`" (conjunction), "`;`" (disjunction), "`->`" (implication), and "`\+`" (negation).
Updates is a comma-separated sequence of updates of the form

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

[6] Defining `+` by "`define X+Y as Z with integer(X), integer(Y), Z is X+Y.`" is, for instance, safe in this respect.

[7] Guarded multi-updates (if-then-else constructs that may be nested) make EAs more convenient; `lean`EA can be extended to allow guarded multi-updates by changing the predicate `rearrange` such that it (recursively) translates guarded multi-updates into Prolog's if-then-else.

An update $f_i(r_{i1}, \ldots, r_{in_i})$ `:=` v_i $(1 \leq i \leq k)$ 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 simultaneously (i.e., in the old state). If one of the function calls is undefined, the assignment fails.[8]

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

Technically, `lean`*EA* maps a transition of the above form into a Prolog clause

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

Likewise to function definitions, this is achieved by modifying the Prolog reader as shown in Figure 1, Lines 11–15.

Since the updates in transitions must be executed simultaneously, all function calls have to be evaluated before the first assignment takes place. The auxiliary predicate `rearrange/3` (Lines 16–22) serves this purpose: it splits all updates into evaluation code, that uses the predicate `=>*/2`, and into subsequent code for storing the new values by asserting an appropriate `=>/2` clause.[9] The sequential code generated by `rearrange` thus simulates a simultaneous update.

The Equality Relation Besides logical operators, `lean`*EA* allows in the condition of transitions the use of the pre-defined predicate `=?/2` (Fig. 1, Line 30) 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.

2.5 An Example Algebra

We conclude this section with an example for an evolving algebra: Figure 2 shows a `lean`*EA* program which is the specification of an EA for computing $n!$.

[8] 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, i.e., attempts to change the meaning of the backslash are ignored.

[9] To retract the old value from the database and, thus, reduce the memory usage during run-time, one could insert the line

```
( clause((Loc => _),true),retract(Loc => _) ; true ),
```

between Lines 21 and 22. (Note, that the default values specifying the initial state will not be retracted, since these clauses have a non-empty body.) We favored, however, the smaller and slightly faster version that does not retract old values.

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 factorial of `reg1`'s value by decrementing `reg1` and storing the intermediate results in `reg2`. If the value of `reg1` is `1`, 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.

```
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.
```

Fig. 2. An Evolving Algebra for Computing $n!$ (in leanEA Syntax)

3 Some Hints for Programmers

Final States. The basic version of leanEA does not have an explicit construct for specifying the final state of an EA; instead, the algebra is in a final state if no more transition is applicable. A more declarative way to halt an algebra is to evaluate an undefined expression, like "`stop := stop`".

Tracing Transitions. Programming in leanEA—like in any other programming language—usually requires debugging. For tracing transitions, it is often useful to include calls to `write` or `trace` at the end of conditions: the code will be executed whenever the transition fires and it allows to provide information about the state of the EA.

Tracing the Evaluation of Terms. A definition of the form

```
define f(X) as _ with write(f(X)), fail.
```

will trace the evaluation of functions: if the above function definition precedes the "actual" definition of `f(X)`, it will print the expression to be evaluated whenever the evaluation takes place.

Examining States. All defined values of locations in the current state can be listed by calling the Prolog predicate `listing(=>)`. Note, that this does not show any default values.

4 Semantics

This section formalizes the semantics of leanEA programs, in the sense that it explains in detail which evolving algebra is specified by a concrete leanEA-program.

Definition 1. Let P be a leanEA-program; then $\mathcal{D}_P$ denotes the sequence of function definitions in P (in the order in which they occur in P), $\mathcal{T}_P$ denotes the sequence of transition definitions in P (in the order in which they occur in P), and $\mathcal{C}_P$ denotes the additional Prolog-code in P, i.e., P without $\mathcal{D}_P$ and $\mathcal{T}_P$.

The function definitions $\mathcal{D}_P$ (that may call predicates from $\mathcal{C}_P$) specify the initial state of an evolving algebra, whereas the transition definitions specify how the algebra evolves from one state to another.

The signature of evolving algebras is in our approach the set $GTerms$ of all ground Prolog terms. The (single) universe, that is not sorted, is a subset of $GTerms$.

Definition 2. $GTerms$ denotes the set of all ground Prolog terms; it is the *signature* of the evolving algebra specified by a leanEA program.

We represent the states S of an algebra (including the initial state S_0) by an evaluation function $[\![]\!]_S$, mapping locations to the universe. Section 4.1 explains how $[\![]\!]_{S_0}$, i.e., the initial state, is derived from the function definitions $\mathcal{D}$. In what way the states evolve according to the transition definitions in $\mathcal{T}$ (which is modeled by altering $[\![]\!]$) is the subject of Section 4.3.

The final state(s) 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 function calls in its updates (Def. 8).[10]

4.1 Semantics of Function Definitions

A function definition "`define` F `as` R `with` G." gives a procedure for calculating the value of a location $f(t_1, \ldots, t_n)$ $(n \geq 0)$. Procedurally, this works by

[10] The user may, however, explicitly terminate the execution of a leanEA-program (see Section 3).

instantiating F to the location and executing G. If G succeeds, then R is taken as the value of the location. If several definitions provide values for a single location, we use the first one. Note, that the value of a location depends on the additional Prolog code $\mathcal{C}_P$ in a `leanEA`-program P, since G may call predicates from $\mathcal{C}_P$.

Definition 3. Let $\mathcal{D}$ be a sequence of function definitions and $\mathcal{C}$ be additional Prolog code.

A function definition

$$D \;=\; \texttt{define}\ F\ \texttt{as}\ R\ \texttt{with}\ G\texttt{.}$$

in $\mathcal{D}$ is *succeeding* for $t \in GTerms$ with answer $r = R\tau$, if

1. there is a (most general) substitution σ such that $F\sigma = t$;
2. $G\sigma$ succeeds (possibly using predicates from $\mathcal{C}$);
3. τ is the answer substitution of $G\sigma$ (the first answer substitution if $G\sigma$ is not deterministic).

If no matching substitutions σ exists or if $G\sigma$ fails, D is *failing* for t.

The partial function

$$[\![]\!]_{\mathcal{D},\mathcal{C}} \;:\; GTerms \longrightarrow GTerms$$

is defined by

$$[\![t]\!]_{\mathcal{D},\mathcal{C}} = r \;\;,$$

where r is the answer (for t) of the first function definition $D \in \mathcal{D}$ succeeding for t. If no function definition $D \in \mathcal{D}$ is succeeding for t, then $[\![t]\!]_{\mathcal{D},\mathcal{C}}$ is undefined.

The following definition formalizes the conditions function definitions have to meet (see Section 2.3):

Definition 4. A sequence $\mathcal{D}$ of function definitions and additional Prolog code $\mathcal{C}$ are *well defining* if

1. no function definition $D_i \in \mathcal{D}$ is for some term $t \in GTerms$ neither succeeding nor failing (i.e., not terminating), unless there is a definition $D_j \in \mathcal{D}$, $j < i$, in front of D_i that is succeeding for t;
2. if $D \in \mathcal{D}$ is succeeding for $t \in GTerms$ with answer r, then $r \in GTerms$;
3. $\mathcal{D}$ does not (syntactically) contain a cut "`!`";[11]
4. the goals in $\mathcal{D}$ and the code $\mathcal{C}$
   (a) do not change the Prolog data base or have any other side effects;
   (b) do not call the internal predicates `transition/1`, `=>*/2`, and `=>/2`.

Proposition 5. *If a sequence $\mathcal{D}$ of function definitions and additional Prolog code $\mathcal{C}$ are well defining, then $[\![]\!]_{\mathcal{D},\mathcal{C}}$ is a well defined partial function on GTerms (a term mapping).*

[11] Prolog-negation and the Prolog-implication "`->`" are allowed.

A well-defined term mapping $[\![\,]\!]$ is the basis for defining the evaluation function of an evolving algebra, that is the extension of $[\![\,]\!]$ to function calls that are not a location:

Definition 6. Let $[\![\,]\!]$ be a well defined term mapping. The partial function

$$[\![\,]\!]^* \;:\; GTerms \longrightarrow GTerms$$

is defined for $t = f(r_1, \ldots, r_n) \in GTerms$ $(n \geq 0)$ as follows:

$$[\![t]\!]^* = \begin{cases} s & \text{if } t = \backslash s \\ [\![f([\![r_1]\!]^*, \ldots, [\![r_n]\!]^*)]\!] & \text{otherwise} \end{cases}$$

4.2 The Universe

A well-defined term mapping $[\![\,]\!]_{\mathcal{D}_P,\mathcal{C}_P}$ enumerates the universe $\mathcal{U}_P$ of the evolving algebra specified by P; in addition, $\mathcal{U}_P$ contains all quoted terms (without the quote) occurring in P:

Definition 7. If P is a LeanEA program, and $[\![\,]\!]_{\mathcal{D}_P,\mathcal{C}_P}$ is a well defined term mapping, then the universe $\mathcal{U}_P$ is the union of the co-domain of $[\![\,]\!]_{\mathcal{D}_P,\mathcal{C}_P}$, i.e.,

$$[\![GTerms]\!]_{\mathcal{D}_P,\mathcal{C}_P} = \{[\![t]\!]_{\mathcal{D}_P,\mathcal{C}_P} \;:\; t \in GTerms,\, [\![t]\!]_{\mathcal{D}_P,\mathcal{C}_P}\downarrow\} \;,$$

and the set

$$\{t \;:\; t \in GTerms,\, \backslash t \text{ occurs in } P\} \;.$$

Note, that (obviously) the co-domain of $[\![\,]\!]^*$ is a subset of the universe, i.e.,

$$[\![GTerms]\!]^*_{\mathcal{D}_P,\mathcal{C}_P} \subset \mathcal{U}_P \;.$$

The universe $\mathcal{U}_P$ as defined above is not necessarily decidable. In practice, however, one usually uses a decidable universe, i.e., a decidable subset of $GTerms$ that is a superset of $\mathcal{U}_P$ (e.g. $GTerms$ itself). This can be achieved by adding function definitions and thus expanding the universe.[12]

4.3 Semantics of Transition Definitions

After having set up the semantics of the function definitions, which constitute the initial evaluation function and thus the initial state of an evolving algebra, we proceed with the dynamic part.

The transition definitions $\mathcal{T}_P$ of a LeanEA-program P specify how a state S of the evolving algebra represented by P maps to a new state S'.

[12] It is also possible to change Definition 7; that, in its current form, defines the minimal version of the universe.

Definition 8. Let S be a state of an evolving algebra corresponding to a well defined term mapping $[\![\,]\!]_S$, and let $\mathcal{T}$ be a sequence of transition definitions.

A transition

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

is said to *fire*, if the Prolog goal *Condition* succeeds in state S (possibly using the predicate `=?/2`, Def. 10).

Let

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

$(k \geq 1, n_i \geq 0)$ be the sequence *Updates* of the first transition in $\mathcal{T}$ that fires. Then the term mapping $[\![\,]\!]_{S'}$ and thus the state S' are defined by

$$[\![t]\!]_{S'} = \begin{cases} [\![v_i]\!]^*_S & \text{if there is a smallest } i,\ 1 \leq i \leq k, \\ & \text{such that } t = f_i([\![r_{i1}]\!]^*_S,\ldots,[\![r_{in_i}]\!]^*_S) \\ [\![t]\!]_S & \text{otherwise} \end{cases}$$

If $[\![\,]\!]^*_S$ is undefined for one of the terms r_{ij} or v_i, $1 \leq i \leq k$, $1 \leq j \leq n_i$ of the first transition in $\mathcal{T}$ that fires, or if no transition fires, then the state S is *final* and $[\![\,]\!]_{S'}$ is undefined.

Proposition 9. *If $[\![\,]\!]_{S'}$ is a well defined term mapping, then $[\![\,]\!]_{S'}$ (as defined in Def. 8) is well defined.*

4.4 The Equality Relation

Besides "`,`" (and), "`;`" (or), "`\+`" (negation), and "`->`" (implication) `lean`*EA* allows in conditions of transitions the pre-defined predicate `=?/2`, that implements the equality relation for examining the current state:

Definition 10. In a state S of an evolving algebra (that corresponds to the well defined term mapping $[\![\,]\!]_S$), for all $t_1, t_2 \in \mathit{GTerms}$, the relation t_1 `=?` t_2 holds iff

$$[\![t_1]\!]^*_S \downarrow \text{ and } [\![t_2]\!]^*_S \downarrow, \qquad \text{and} \qquad [\![t_1]\!]^*_S = [\![t_2]\!]^*_S.$$

4.5 Runs of `lean`*EA*-programs

A run of a `lean`*EA*-program P is a sequence of states $S_0, S_1, S_2, \ldots$ of the specified evolving algebra. Its initial state S_0 is given by

$$[\![\,]\!]_{S_0} = [\![\,]\!]_{\mathcal{D}_P,\mathcal{C}_P}$$

(Def. 6). The following states are determined according to Definition 8 and using

$$S_{n+1} = (S_n)' \qquad (n \geq 0)\ .$$

This process continues iteratively until a final state is reached.

Proposition 11. leanEA *implements the semantics as described in this section; i.e., provided* $[\![]\!]_{\mathcal{D}_P,\mathcal{C}_P}$ *is well defined,*

1. *in each state* S *of the run of a* leanEA*-program* P *the goal "*`[`t`] =>* `[X]`*" succeeds and binds the Prolog variable* `X` *to* u *iff* $[\![t]\!]^*_S = u$*;*
2. *the execution of* P *terminates in a state* S *iff* S *is a final state;*
3. *the predicate* `=?` *implements the equality relation.*

4.6 Peculiarities of leanEA's Semantics

The Lipari Guide (Gurevich, 1995) defines what is usually understood as the standard semantics of EAs. Although leanEA is oriented at this semantics, there are a couple of details where leanEA's semantics differ. The reason for the differences is not that the standard semantics could not be implemented, but that we decided to compromise for the sake of elegance and clearness of our program.

leanEA complies with the semantics given in (Gurevich, 1995) with the following exceptions:

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, see Section 5.2) is represented by succeeding (resp. failing) of the corresponding predicate.

Undefined Function 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 as the last function definition:

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

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, 1995). 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.

External functions are reiterated (i.e., evaluated multiply in one state) if they occur multiply in a term being evaluated. If an external function is non-deterministic (an oracle), this can lead to inconsistencies and should be avoided.

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

```
define import as X with gensym(f,X).
```
[13]

Local Non-determinism If the updates of a firing transition are inconsistent, i.e., several updates define a new value for the same location, the first value is chosen (this is called local non-determinism in (Gurevich, 1995)).

5 Extensions

5.1 The `let` Instruction

It is often useful to use local abbreviations in a transition. The possibility to do so can be implemented by adding a clause

```
rearrange((let Var = Term),
          ([Term] =>* [Val],Var = \Val), true).
```

to `leanEA`.[14] Then, in addition to updates, instructions of the form

`let` x `=` t

can be used in the update part of transitions, where x is a Prolog variable and t a Prolog term. This allows to use x instead of t in subsequent updates (and `let` instructions) of the same transition. A variable x must be defined only once in a transition using `let`. Note, that x is bound to the quoted term $\backslash[\![t]\!]^*$; thus, using an x inside another quoted term may lead to undesired results (see the first part of Example 3).

Example 3. "`let X = \a, reg := \f(X)`" is equivalent to "`reg := \f(\a).`" (which is different from "`reg := \f(a).`").

```
let X = \b,
let Y = f(X,X),
reg1 := g(Y,Y),
reg2(X) := X.
```
is equivalent to
```
reg1 := g(f(\b,\b),f(\b,\b)),
reg2(\b) := \b.
```

Using `let` not only shortens updates syntactically, but also enhances efficiency, because function calls that occur multiply in an update do not have to be re-evaluated.

[13] The Prolog predicate `gensym` generates a new atom every time it is called.
[14] And defining the operator `let` by adding "`:- op(910,fx,(let)).`".

5.2 Additional Relations

The Prolog predicate `=?`, that implements the equality relation (Def. 10), is the only one that can be used in the condition of a transition (besides the logical operators). It is possible to implement similar relations using the leanEA 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

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

to leanEA.[15] 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` $p'(x_1, \ldots, x_n)$ `as true with` *Code*`.`

(which is the standard way of implementing relations using function definitions). Note, that p fails, if one of $[\![t_1]\!]^*_S, \ldots, [\![t_n]\!]^*_S$ is undefined in the current state S.

Example 4. The predicate `<>` implements the is-not-equal relation: t_1 `<>` t_2 succeeds iff $[\![t_1]\!]^* \downarrow$, $[\![t_2]\!]^* \downarrow$, and $[\![t_1]\!]^* \neq [\![t_2]\!]^*$. `<>` is implemented by adding the following clause to leanEA:

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

5.3 Non-determinism

It is not possible to define non-deterministic EAs in the basic version of leanEA. If more than one transition fire in a state, the first is chosen.

This behavior can be changed—such that non-deterministic EAs can be executed—in the following way:

- The cut from Line 13 has to be removed. Then, further firing transitions are executed if backtracking occurs.
- A "retract on backtrack" must be added to the transitions for removing the effect of updates and restoring the previous state if backtracking occurs. Line 22 has to be changed to

  ```
  ( asserta(Loc => Val) ; (retract(Loc => Val),fail) ).
  ```

Now, leanEA will enumerate all possible sequences of transitions. Backtracking is initiated, if a final state is reached, i.e., if the further execution of a leanEA program fails.

The user has to make sure that there is no infinite sequence of transitions (e.g., by imposing a limit on the length of sequences).

Note, that usually the number of possible transition sequences grows exponentially in their length, which leads to an enormous search space if one tries to find a sequence that ends in a "successful" state by enumerating all possible sequences.

[15] $x_1, \ldots, x_n$ must be n distinct and uninstantiated Prolog variables when `=>*` is called. Thus, "`(S =? T) :- ([S,T] =>* [V1,V2]), V1 == V2.`", rather than "`(S =? T) :- ([S,T] =>* [V,V])`" is needed for implementing "`=?`".

6 Modularized Evolving Algebras

One of the main advantages of EAs is that they allow a problem-oriented formalization. This means, that the level of abstraction of an evolving algebra can be chosen as needed. In the example algebra for computing $n!$ in Section 2.5 for instance, we simply used Prolog's arithmetics over integers and did not bother to specify what multiplication or subtraction actually means. In this section, we demonstrate how such levels of abstraction can be integrated into `lean`EA; the basic idea behind it is to exploit the module-mechanism of the underlying Prolog implementation.

6.1 The Algebra Declaration Statement

In the modularized version of `lean`EA, each specification of an algebra will become a Prolog module; therefore, each algebra must be specified in a separate file. For this, we add an *algebra declaration statement* that looks as follows:

```
algebra Name(In,Out)
      using [Include-List]
      start Updates
      stop Guard.
```

Name is an arbitrary Prolog atom that is used as the name of the predicate for running the specified algebra, and as the name of the module. It is required that *Name*`.pl` is also the file name of the specification and that the algebra-statement is the first statement in this file.

In, Out are two lists containing the input and output parameters of the algebra. The elements of *Out* will be evaluated if the algebra reaches a final state (see below).

Include-List is a list of names of sub-algebras used by this algebra.

Updates is a list of updates; it specifies that part of the initial state of the algebra (see Section 2.4), that depends on the input *In*.

Guard is a condition that specifies the final state of the evolving algebra. If *Guard* is satisfied in some state, the computation is stopped and the algebra is halted (see Section 2.4).

Example 5. As an example consider the algebra statement in Figure 3: an algebra `fak` is defined that computes $n!$. This is a modularized version of the algebra shown in Section 2.5. The transitions `start` and `result` are now integrated into the algebra statement.

The last function definition in the algebra is of particular interest: it shows how the sub-algebra `mult`, included by the algebra statement, is called. Where the earlier algebra for computing $n!$ in Section 2.5 used Prolog's built-in multiplication, a sub-algebra for carrying out multiplication is called. Its definition can be found in Figure 4.

```
algebra  fak([N],[reg2])
  using  [mult]
  start  reg1 := N,
         reg2 := 1
  stop   reg1 =? 1.

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 mult([X,Y],[R]).

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

Fig. 3. A Modularized EA for Computing $n!$

```
algebra  mult([X,Y],[result])
  using  []
  start  reg1 := X,
         reg2 := Y,
         result := 0
  stop   reg1 =? 0.

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   \+(reg1 =? 0)
     then reg1 := (reg1-1),
          result := (result+reg2).
```

Fig. 4. A Modularized EA for Multiplication

6.2 Implementation of Modularized EAs

The basic difference between the basic version of `leanEA` and the modularized version is that the `algebra`-statement at the beginning of a file containing an EA specification is mapped into appropriate `module` and `use_module` statements in Prolog. Since the algebra will be loaded within the named module, we also need an evaluation function that is defined internally in this module. This allows to use functions with the same name in different algebras without interference.

Figure 5 lists the modularized program. It defines four additional opera-

```
:- 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,(\)),           op(1199,fx,(algebra)),
   op(1190,xfy,(start)),     op(1170,xfx,(stop)),
   op(1180,xfy,(using)).

term_expansion((algebra Head using Include_list
                      start Updates stop Guard),
               [(:- module(Name,[Name/2])),
                Name:(:- use_module(Include_list)),
                (:- dynamic(Name:(=>)/2)),
                Name:(([H|T] =>* [HVal|TVal]) :-
                     (   H = \HVal
                       ;   H  =.. [Func|Args], Args =>* ArgVals,
                       H1 =.. [Func|ArgVals], H1 => HVal),!,
                       T =>* TVal),
                Name:([] =>* []),
                Name:((A =? B) :- ([A,B] =>* [Val1,Val2]),
                                  Val1 == Val2),
                Name:(NewHead :- FrontCode,BackCode,!,
                         (transition _),Out =>* Value),
                Name:(transition(result) :- (Guard,!))]):-
        Head =..[Name,In,Out], NewHead =..[Name,In,Value],
        rearrange(Updates,FrontCode,BackCode).

term_expansion((define Location as Value with Goal),
               ((Location => Value) :- Goal,!)).

term_expansion((transition Name if Condition then Updates),
               (transition(Name) :-
                  (Condition,!,
                   FrontCode,BackCode,transition(_)))) :-
        rearrange(Updates,FrontCode,BackCode).

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

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

Fig. 5. Modularized EAs: the Program

tors (`algebra`, `start`, `stop`, and `using`) that are needed for the algebra statement. The first `term_expansion` clause (Lines 9–26) translates such a statement into a Prolog module header, declares `=>/2` to be dynamic in the module, and defines the evaluation predicate `=>*` for this module.[16] The effect of the `term_expansion`-statement is probably best seen when setting an example: the module declaration in Figure 3, for instance, is mapped into

```
:- module(fak,[fak/2]).
fak:(:-use_module([mult])).
:- dynamic fak:(=>)/2.
```

plus the usual definition of `=>*/2`.

6.3 Running Modularized EAs

In contrast to the basic version of `leanEA`, a modularized EA has a defined interface to the outside world: The algebra-statement defines a Prolog predicate that can be used to run the specified EA. Thus, the user does not need to start the transitions manually. Furthermore, the run of a modularized EA does not end with failure of the starting predicate, but with success. This is the case since a modularized EA has a defined final state. If the predicate succeeds, the final state has been reached.

For the example algebra above (Figure 3), the run proceeds as follows:

```
| ?- compile([leanea,fak]).
{leanea.pl compiled, 190 msec 4768 bytes}
{compiled mult.pl in module mult, 120 msec 10656 bytes}
{compiled fak.pl in module fak, 270 msec 20944 bytes}
yes

| ?- fak(6,Result).
Result = [720] ?
yes
```

After compiling `leanEA`, the EA shown in Figure 3 is loaded from the file `fak.pl`, which in turn loads the algebra for multiplication from `mult.pl`. The algebra is then started and the result of 6! is returned. The computation takes roughly one second on a Sun SPARC 10.[17]

---

[16] This implementation is probably specific for SICStus Prolog and needs to be changed to run on other Prolog systems. The "`Name:`"-prefix is required in SICStus, because a "`:- module(...)`"-declaration becomes effective *after* the current term was processed.

[17] Recall that all multiplications are carried out within the EA and not by Prolog's internal multiplication predicate.

7 Conclusion

We presented leanEA, an approach to simulation evolving algebra specifications. 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 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 (1995)), was achieved.

References

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

Börger, Egon, Durdanovic, Igor, & Rosenzweig, Dean. 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, Egon, Del Castillo, Giuseppe, Glavan, P., & Rosenzweig, Dean. 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.

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

Gurevich, Yuri. 1995. 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. 1995. 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.