In *Proc. Fifth International Symposium on Programming Language Implementation and Logic Programming*, Tallinn (Estonia), pp. 28–43, Springer LNCS 714, 1993

A Debugging Model for Functional Logic Programs*

Michael Hanus[1] and Berthold Josephs[2]

[1] Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
e-mail: `michael@mpi-sb.mpg.de`
[2] Informatik V, Universität Dortmund, D-44221 Dortmund, Germany

Abstract. This paper presents a box-oriented debugging model for the functional logic language ALF. Due to the sophisticated operational semantics of ALF which is based on innermost basic narrowing with simplification, the debugger must reflect the application of the different computation rules during program execution. Hence our debugging model includes not only one box type as in Byrd's debugging model for logic programs but several different kinds of boxes corresponding to the various computation rules of the functional logic language (narrowing, simplification etc.). Moreover, additional box types are introduced in order to allow skips over (sometimes) uninteresting program parts like proofs of the condition in a conditional equation. Since ALF is a genuine amalgamation of functional and logic languages, our debugging model subsumes operational aspects of both kinds of languages. As a consequence, it can be also used for pure logic languages, pure functional languages with eager evaluation, or functional logic languages with a less sophisticated operational semantics like SLOG or eager BABEL.

1 Introduction

The interest in the amalgamation of functional and logic programming languages has been increased during the last years (see [5] for a survey). Such integrated languages have at least two advantages. In comparison with pure functional languages, functional logic languages have more expressive power due to the availability of features like function inversion, partial data structures and logic variables [25]. In comparison with pure logic languages, functional logic languages have a more efficient operational behavior since functions allow deterministic evaluations if arguments are sufficiently instantiated [13]. Recently, functional logic languages became relevant for practical applications because efficient implementations have been developed [1, 4, 12, 19, 20, 21, 28]. Therefore there is a need for debugging tools for such kind of languages. Since the operational semantics of these languages is different from pure logic languages, we cannot easily adopt an existing debugging framework from

* Please address correspondence to the first author. The research described in this paper was partially funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9103. The responsibility for the contents of this publication lies with the authors.

logic programming. Hence we develop a new debugging model for ALF, a functional logic language which combines the nondeterministic computation principle of logic programming (resolution) with the deterministic computation principle of functional programming (reduction). Our debugging model is based on Byrd's box model for logic programs [3] but refined in two directions. Firstly, the four ports of Byrd's model are enriched by new ports in order to allow the observation of the head unification [8, 24, 26] which is very important in a language which distinguishes between matching and unification. Secondly, new box types are introduced in order to reflect the different computation rules of the functional logic language.

In the next section we give a description of ALF's operational semantics. After a short outline of the standard debugging model for pure logic programs in Section 3 we present in Section 4 the new debugging model corresponding to ALF's execution principles. Comments to the current implementation are given in Section 5 and Section 6 discusses applications of the debugging model.

2 The execution principles of ALF

Different execution principles have been proposed for functional logic languages. A sound and complete operational semantics is usually based on narrowing [9, 17]. Since pure narrowing is extremely nondeterministic and creates a huge search space, refined narrowing strategies are used in functional logic languages. For instance, SLOG [10] is based on innermost narrowing, K-LEAF [1] and BABEL [20] use a lazy strategy, and ALF [11, 12] combines innermost basic narrowing with simplification between narrowing steps. Since the latter strategy prefers deterministic computations, it can be shown that ALF programs are more efficiently executed than equivalent logic programs [13]. Therefore we are interested in this strategy and we will develop a debugger for such kind of programs. However, we remark that this debugging model is general enough to be applicable to other functional logic languages with an eager evaluation principle (cf. Section 6.2). Before presenting the debugging model we describe ALF's operational semantics in more detail.

ALF is a constructor-based language, i.e., the user must specify for each symbol whether it is a constructor or a defined function. Constructors must not be the outermost symbol of the left-hand side of a defining equation, i.e., constructor terms are always irreducible. Hence constructors are used to build data types, and defined functions are operations on these data types.

An ALF program is a set of conditional equations.[3] Equations define functions and are used in two ways. In a narrowing step an equation is applied to *compute*

[3] ALF has more features than presented in this paper, e.g., a module system with parameterization, a type system based on many-sorted logic, predicates which are resolved by resolution etc. [11]. We omit these features in this paper because they have no interesting influence on the debugging model (note that predicates can also be considered as Boolean functions).

```
module lists.

  datatype elem = { a ; b ; c }.
  datatype list = { '.'(elem,list) ; [] }.
  func append: list, list -> list.

rules.

  append([],L)  = L.
  append([E|R],L) = [E|append(R,L)].

end lists.
```

Figure 1. ALF program for concatenating lists

a solution of a goal (i.e., variables in the goal may be bound to terms), whereas in a rewrite step an equation is applied to *simplify* a goal (i.e., without binding goal variables). Therefore we distinguish between *narrowing rules* (equations applied in narrowing steps) and *rewrite rules* (equations applied in rewrite steps). Usually, all conditional equations of an ALF program are used as narrowing and rewrite rules, but it is also possible to specify additional rules which are only used for rewriting.

Figure 1 shows an ALF module which defines lists and a concatenation function on lists. `a`, `b` and `c` are the constructors of the data type `elem` and lists are defined as in Prolog. The two equations (with empty conditions) in this module define the function `append` for concatenating two lists.

The *declarative semantics* of ALF is the well-known Horn clause logic with equality as to be found in [23]. The *operational semantics* of ALF is based on innermost basic narrowing with normalization. In the following description of this operational semantics we distinguish two kinds of nondeterminism by the keywords "don't know" and "don't care": *don't know* indicates a branching point in the computation where all alternatives must be explored (by a backtracking strategy in our implementation); *don't care* indicates a branching point where it is sufficient to select one alternative and disregard all other possibilities. We represent a goal (a list of equations to be solved) by a skeleton and an environment part [16, 22]: the *skeleton* is a list of equations composed of terms occurring in the original program, and the *environment* is a substitution which has to be applied to the equations in order to obtain the actual goal. The initial goal G is represented by the pair $\langle G; id\rangle$ where id is the identity substitution. The following scheme describes the operational semantics (if π is a position in a term t, then $t|_\pi$ denotes the subterm of t at position π and $t[s]_\pi$ denotes the term obtained by replacing the subterm $t|_\pi$ by s in t [6]; π is called an *innermost position* of t if the subterm $t|_\pi$ has a defined function symbol at the top and all argument terms consist of variables and constructors). Let $\langle E_1, \ldots, E_n \; ; \; \sigma\rangle$ be a given goal ($E_1, \ldots, E_n$ are the skeleton equations and σ is the environment):

1. Select *don't care* a non-variable position π in E_1 and a new variant $l = r \leftarrow C$ of a rewrite rule such that σ' is a substitution with $\sigma(E_1|_\pi) = \sigma'(l)$ and the goal $\langle C \; ; \; \sigma'\rangle$ can be derived to the empty goal without instantiating any variables

from $\sigma(E_1)$. Then

$$\langle E_1[\sigma'(r)]_\pi, E_2, \ldots, E_n \ ; \ \sigma \rangle$$

is the next goal derived by **rewriting**; go to 1. Otherwise go to 2.

2. If the two sides of equation E_1 have different constructors at the same outer position (a position not belonging to arguments of functions), then the whole goal is **rejected**, i.e., the proof fails. Otherwise go to 3.
3. Let π be the leftmost-innermost position in E_1 (if there exists no such position in E_1, go to 4). Select *don't know* (a) or (b):
   (a) Select *don't know* a new variant $l = r \leftarrow C$ of a narrowing rule such that $\sigma(E_1|_\pi)$ and l are unifiable with most general unifier (mgu) σ'. Then

   $$\langle C, E_1[r]_\pi, E_2, \ldots, E_n \ ; \ \sigma' \circ \sigma \rangle$$

   is the next goal derived by **innermost basic narrowing**; go to 1. Otherwise: fail.

   (b) Let x be a new variable and σ' be the substitution $\{x \mapsto \sigma(E_1|_\pi)\}$. Then

   $$\langle E_1[x]_\pi, E_2, \ldots, E_n \ ; \ \sigma' \circ \sigma \rangle$$

   is the next goal derived by **innermost reflection**; go to 3 (this corresponds to the elimination of an innermost redex and it is only necessary in the presence of partially defined functions [16]).
4. If E_1 is the equation $s = t$ and there is a mgu σ' for $\sigma(s)$ and $\sigma(t)$, then

   $$\langle E_2, \ldots, E_n \ ; \ \sigma' \circ \sigma \rangle$$

   is the next goal derived by **reflection**; go to 1. Otherwise: fail.

In the actual ALF implementation the *don't care* nondeterminism during rewriting (step 1) is implemented by an innermost strategy, i.e., rewriting is performed from innermost to outermost positions, and the *don't know* nondeterminism in narrowing steps (step 3) is implemented by a backtracking strategy as in Prolog.

This operational semantics may look complicated at first sight, but it is a consistent realization of the execution principle "prefer deterministic computations as long as possible" (i.e., apply deterministic rewrite steps before nondeterministic narrowing steps). This yields an efficient operational behavior compared to Prolog's nondeterministic resolution principle but without loosing completeness as in other efficient approaches to execute functional logic programs (cf. [15]). A more detailed discussion of the completeness of this operational semantics and the advantages of it in comparison to other execution principles can be found in [12, 13]. We want to point out that ALF's operational semantics can be implemented with the same efficiency as current Prolog implementations by extending Warren's Abstract Machine to deal with functional computations [12, 14]. Moreover, the search space of ALF programs may be smaller than equivalent Prolog programs due to rewriting and rejection. For instance, the execution of the following goal fails w.r.t. the list module (cf. Figure 1):

`append(append([a|L1],L2),L3) = [b|L4]`

$\vdash$ *rewriting the innermost call to* `append`*:*

`append([a|append(L1,L2)],L3) = [b|L4]`

$\vdash$ *rewriting the outermost call to* `append`*:*

`[a|append(append(L1,L2),L3)] = [b|L4]`

$\vdash$ *rejection (*`a` *and* `b` *are different constructors):*

fail

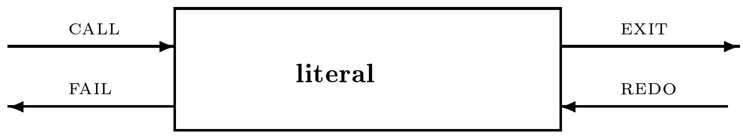
On the other hand the equivalent (flattened) Prolog goal

```
append([a|L1],L2,L), append(L,L3,[b|L4])
```

causes an infinite loop for any order of literals and clauses of the Prolog program for `append`. This example shows that the simplification process followed by the rejection rule is essential for the improved efficiency of ALF programs (see [13] for more details).[4] Therefore a debugger must show the (successful) application of rewriting and rejection to the programmer. This requires an extension of the standard box-oriented debugging model for Prolog [3, 8] to these new computation rules. Before we show such an extended debugging model in Section 4, we will shortly review the standard debugging model for logic programs in the next section.

3 The standard box-oriented debugger for logic programs

Byrd's debugging model [3] has been used as the standard source-level debugger in many Prolog systems. It is based on the idea that during the computation process a box of the following kind is associated to each literal:



This box is created when the literal should be proved for the first time. The box is entered through the CALL port. If the literal is successfully proved, the box is left through the EXIT port, otherwise (if the proof fails) through the FAIL port. If it is necessary to find an alternative proof for this literal (due to the failure of a subsequent literal), then the box is entered again through the REDO port. Depending on success or failure of finding an alternative proof, the box is left through the EXIT or FAIL port. Note that the boxes have a recursive structure: if a clause is used for the proof of the literal, then new boxes are created inside this box for each literal in the body of the clause.

The basic principle of this debugging model is the observability of these four ports: the ports are the only visible points in the computation process, i.e., the

[4] For instance, "generate-and-test" programs are executed in ALF with a lower complexity than in Prolog.

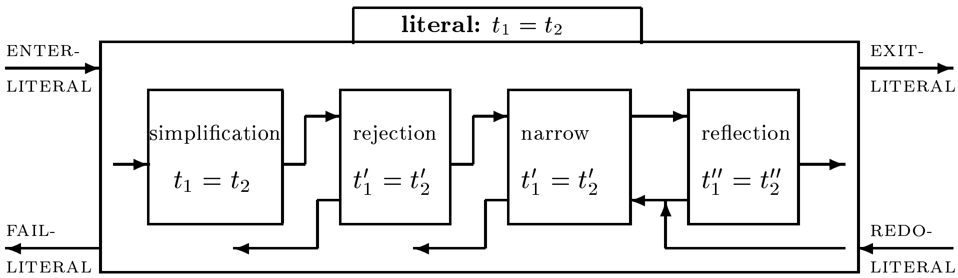
debugger or tracer[5] outputs the ports together with the literal. During the debugging process, the user can turn off the observability of some ports or he can skip from one port to the next port of the same box in order to omit unnecessary details of a subcomputation.

It has been criticized that this four-port debugging model is too weak to explain the control flow of logic programs to the user. For instance, the user cannot see the reason of a failure, i.e., it is not visible whether there are no clauses for a literal or the clause heads do not unify with the literal. Therefore several refinements of this standard debugging model have been proposed in order to visualize the head unification process [8, 24, 26]. Since the difference between matching and unification is important in the operational semantics of functional logic languages (compare definition of rewriting and narrowing in Section 2), we will also propose such a refined debugging model in the next section.

4 A debugging model for functional logic programs

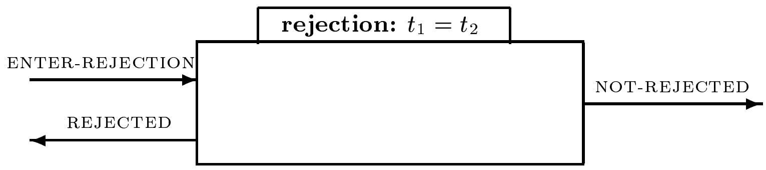
The standard box model for Prolog is used as an interface between the program execution and the programmer. Each box represents the proof of a literal and the programmer can stop and observe the proof at the ports of a box. Moreover, he can set spy points on some ports and skip from one port to another in order to skip over uninteresting details of the execution. In order to provide a similar debugging model for ALF, it is necessary to introduce new box types for the different computation rules (simplification, rejection etc.) and for the new logical units in a proof (e.g., simplification of an entire literal, proving the condition in a conditional equation). Therefore the box-oriented debugger for ALF is based on the following box types:

Literal box: In order to allow the programmer to skip over the proof of a literal (equation), there is a box for each literal as in Byrd's box model [3]. Since a literal is proved by applying simplification, rejection, narrowing, and reflection, a literal box contains four other boxes which correspond to the ongoing computation w.r.t. these rules. Hence the literal box has the following structure (if the literal does not contain any defined function symbol, the simplification and narrow boxes are omitted):

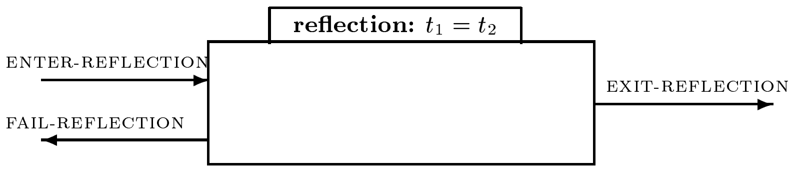


[5] Standard Prolog debuggers show a trace of the program execution to the user. Therefore this part of the debugger is also called *tracer*. Although we will describe only the trace component of our debugger, we will use the more general term "debugger" in this paper.

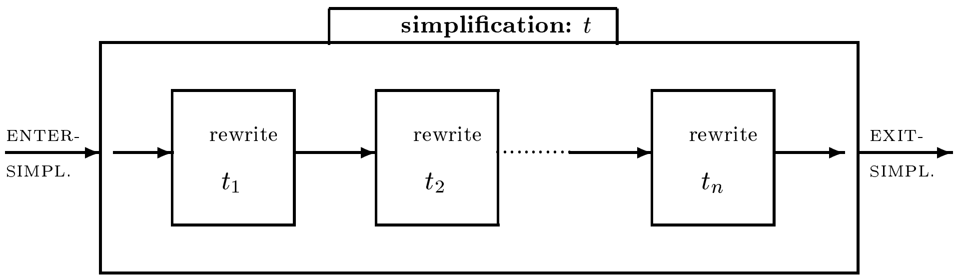
Rejection box: This box corresponds to an application of the rejection rule to an equation. If the equation has different constructors at the same outer position, the equation is *rejected*, otherwise *not rejected*. For instance, the equation `[a|append(L,[])]=[b|M]` is rejected while the equation `append(L,[])=[a|M]` is not rejected. The rejection box has no REDO port because rejection is a deterministic test:



Reflection box: This box corresponds to an application of the reflection rule to an equation. If the two sides of the equation are unifiable, the box is left with success, otherwise with failure. Similarly to the rejection box, this box has no REDO port:

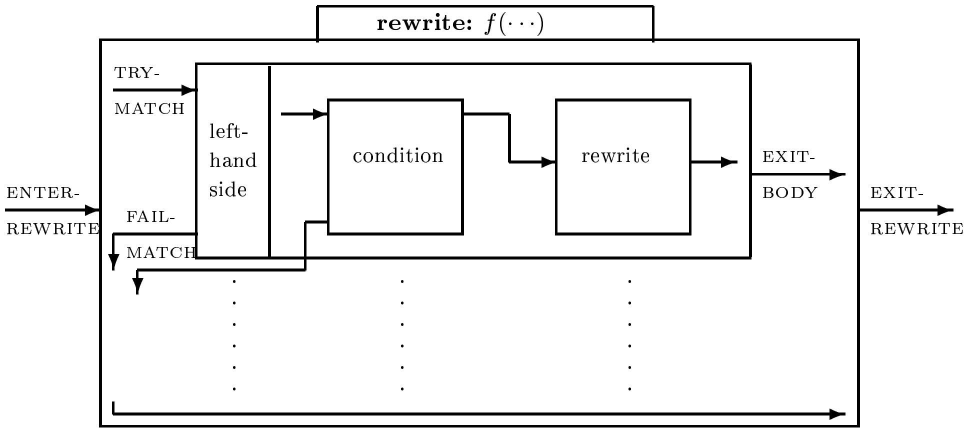


Simplification box: This box corresponds to the simplification of an entire term (or equation). It contains a rewrite box for each function symbol in the term in leftmost-innermost order (e.g., a simplification box for `append(append([a|V],W),Y)` contains a first rewrite box for `append([a|V],W)` and a second rewrite box for the outermost call to `append`). This box has no REDO port because simplification is a deterministic process. Moreover, it has no FAIL port because simplification computes the normal form of a term and hence it is always successful.

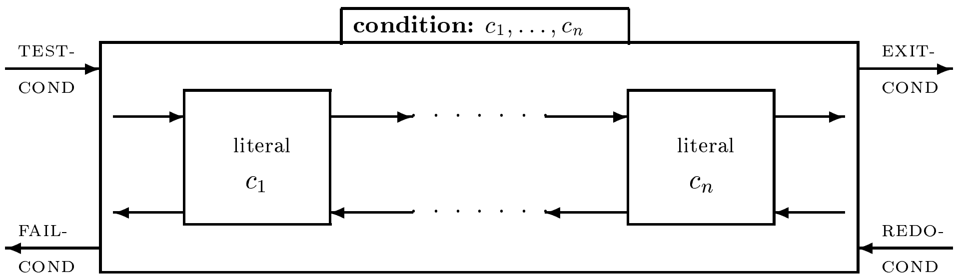


Note that this box is not essentially necessary since it represents no particular computation rule of the operational semantics. However, this box is useful to structure the entire proof process: if the programmer is not interested in the details of the simplification process between two narrowing steps, he can simply skip from the ENTER-SIMPLIFICATION port to the EXIT-SIMPLIFICATION port (see also Section 5).

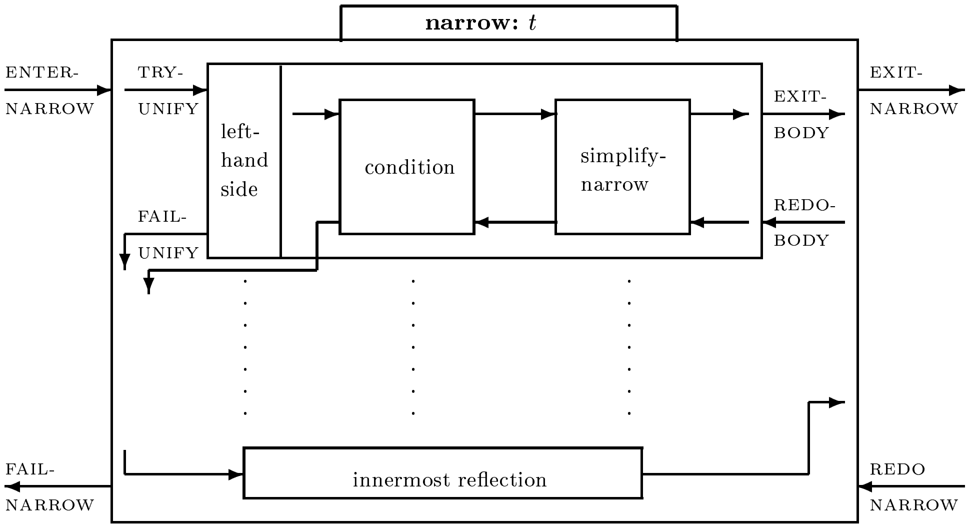
Rewrite box: This box corresponds to the application of a rewrite rule at a subterm. It contains a box for each rule defining the function at the subterm's head (these inner boxes are similar to the OR-boxes of the refined box model in [26]). Such a rule can be applied if the left-hand side *matches* the subterm and the condition is provable. In this case the subterm is replaced by the right-hand side and the right-hand side is simplified by creating a rewrite box for each function symbol occurring in the right-hand side (in the following figure it is assumed that the right-hand side contains only one defined function symbol). The condition box in a rule box is omitted if the rule does not contain a condition. The FAIL-MATCH port of a rule box is connected to the TRY-MATCH port of the subsequent rule box. But note that the FAIL-MATCH port of the last equation is connected to the exit port of the whole rewrite box because the subterm is in normal form if no equation is applicable.



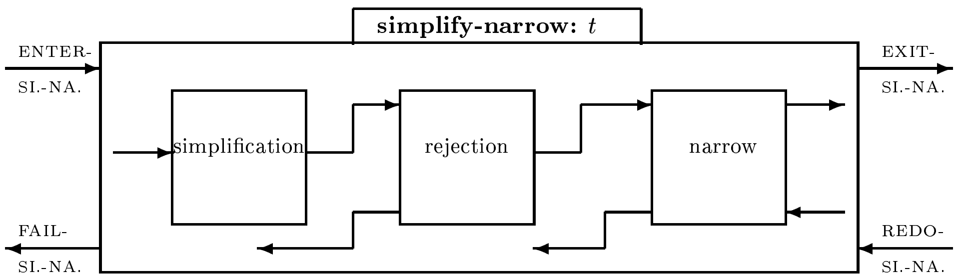
Condition box: This box covers the proof of the entire condition of a conditional rewrite or narrowing rule. It is introduced in order to skip over the proof of the condition of a rule. This box simply contains the literals (equations) in the condition (the REDO-COND port is not used in case of rewrite rules):



Narrow box: The structure of this box is very similar to the rewrite box but it has in addition to the boxes for each defining rule an innermost reflection box as the final rule which is necessary for partially defined functions. In contrast to the rewrite box, the right-hand side of a narrowing rule cannot be represented by a sequence of boxes corresponding to the defined function symbols occurring in the right-hand side. This is due to the fact that after replacing the subterm by the right-hand side in a narrowing step the whole term is simplified and then checked for rejection before the next narrowing step takes place. Since the simplification process may change the whole structure of the term, the subterm where the next narrowing rule will be applied is not fixed after the application of the narrowing rule. Hence the narrow box as well as the simplify narrow box (see below) have the whole term or literal as a parameter and the narrowing rule is applied at the leftmost-innermost position of this term. Note that due to the innermost reflection rule (which is always applicable) narrowing cannot fail. However, an ALF programmer can explicitly prevent the application of the innermost reflection rule by declaring a function as "total". It is a programming error if no narrowing rule is applicable to total functions. In order to show such errors to the programmer, the debugging model contains also a FAIL port in the narrow box.



Simplify narrow box: As mentioned above this box covers the simplification, rejection and narrowing performed after each narrowing rule. Hence it has the following structure:



Now we have described all box types of ALF's debugging model. At first sight the increased number of boxes seems to be confusing. But we think that these boxes are necessary to give the user the right impression of the program execution and to allow him to skip over unnecessary details. Since this debugging model can be considered as a precise description of ALF's operational semantics, there is no learning overhead when this debugger is used. Moreover, we believe that the use of this debugging model simplifies the learning of the execution principles of functional-logic languages. These principles are necessarily more complex than the execution of pure functional or pure logic languages. However, the advantages of these principles are convincing: more expressive power than functional languages due to the presence of logic variables [25] and more efficiency than logic languages due to the integration of a deterministic simplification process [13]. In Section 6 we will see how the debugging model can be simplified if a less sophisticated operational semantics is used.

5 Implementation

The debugging model presented in the previous section is implemented as an extended interpreter for ALF programs. The implementation language is also ALF in order to test the ALF system and to demonstrate that ALF can be used for larger applications. The functionality of the current ALF debugger is similar to standard Prolog debuggers. For instance, it allows

- to turn off/on the observability of some ports,
- to set spy points on defined functions,
- to skip over subcomputations inside a box (i.e., to skip from one box port to the next port in this box),

etc. (see [18] for details). In the current implementation the debugger shows the literal or the subterm corresponding to the computation step. Additionally, at the TRY-MATCH port the left-hand side of the applied rule is printed before it is matched against the current subterm in a rewrite step (similarly for the TRY-UNIFY port). Although this information is sufficient in many cases, sometimes the programmer wants to see the entire rule which is currently used. This can be supported by showing the entire rule in rewrite/narrow boxes as in the Coda debugger [24].

Finally, we want to present the current debugging model from a user's point of view by showing some example traces. The first example is a complete trace of the `append` program introduced in Section 2. The initial goal is `append(append([a|V],W),Y)=[b|Z]`. This goal will be disproved due to the rewriting and rejection rule as shown at the end of Section 2. The full trace is lengthy since all rewrite rules for `append` must be applied to the subterms of this goal:

```
?- append(append([a|V],W),Y)=[b|Z].
ENTER-LITERAL: append(append([a|V],W),Y)=[b|Z] ?
ENTER-SIMPLIFICATION: append(append([a|V],W),Y)=[b|Z] ?
```

```
ENTER-REWRITE: append([a|V],W) ?
TRY-MATCH: append([],L)  WITH: append([a|V],W) ?
FAIL-MATCH: append([a|V],W) ?
TRY-MATCH: append([E|R],L)  WITH: append([a|V],W) ?
ENTER-REWRITE: append(V,W) ?
TRY-MATCH: append([],L)  WITH: append(V,W) ?
FAIL-MATCH: append(V,W) ?
TRY-MATCH: append([E|R],L)  WITH: append(V,W) ?
FAIL-MATCH: append(V,W) ?
EXIT-REWRITE: append(V,W) ?
EXIT-REWRITE-BODY: append([a|V],W) ?
EXIT-REWRITE: [a|append(V,W)] ?
ENTER-REWRITE: append([a|append(V,W)],Y) ?
TRY-MATCH: append([],L)  WITH: append([a|append(V,W)],Y) ?
FAIL-MATCH: append([a|append(V,W)],Y) ?
TRY-MATCH: append([E|R],L)  WITH: append([a|append(V,W)],Y) ?
ENTER-REWRITE: append(append(V,W),Y) ?
TRY-MATCH: append([],L)  WITH: append(append(V,W),Y) ?
FAIL-MATCH: append(append(V,W),Y) ?
TRY-MATCH: append([E|R],L)  WITH: append(append(V,W),Y) ?
FAIL-MATCH: append(append(V,W),Y) ?
EXIT-REWRITE: append(append(V,W),Y) ?
EXIT-REWRITE-BODY: append([a|append(V,W)],Y) ?
EXIT-REWRITE: [a|append(append(V,W),Y)] ?
EXIT-SIMPLIFICATION: [a|append(append(V,W),Y)]=[b|Z] ?
ENTER-REJECTION: [a|append(append(V,W),Y)]=[b|Z] ?
REJECTED: [a|append(append(V,W),Y)]=[b|Z] ?
FAIL-LITERAL: [a|append(append(V,W),Y)]=[b|Z] ?
goal failed: append(append([a|V],W),Y)=[b|Z]
```

However, this is the extreme case for our debugging model. Usually, the observability of several ports (like TRY-MATCH) is switched off and the user skips over entire subcomputations which is possible due to the refined box structure of our debugging model. For instance, it is often the case that the user wants to skip the entire simplification process. Then the above trace is reduced as follows (the user command *skip* does not show a subcomputation inside a box and forces the debugger to stop at the next port of the current box):

```
?- append(append([a|V],W),Y)=[b|Z].
ENTER-LITERAL: append(append([a|V],W),Y)=[b|Z] ?
ENTER-SIMPLIFICATION: append(append([a|V],W),Y)=[b|Z] ?  skip
EXIT-SIMPLIFICATION: [a|append(append(V,W),Y)]=[b|Z] ?
ENTER-REJECTION: [a|append(append(V,W),Y)]=[b|Z] ?
REJECTED: [a|append(append(V,W),Y)]=[b|Z] ?
FAIL-LITERAL: [a|append(append(V,W),Y)]=[b|Z] ?
goal failed: append(append([a|V],W),Y)=[b|Z]
```

Another example trace will be shown in the next section.

6 Application of the debugging model

In this section we point out some aspects related to the application of our debugging model.

6.1 Filtering

Due to the increased number of ports in our debugging model, too many details of the computation process are usually presented to the user. Therefore it is necessary to filter the standard output in order to concentrate on the relevant part of the computation process. One possible implementation of filtering is a programmable debugger where the user can configure the debugger to his requests [7]. This could also be implemented on the basis of our debugging model. Another much simpler solution is to turn off the observability of ports in which the user is not interested. Therefore, in a typical configuration of our debugger the observability of the TRY-MATCH, TRY-UNIFY and EXIT-BODY ports in rewrite and narrow boxes is switched off (the user can turn on and off the observability of particular ports during the debugging session). The ports ENTER-REJECTION, NOT-REJECTED, ENTER-REFLECTION and EXIT-REFLECTION are also turned off since these belongs to elementary operations and the user is usually interested in failure situations, i.e., in the ports REJECTED and FAIL-REFLECTION. The following trace shows the computation of the initial goal `append(_,[T])=[a,b]` for such a configuration. The goal is provable if the variable `T` is the last element of the given list at the right-hand side. During this trace the user skips the simplification process of the initial goal and the simplification/narrowing process after the application of the second narrowing rule for `append`:

```
?- append(_,[T])=[a,b].
ENTER-LITERAL: append(_,[T])=[a,b] ?
ENTER-SIMPLIFICATION: append(_,[T])=[a,b] ?  skip
EXIT-SIMPLIFICATION: append(_,[T])=[a,b] ?
ENTER-NARROW: append(_,[T])=[a,b] ?
EXIT-NARROW: [T]=[a,b] ?
FAIL-REFLECTION: [T]=[a,b] ?
REDO-NARROW: append(_,[T])=[a,b] ?
ENTER-SIMP.-NARR.: [E1|append(R1,[T])]=[a,b] ?  skip
EXIT-SIMP.-NARR.: [E1,T]=[a,b] ?
EXIT-NARROW: [E1,T]=[a,b] ?
EXIT-LITERAL: [a,b]=[a,b] ?
goal proved: append([a],[b])=[a,b]
```

The standard trace without filtering consists of 40 steps for the same example. This filtered trace shows that our debugging model can be adjusted to a good reflection of the operational principles of functional logic languages. The experiences with the current implementation of the debugger give us the persuasion that this model is suitable for debugging larger programs and also for understanding the control flow of functional logic programs.

6.2 Debugging other declarative languages

The presented debugging model is adjusted to the operational semantics of ALF which consists of the inference rules rewriting, rejection, innermost basic narrowing, innermost reflection and reflection. These inference rules model a complete and efficient execution mechanism for functional logic programs. If one is interested in similar languages with a more restricted operational semantics, our debugging model can also be applied. But in this case the structure of our model can be simplified as shown in the following.

ALF is a genuine amalgamation of functional and logic languages, i.e., pure logic programming and (first-order) functional programming are contained in ALF. This is also reflected by our debugging model. For instance, a *pure logic ALF program* contains only Boolean functions, has no nested functional expressions, and has only narrowing rules of the form

$$\mathtt{p}_0(\cdots)\mathtt{=true :- }\ \mathtt{p}_1(\cdots)\mathtt{=true,}\ldots\mathtt{,p}_k(\cdots)\mathtt{=true.}$$

Therefore all boxes except the narrow and reflection box can be omitted for such programs (the innermost reflection boxes inside narrow boxes are also superfluous). The result is a restricted debugging model which is very close to the extended debuggers for Prolog [8, 24, 26].

The other extreme is a *pure functional ALF program* which consists of a set of rewrite rules and has no narrowing rules. Moreover, the initial goal is ground, i.e., no logical variables occur during program execution. Consequently, the literal, reflection, narrow, and simplify narrow boxes can be omitted. In this restricted debugging model the user can observe the evaluation of each function call and the matching of a function call with the left-hand sides of the corresponding rules. Therefore it is very similar to symbolic debuggers proposed for functional languages with pattern matching and eager evaluation like Standard-ML [27].

Our debugging model can also be used for other functional logic languages which use some variant of innermost narrowing as their operational semantics. For instance, SLOG [10] executes functional logic programs by innermost narrowing and rewriting. SLOG differs from ALF in the innermost reflection rule which is not included in SLOG since it is assumed that all functions in SLOG are totally defined. Therefore our debugging model can be applied to SLOG with the difference that the innermost reflection boxes inside narrow boxes are deleted. Further simplifications are possible for functional logic languages based on innermost narrowing without simplification like eager BABEL [19, 20]. In this case the simplification, rewrite, rejection, and simplify narrow boxes can also be omitted.

7 Conclusions

We have presented a debugging model for the functional logic language ALF, a language that combines nondeterministic search as in logic languages with deterministic reduction as in functional languages. This debugging model reflects the different computations rules of the operational semantics and allows the user to skip over logically related parts of the execution process. Beyond the possibility of debugging a faulty ALF program, the debugging model can also be used to explain the operational principles of functional logic languages. Note that for pure functional programs where a ground term is reduced to normal form the operational semantics of ALF is identical to the reduction principle of functional languages with pattern matching since narrowing is not applied. Hence our debugging model can also used for functional languages. Moreover, we have shown that our debugging model is general enough to be applied to other functional logic languages with an eager evaluation strategy like SLOG or eager BABEL.

There are several directions for further work. On the one hand the implementation of the debugger must be improved in order to use it for large applications. For this purpose the debugger must be integrated into the A-WAM [12], the abstract machine into which ALF programs are compiled. This can be done similarly to the integration of debuggers in WAM-based Prolog implementations [2]. Another important topic is the extension of the debugging features. For instance, for larger applications it is useful to integrate user-defined pre- and postconditions for functions into the debugging process instead of the simple spy points. Such applications require a more flexible and programmable debugger [7]. Such debuggers are based on the idea to show the user only distinct events of the program execution. Since we have defined the principle events which are observable by the programmer, our debugging model can be seen as a first step to develop advanced symbolic debuggers for functional logic languages.

References

1. P.G. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF: a WAM-based compilation of conditional narrowing. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 318–333. MIT Press, 1989.
2. K.A. Buettner. Fast Decompilation of Compiled Prolog Clauses. In *Proc. Third International Conference on Logic Programming (London)*, pp. 663–670. Springer LNCS 225, 1986.
3. L. Byrd. Understanding the Control Flow of Prolog Programs. In *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.
4. M.M.T. Chakravarty and H.C.R. Lock. The Implementation of Lazy Narrowing. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 123–134. Springer LNCS 528, 1991.
5. D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

6. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
7. M. Ducassé. A general trace query mechanism based on Prolog. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 400–414. Springer LNCS 631, 1992.
8. M. Eisenstadt. A Powerful Prolog Trace Package. In *Advances in Artificial Intelligence*, pp. 149–158. Elsevier Science Publishers, 1985.
9. M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.
10. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
11. M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.
12. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.
13. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.
14. M. Hanus. Incremental Rewriting in Narrowing Derivations. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 228–243. Springer LNCS 632, 1992.
15. M. Hanus. On the Completeness of Residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pp. 192–206. MIT Press, 1992.
16. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.
17. J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.
18. B. Josephs. The development of a debugger for the functional logic language ALF (in German). Diploma thesis, Univ. Dortmund, 1992.
19. H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language. In *Proc. ESOP 90*, pp. 271–290. Springer LNCS 432, 1990.
20. R. Loogen. From Reduction Machines to Narrowing Machines. In *Proc. of the TAPSOFT '91*, pp. 438–457. Springer LNCS 494, 1991.
21. A. Mück. Compilation of Narrowing. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 16–29. Springer LNCS 456, 1990.
22. W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.
23. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
24. D. Plummer. Coda: An Extended Debugger for PROLOG. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 496–511. MIT Press, 1988.
25. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.

26. A. Schleiermacher and J.F.H. Winkler. The Implementation of ProTest a Prolog-Debugger for a Refined Box Model. *Software - Practice & Experience*, Vol. 20, No. 10, pp. 985–1006, 1990.
27. A.P. Tolmach and A.W. Appel. Debugging Standard ML Without Reverse Engineering. In *Proc. ACM Lisp and Functional Programming Conference '90*, pp. 1–12, Nice, 1990.
28. D. Wolz. Design of a Compiler for Lazy Pattern Driven Narrowing. In *Recent Trends in Data Type Specification*, pp. 362–379. Springer LNCS 534, 1990.