Verification of Compiler Correctness for the WAM

Cornelia Pusch*

Fakultät für Informatik, Technische Universität München
80290 München, Germany

E-mail: pusch@informatik.tu-muenchen.de

Abstract. Relying on a derivation of the Warren Abstract Machine (WAM) by stepwise refinement of Prolog models by Börger and Rosenzweig we present a formalization of an operational semantics for Prolog. Then we develop four refinement steps towards the Warren Abstract Machine (WAM). The correctness and completeness proofs for each step have been elaborated with the theorem prover Isabelle using the logic HOL.

1 Introduction

In the area of logic programming, Prolog ranks among the most prominent programming languages. The development of efficient compilation techniques allows the application of logic programming even in large-scale software development. One of the main contributions to this field is due to D. Warren, having developed a sophisticated compilation concept known as the Warren Abstract Machine (WAM), which serves as basis for a large number of prolog implementations.

While Prolog benefits from a well-defined semantics derived from its logical roots, the development of the WAM is not based on theoretical investigations, correctness is in general "proved" by successful testing. For this reason, several approaches have been made to develop a formal verification for the WAM [Rus92], [BR94].

While in [Rus92], correctness is shown for a specific Prolog compiler translating Prolog programs into WAM code, [BR94] provides a correctness proof for a whole class of compilers by formulating general compiler assumptions. The specification of an operational semantics for Prolog is given in terms of *evolving algebras*, and a development of the WAM is given by stepwise refinement, outlining the proofs for correctness and completeness of each refinement step. However, the proofs are not complete, their description remains semi-formal.

A first attempt to check the proofs by machine was made with the theorem proving system KIV [Sch95]. This case study revealed that the formal proofs were significantly more involved than estimated by [BR94].

* Research supported by DFG grant Br 887/4-3, Deduktive Programmentwicklung

In our paper we present the formalization of the correctness proofs in the Isabelle system [Pau94]. In contrast to the KIV formalization we do not use the framework of evolving algebras. Our development starts from a slightly different operational semantics for Prolog [DM88]. The operational semantics and all the refinement steps towards the WAM considered so far are formalized in higher order logic. The reasons for refraining from embedding the formalism of evolving algebras are discussed in section 4.

The rest of this paper is structured as follows. Section 2 provides a short introduction to Isabelle. In section 3 formalizations of the syntax and operational semantics of Prolog are given. In section 4 the refinement steps towards the WAM are elaborated. In section 5 the proof principles are discussed, and section 6 summarizes the results of the case study and outlines future work.

2 Isabelle

Isabelle is a generic theorem prover, where new logics are introduced by specifying their syntax and rules of inference. Proof procedures can be expressed using tactics and tacticals. A detailed introduction to the Isabelle system can be found in [Pau94].

The formalization and proofs described in this paper are based on the instantiation of Isabelle for higher order logic, called Isabelle/HOL.

The new release of Isabelle/HOL comes along with a graphical interface, allowing the use of mathematical symbols like $\forall$ and $\exists$. Therefore, the presentation of the formalization in this paper corresponds to the Isabelle input (except the introduction of some abbreviations for better readability).

3 Prolog Syntax and Semantics

This section describes the syntactic categories of Prolog programs and their formalization in Isabelle/HOL. Then we give an operational semantics by means of inference rules. More detailed information about logic programming can be found in [Apt90] and [Llo87].

3.1 Syntax

Since we do not have to reason about the exact structure of terms and formulae, we start with the notions of predicates and atoms. Furthermore we do not deal with the explicit construction of atoms by a predicate symbol followed by a list of terms. We just assume the existence of types for predicates and atoms together with a function returning the predicate symbol of an atom. Therefore, this part of our formalization is not definitional:

```
types  Pred
       Atom
consts pname :: Atom ⇒ Pred
```

In logic programming syntax, a positive literal is an atom, a negative literal is the negation of an atom, and a program clause is a set of literals, containing exactly one positive literal, which is called the head. The remaining negative literals are called the body of the clause. Finally, a logic program is a set of program clauses.

However, these definitions are not suitable for the discussion of Prolog implementations. In order to describe the computational behavior of Prolog programs, one usually considers specific depth-first search strategies (SLD-resolution) and the use of the cut symbol, which is an impure but central control facility of Prolog. Therefore, we have to redefine the notions of literals and program clauses as follows[2]:

$$\begin{aligned} \mathsf{datatype\ Lit} &= \mathsf{Cut} \\ &\mid \mathsf{Atm\ Atom} \\ \mathsf{types\ Clause} &= \mathsf{Atom} \times (\mathsf{Lit\ list}) \end{aligned}$$

In our terminologies a literal is either the cut symbol or an atom. This covers the notion of a negative literal and introduces the cut symbol. A program clause is a pair consisting of an atom (the head) and a list of literals (the body). This definition ensures that the cut never occurs as the head of a clause. A logic program is again a list of program clauses and a goal is a list of literals:

$$\begin{aligned} \mathsf{types\ Program} &= \mathsf{Clause\ list} \\ \mathsf{Goal} &= \mathsf{Lit\ list} \end{aligned}$$

In the following we introduce the concepts of substitution, unification and renaming. Since we abstracted away from terms and the construction of atoms we cannot give complete definitions for these functions. However, this is not a severe drawback as these concepts are well understood. Therefore we just axiomatize some minimal properties essential to the further proofs.

Substitutions are represented by the type Subst coming along with the functions

$$\begin{aligned} \mathsf{consts\ @subcomp} &:: \mathsf{Subst} \Rightarrow \mathsf{Subst} \Rightarrow \mathsf{Subst} \quad (\text{"\_ } \circ \text{ \_"}) \\ \mathsf{@subapp} &:: \mathsf{Subst} \Rightarrow \mathsf{Atom} \Rightarrow \mathsf{Atom} \quad (\text{"\$"}) \end{aligned}$$

for composition and substitution application. The identifiers beginning with a @ declare the names to be used just for the internal representation in Isabelle. The user has to apply the names given in parentheses, where $\circ$ is introduced as infix operator. The function

$$\mathsf{consts\ mgu} :: \mathsf{Atom} \Rightarrow \mathsf{Atom} \Rightarrow \mathsf{Result}$$

returns the most general unifier of two atoms, if there is one, and a fail value otherwise. This optional result value is modeled by defining an error monad:

$$\begin{aligned} \mathsf{datatype\ 'a\ maybe} &= \mathsf{Ok\ 'a} \\ &\mid \mathsf{Fail} \\ \mathsf{types\ Result} &= \mathsf{Subst\ maybe} \end{aligned}$$

[2] The datatype construct generates axioms for free data types: injectiveness, distinctness and an induction rule. The types construct is used here to introduce type synonyms.

When selecting a clause for unification, all variables occuring in that clause should be renamed, such that the so called variant does not have a variable name in common with the original goal and the set of clauses already used in the derivation process. The relevance of this renaming is discussed for example in [Apt90]. As clauses are built up by an atom and a list of literals, it is convenient to define an overloaded renaming function working on atoms and lists of literals. Overloading of function symbols can be realized in Isabelle by introducing a new type class, which is a subclass of the predefined class term of higher order terms:

```
classes renamecl < term
```

We then make Atom and Lit elements of renamecl. Furthermore we have to ensure that application of the type constructor list to an element of class renamecl yields an element of the same class:

```
arities Atom :: renamecl
        Lit  :: renamecl
        list :: (renamecl)renamecl
```

The intention behind the renaming function is that it takes a value and a renaming index returning a value which is equal to the input up to the variable names. By that each instance of a variable is made unique. As we will see later the renaming index is increased by the interpreter function after each successfull unification:

```
types  Rename  = nat
consts @rename :: ('a :: renamecl) ⇒ Rename ⇒' a :: renamecl      (" ↑ ")
```

For the proofs carried out so far, we only needed the following basic properties of the functions for substitution, unification and renaming:

```
rules pname a1 ≠ pname a2 ⟹ mgu a1 a2 = Fail
      pname (↑ a vi) = pname a
      pname ($ sub a) = pname a
```

The first axiom states that unification fails for two atoms with different predicate symbols. The next two axioms express that renaming and substitution application do not affect the predicate symbol of an atom.

3.2 Operational Semantics

The semantics of Prolog programs is usually given in terms of the model theory of first order logic. Since this approach ignores the behavioral aspects of Prolog like sequential depth-first search and the cut control facility, S. Debray and P. Mishra developed a denotational as well as an equivalent operational semantics expressing the properties of interest [DM88].

Hereafter we will give an operational semantics by the definition of an interpreter, which is almost identical to the one described in [DM88] with just some

slight modifications. In our approach we chose a different renaming function, according to the formalization of Börger and Rosenzweig [BR94]. Furthermore we considered just one single possible answer substitution (an extension to multiple answer substitutions is under construction). Another difference consists in the formalization of the interpreter function. In [DM88], the interpreter is defined by a set of recursive equations. Because computation does not necessarily terminate in Prolog programming, we have to deal with partial functions. Since in Isabelle/HOL functions are total, we have to model partiality by inductive relations.

The computation state of the interpreter is described by so called configurations. A configuration consists of a list of clauses describing the Prolog program, a computation stack storing different backtrack points, and a renaming index. Therefore, we define:

$$\mathsf{types}\ \mathsf{Config} = \mathsf{Program} \times \mathsf{CStack} \times \mathsf{Rename}$$

Each element of the computation stack consists of the substitution computed so far, the goal still to be executed, and a list of candidate clauses that are yet to be tried in solving the leftmost literal of the corresponding goal.

To model the effect of cuts in Prolog the goal is not presented linearly but decomposed in a list of "decorated" subgoals: each subgoal has to maintain its own continuation information, which is just a part of the entire computation stack. If a cut is encountered while processing a subgoal, the tail of the current computation stack is replaced by the continuation stack stored along with the subgoal. This corresponds to the deletion of all those backtrack points set up by literals on the left of the cut as well as the backtrack point for the parent goal (i.e. the goal which caused the clause containing the cut to be activated), which is the usual Prolog semantics for cut. At the end of this section we will see an example for this. Now we define the computation stack as:

$$\begin{array}{l}\mathsf{datatype}\ \mathsf{CStack} = \mathsf{Es} \\ \qquad\qquad \mid\ \text{"\#\#"}\ (\mathsf{Subst} \times (\mathsf{Goal} \times \mathsf{CStack})\mathsf{list} \times \mathsf{Clause\ list})\ \ \mathsf{CStack} \\ \qquad\qquad\qquad (\mathsf{infixr}\,70)\end{array}$$

Now, we define an interpreter relation for Prolog programs:

$$\begin{array}{lll}\mathsf{consts} & \mathsf{interp0} & :: (\mathsf{Config} \times \mathsf{Result})\mathsf{set} \\ \mathsf{syntax} & \mathsf{@interp0} & :: \mathsf{Config} \Rightarrow \mathsf{Result} \Rightarrow \mathsf{bool} \qquad (\text{"\_} \xrightarrow{i_0} \text{\_"}[0, 95]95) \\ \mathsf{translations} & \mathsf{config} \xrightarrow{i_0} \mathsf{res} & == (\mathsf{config}, \mathsf{res}) \in \mathsf{interp0}\end{array}$$

The interpreter relation interp0 is defined as a set of pairs. The syntax section introduces an infix operator $\xrightarrow{i_0}$ for this relation, for which a translation into the set representation is given. Note that the i_0 denotes the base interpreter. We will refer to the interpreter after n refinement steps as i_n.

We give an inductive definition of the interpreter $\xrightarrow{i_0}$ by means of inference rules, where multiple premises are stacked on top of each. Note that the predefined type of lists comes along with [] for the empty list and the infix operator

for list construction:

If the computation stack is empty, execution terminates returning a fail value:

$$\text{query\_failed} \qquad \frac{}{(\mathsf{db}, \mathsf{Es}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{Fail}}$$

If the current list of decorated subgoals is empty, execution terminates returning the current substitution:

$$\text{query\_success} \qquad \frac{}{(\mathsf{db}, (\mathsf{sub}, [], \mathsf{cll})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{Ok\ sub}}$$

If we are interested in all possible answer substitutions, execution has to be continued by processing the tail sl of the stack.

If the first element of the current decorated subgoals list is empty, execution continues by processing the rest of the decorated subgoals list:

$$\text{goal\_success} \qquad \frac{(\mathsf{db}, (\mathsf{sub}, \mathsf{ds}, \mathsf{db})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}{(\mathsf{db}, (\mathsf{sub}, ([], \mathsf{ctp})\#\mathsf{ds}, \mathsf{cll})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}$$

If the first subgoal of the current decorated subgoals list begins with a cut, the tail of the computation stack is replaced by the continuation ctp of the current subgoal:

$$\text{cut} \qquad \frac{(\mathsf{db}, (\mathsf{sub}, (\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{db})\#\#\mathsf{ctp}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}{(\mathsf{db}, (\mathsf{sub}, (\mathsf{Cut}\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{cll})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}$$

All remaining rules hold for configurations, where the current decorated subgoals list is not empty and the first subgoal does not start with a cut but with an atom.

If the current choice point contains no more candidate clauses, execution proceeds by backtracking to the most recent choice point, i.e. popping the current one from the computation stack:

$$\text{back} \qquad \frac{(\mathsf{db}, \mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}{(\mathsf{db}, (\mathsf{sub}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, [])\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}$$

If the list of clauses still to be tried contains at least one element, two different cases have to be considered:

If unification of the leftmost literal in the current subgoal with the head of the first candidate clause fails, the next candidate clause has to be tried:

$$\text{atm1} \qquad \frac{\begin{array}{c}\mathsf{mgu}(\$\ \mathsf{sub\ x})(\uparrow\ \mathsf{h\ vi}) = \mathsf{Fail} \\ (\mathsf{db}, (\mathsf{sub}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{cs})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}\end{array}}{(\mathsf{db}, (\mathsf{sub}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}$$

If unification succeeds with a substitution sub', execution proceeds by extending the computation stack with a new choice point chp and incrementing the renaming index:

$$\text{atm2}\quad \frac{\mathsf{mgu}(\$\ \mathsf{sub}\ \mathsf{x})(\uparrow\ \mathsf{h}\ \mathsf{vi}) = \mathsf{Ok}\ \mathsf{sub}' \qquad (\mathsf{db}, \mathsf{chp}\#\#(\mathsf{sub}, ((\mathsf{Atm}\ \mathsf{x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{cs})\#\#\mathsf{sl}, \mathsf{vi}{+}1) \xrightarrow{i_0} \mathsf{res}}{(\mathsf{db}, (\mathsf{sub}, ((\mathsf{Atm}\ \mathsf{x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs})\#\#\mathsf{sl}, \mathsf{vi}) \xrightarrow{i_0} \mathsf{res}}$$

$$\text{where } \mathsf{chp} = (\mathsf{sub}' \circ \mathsf{sub}, (\uparrow\ \mathsf{b}\ \mathsf{vi}, \mathsf{sl})\#(\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{db})$$

The new choice point chp contains an updated substitution, a subgoal list where the unified atom is replaced by a new subgoal containing the body of the selected clause, and the whole program serving as candidate clauses for the new subgoal. The decoration of the new subgoal has to be set to the tail of the old computation stack. In the old choice point, the selected clause has to be removed from the list of untried clauses.

The formalization of inductive sets is supported in Isabelle/HOL by a special package, where all generated rules are automatically proved as theorems.

Example 1 In the following we give a little example, to see how the interpreter works on the computation stack. The most interesting point is to see how the list of decorated subgoals evolves. Therefore we omit in the representation the substitutions and candidate clauses:
Consider the Prolog program

o :− p, x.	q :− s.
p :− q, !, r.	q .
p .	x .

and query o. Computation starts with an initial computation stack containing the query decorated with an empty stack:

$$\boxed{[\langle[\mathsf{o}], \mathsf{Es}\rangle]}$$

After some execution steps the computation stack looks as follows:

$$\begin{array}{|c|}\hline [\langle[\mathsf{s}], c_2\rangle, \langle[!, \mathsf{r}], c_1\rangle, \langle[\mathsf{x}], \mathsf{Es}\rangle, \langle[], \mathsf{Es}\rangle] \\ \hline [\langle[\mathsf{q}, !, \mathsf{r}], c_1\rangle, \langle[\mathsf{x}], \mathsf{Es}\rangle, \langle[], \mathsf{Es}\rangle] \\ \hline [\langle[\mathsf{p}, \mathsf{x}], \mathsf{Es}\rangle, \langle[], \mathsf{Es}\rangle] \\ \hline [\langle[\mathsf{o}], \mathsf{Es}\rangle] \\ \hline \end{array}$$

$$\text{where } c_1 = \boxed{[\langle[\mathsf{o}], \mathsf{Es}\rangle]} \text{ and } c_2 = \begin{array}{|c|}\hline [\langle[\mathsf{p}, \mathsf{x}], \mathsf{Es}\rangle, \langle[], \mathsf{Es}\rangle] \\ \hline [\langle[\mathsf{o}], \mathsf{Es}\rangle] \\ \hline \end{array}$$

Since unification with s fails for all program clauses, the top element of the computation stack is popped. The next clause to be tried for q succeeds immediately, then we get:

$$\begin{array}{|c|}\hline [\langle[!, \mathsf{r}], c_1\rangle, \langle[\mathsf{x}], \ \mathsf{Es}\rangle, \langle[], \mathsf{Es}\rangle] \\ \hline [\langle[\mathsf{p}, \mathsf{x}], \mathsf{Es}\rangle, \langle[], \mathsf{Es}\rangle] \\ \hline [\langle[\mathsf{o}], \mathsf{Es}\rangle] \\ \hline \end{array}$$

Now a cut is encountered in the current subgoal. As described, the tail of the computation stack is replaced by c_1, which yields:

$[\langle [r], c_1 \rangle, \langle [x], \ Es \rangle, \langle [], Es \rangle]$
$[\langle [o], Es \rangle]$

Unification with r fails for all program clauses, therefore backtracking has to be executed. We see now that the remaining clause for p is no more considered, according to the meaning of the cut.

4 Towards the WAM

The first Prolog compiler was developed at the University of Edinburgh by D.H. Warren in 1977. The Warren Abstract Machine (WAM) is a refinement of this system. Roughly speaking, the WAM is an abstract machine consisting of a memory architecture and an instruction set tailored to Prolog. It is based on the concept of a virtual machine in order to achieve portability to a wide range of hardware configurations [Boi93]. In this paper, we do not describe the details of the WAM, since we are just doing some refinement steps towards the WAM, starting from our operational semantics presented in the previous section. For more information the reader might refer to [AK91], which gives a more detailed introduction to the WAM, rather than the original paper [War83].

In [BR94] Börger and Rosenzweig developed a methodical derivation of the WAM starting from an operational semantics for Prolog. They provide a correctness proof for a whole class of compilers by formulating general compiler assumptions. The specification of an operational semantics for Prolog is given in terms of *evolving algebras* [Gur95]. Their development of the WAM is partitioned into 12 refinement steps, each of which introduces a new aspect of the WAM. For each step, they outline the proofs for correctness and completeness. However, these proofs are not complete, their description remains semi-formal.

A first attempt to check the proofs by machine was made with the theorem proving system KIV [Sch95]. This case study revealed that the formal proofs were significantly more involved than estimated by [BR94]. For example, the correctness proof of the first refinement step used an invariant property which covered an entire page. Studying this proof, we got the impression that this complexity is caused to a large extent by the formalism of evolving algebras. For instance, the manipulation of inductive data structures seems to be quite tedious. However, this concept turned out to be central to the formalization of this case study. On the other hand higher order logic offers advanced features for the treatment of inductive data structures. Therefore, we coded the operational semantics directly in HOL as presented above and refrained from embedding the formalism of evolving algebras.

Nevertheless, we could adopt the structure of the refinement steps developed by Börger and Rosenzweig in [BR94] which turned out to be very suitable for the realization of the proof task.
We now outline the steps of our development and present the formal definition of the refined interpreter after the fourth step.

4.1 Introduction of pointers

Copying parts of the computation stack into the decorated subgoals list is very inefficient. Therefore, the first improvement consists of replacing the copies by pointers to the original stack, called cut points. Hence, we define:

$$\begin{aligned} \mathsf{types\ Index} &= \mathsf{nat} \\ \mathsf{CArray} &= (\mathsf{Subst} \times (\mathsf{Goal} \times \mathsf{Index})\mathsf{list} \times \mathsf{Clause\ list})\mathsf{list} \end{aligned}$$

The type CArray replaces CStack in the configuration. Since the new stack definition no longer contains nested recursion, the definition of CArray can be based on the predefined type of lists. We do not need to introduce a new data type. To allow the deletion of choice points on the stack up to a given index, we provide a function

$$\mathsf{consts\ ntail} :: \mathsf{Index} \Rightarrow \mathsf{CArray} \Rightarrow \mathsf{CArray}$$

which takes an index i and a computation stack, and returns the back end of the stack of length i.

4.2 Optimizing the list of candidate clauses

Up to now, the list of candidate clauses for a new subgoal consists of the entire program. However, it is clear that only some of the clauses are likely to match the selected atom. Therefore, in a second step we restrict the set of candidate clauses by a preselection depending on the currently selected atom. This is done by a function

$$\mathsf{consts\ pdef} :: \mathsf{Atom} \Rightarrow \mathsf{Program} \Rightarrow \mathsf{Clause\ list}$$

which filters out those clauses from a given program whose heads consist of the same predicate symbol as the currently selected atom. Additionally, the configuration is extended by a new component which describes a triple of registers holding the current values for substitution, decorated subgoals list and candidate clause list. The computation stack is left to maintain the remaining backtracking points.

4.3 Reusing choice points

During execution, it is often the case that information is popped from the stack into the registers, and in a later stage, almost identical information is pushed back onto the stack. This information transfer can be optimized by leaving the formerly popped choice point on the stack and just changing part of its contents. This is related to the well-known peephole optimization in compiler construction.

4.4 Deleting useless choice points

The next optimization step consists of deleting trivial choice points. This means a choice points including an empty candidate clause list is no longer pushed onto the stack: whenever execution returned to this point, it would be immediately popped by backtracking.

4.5 The optimized interpreter model

After these four refinement steps, the formalization of the interpreter has undergone the following changes:

The configuration of our interpreter has been extended by two components. The first one describes different modes of the computation. We distinguish four modes: In call mode execution proceeds until an atom is encountered in the current subgoal or computation terminates. In try mode a choice point is pushed onto the computation stack. In enter mode unification is attempted, and in retry mode reuse of choice points is done. The second extension is an index to the computation stack, which stores the value of the current cut point. This cut point register will be needed for the further development. We therefore define:

$$\begin{array}{lll} \mathsf{datatype}\ \mathsf{Mode} & = \mathsf{Call} \mid \mathsf{Try} \mid \mathsf{Enter} \mid \mathsf{Retry} \\ \mathsf{types} & \mathsf{Regs} & = (\mathsf{Subst} \times (\mathsf{Goal} \times \mathsf{Index})\mathsf{list} \times \mathsf{Clause\ list}) \\ & \mathsf{Config} & = \mathsf{Program} \times \mathsf{CArray} \times \mathsf{Regs} \times \mathsf{Rename} \times \mathsf{Mode} \times \mathsf{Index} \end{array}$$

The inductive definition of the interpreter $\xrightarrow{i_4}$ is as follows:

If the decorated goals register is empty, the query was successful, returning the content of the substitution register as result:

$$\mathsf{query\_success} \quad \frac{}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, [], \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{Ok\ sreg}}$$

If the first subgoal in the decorated subgoals register is empty, execution proceeds the rest of the decorated goals list:

$$\mathsf{goal\_succes} \quad \frac{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, \mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, ([], \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If a cut is encountered, the backtracking stack is shortened upto the cut point of the current subgoal:

$$\mathsf{cut} \quad \frac{(\mathsf{db}, \mathsf{ntail\ ctp\ arr}, (\mathsf{sreg}, (\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, (\mathsf{Cut}\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

The following two rules hold for configurations, where the current subgoal begins with an atom, but the predicate of the current atom is not defined. This is the case, if the current atom does not occur in any head of a program clause.

If the backtracking stack is empty, computation fails:

$$\mathsf{call1} \quad \frac{\mathsf{pdef\ x\ db} = []}{(\mathsf{db}, [], (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{Fail}}$$

If the backtracking stack is not empty, execution is processed in Retry mode:

$$\mathsf{call2} \quad \frac{\mathsf{pdef\ x\ db} = [] \qquad (\mathsf{db}, \mathsf{x}\#\mathsf{xs}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Retry}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, \mathsf{x}\#\mathsf{xs}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If the definition of the current atom contains at least one clause, computation continues with mode set to Try and the candidate clauses and cut point registers updated:

$$\text{call3}\quad \frac{\begin{array}{c}\mathsf{pdef\ x\ db} = \mathsf{c\#cs} \\ (\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{c\#cs}), \mathsf{vi}, \mathsf{Try}, \mathsf{length\ arr}) \xrightarrow{i_4} \mathsf{res}\end{array}}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, \mathsf{creg}), \mathsf{vi}, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If computation is in Try mode, two different cases have to be distinguished.

In the first case, the candidate clauses register contains at least two clauses, one to be tried immediately and at least one to be pushed onto the stack. Then execution proceeds in Enter mode with a new choice point pushed onto the stack:

$$\text{try1}\quad \frac{(\mathsf{db}, (\mathsf{sreg}, \mathsf{dreg}, \mathsf{c2\#cs})\#\mathsf{arr}, (\mathsf{sreg}, \mathsf{dreg}, \mathsf{c1\#c2\#cs}), \mathsf{vi}, \mathsf{Enter}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, \mathsf{dreg}, \mathsf{c1\#c2\#cs}), \mathsf{vi}, \mathsf{Try}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If there is only one candidate clause to be tried, no additional choice point has to be stored on the stack:

$$\text{try2}\quad \frac{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, \mathsf{dreg}, [\mathsf{c}]), \mathsf{vi}, \mathsf{Enter}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, \mathsf{dreg}, [\mathsf{c}]), \mathsf{vi}, \mathsf{Try}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If unification fails in Enter mode, the result of the computation depends on the contents of the backtracking stack.

If there are no more backtracking points, computation terminates returning a fail value:

$$\text{enter1}\quad \frac{\mathsf{mgu}(\$\ \mathsf{sreg\ x})(\uparrow\ \mathsf{h\ vi}) = \mathsf{Fail}}{(\mathsf{db}, [], (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs}), \mathsf{vi}, \mathsf{Enter}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{Fail}}$$

If the backtracking contains at least one element, computation is continued in Retry mode :

$$\text{enter2}\quad \frac{\begin{array}{c}\mathsf{mgu}(\$\ \mathsf{sreg\ x})(\uparrow\ \mathsf{h\ vi}) = \mathsf{Fail} \\ (\mathsf{db}, \mathsf{x\#xs}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs}), \mathsf{vi}, \mathsf{Retry}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}\end{array}}{(\mathsf{db}, \mathsf{x\#xs}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs}), \mathsf{vi}, \mathsf{Enter}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If unification succeeds, execution proceeds in Call mode after updating the registers:

$$\text{enter3}\quad \frac{\begin{array}{c}\mathsf{mgu}(\$\ \mathsf{sreg\ x})(\uparrow\ \mathsf{h\ vi}) = \mathsf{Ok\ sub}' \\ (\mathsf{db}, \mathsf{arr}, \mathsf{regs}, \mathsf{vi}+1, \mathsf{Call}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}\end{array}}{(\mathsf{db}, \mathsf{arr}, (\mathsf{sreg}, ((\mathsf{Atm\ x})\#\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs}), \mathsf{vi}, \mathsf{Enter}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

$$\text{where } \mathsf{regs} = (\mathsf{sub}' \circ \mathsf{sreg}, (\uparrow\ \mathsf{b\ vi}, \mathsf{sl})\#(\mathsf{ls}, \mathsf{ctp})\#\mathsf{ds}, (\mathsf{h}, \mathsf{b})\#\mathsf{cs})$$

In Retry mode, the information of the top level backtracking point is reused, where two different cases have to be considered:

In the first case, the top level element contains more than one candidate clauses. Then the backtrack information is loaded into the registers while the candidate clauses list is updated in the top level stack element:

$$\text{retry1}\quad \frac{(\mathsf{db}, (\mathsf{sub}, \mathsf{dcl}, \mathsf{c1}\#\mathsf{cs})\#\mathsf{xs}, (\mathsf{sub}, \mathsf{dcl}, \mathsf{c}\#\mathsf{c1}\#\mathsf{cs}), \mathsf{vi}, \mathsf{Enter}, \mathsf{length\ xs}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, (\mathsf{sub}, \mathsf{dcl}, \mathsf{c}\#\mathsf{c1}\#\mathsf{cs})\#\mathsf{xs}, (\mathsf{sreg}, \mathsf{dreg}, \mathsf{creg}), \mathsf{vi}, \mathsf{Retry}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

If the backtracking point contains just one single clause still to be tried, the backtrack information is loaded into the registers and the current top level stack element is deleted:

$$\text{retry2}\quad \frac{(\mathsf{db}, \mathsf{xs}, (\mathsf{sub}, \mathsf{dcl}, [\mathsf{c}]), \mathsf{vi}, \mathsf{Enter}, \mathsf{length\ xs}) \xrightarrow{i_4} \mathsf{res}}{(\mathsf{db}, (\mathsf{sub}, \mathsf{dcl}, [\mathsf{c}])\#\mathsf{xs}, (\mathsf{sreg}, \mathsf{dreg}, \mathsf{creg}), \mathsf{vi}, \mathsf{Retry}, \mathsf{ct}) \xrightarrow{i_4} \mathsf{res}}$$

Example 2 Now we will see how computation has changed in our example: In addition to the computation stack there is now a register containing the current decorated subgoal list. In the initial state, the query is stored in the register and the stack is empty. After some execution steps these components look as follows:

$$[\langle[\mathsf{s}], 1\rangle, \langle[!, \mathsf{r}], 0\rangle, \langle[\mathsf{x}], 0\rangle, \langle[], 0\rangle] \qquad \begin{array}{|c|}\hline [\langle[\mathsf{q}, !, \mathsf{r}], 0\rangle, \langle[\mathsf{x}], 0\rangle, \langle[], 0\rangle] \\ \hline [\langle[\mathsf{p}, \mathsf{x}], 0\rangle, \langle[], 0\rangle] \\ \hline \end{array}$$

You may notice that the bottom stack element of the example in 3.2 does no longer occur in this computation. This results of the fact that there exists just one single program clause for o. After having tried it, it would be useless to return to this point since the list of candidate clauses would be empty.

Since there is no program clause for s the information of the top level backtracking point is reused. There is just one clause still to be tried, therefore the backtracking point is popped from the stack into the register:

$$[\langle[!, \mathsf{r}], 0\rangle, \langle[\mathsf{x}], 0\rangle, \langle[], 0\rangle] \qquad [\langle[\mathsf{p}, \mathsf{x}], 0\rangle, \langle[], 0\rangle]$$

Now a cut is encountered in the current subgoal which causes the backtracking stack to be set to the empty stack:

$$[\langle[\mathsf{r}], 0\rangle, \langle[\mathsf{x}], 0\rangle, \langle[], 0\rangle] \qquad []$$

Since there is no program clause for r, computation terminates returning Fail.

5 Proof principles

In a refinement step, a more concrete interpreter model is developed from an abstract model. To establish a relationship between two different levels, we have to define an abstraction function F, translating configurations of the concrete interpreter to configurations of the abstract one.

We call an interpreter $\xrightarrow{i_1}$ a correct refinement of the interpreter $\xrightarrow{i_0}$, if every computation of $\xrightarrow{i_1}$ starting with an initial configuration terminates returning a result res provided the computation of $\xrightarrow{i_0}$ returns res starting with an equivalent initial configuration. The notion of initial configuration is explained below.

A configuration of $\xrightarrow{i_0}$ is a triple consisting of the Prolog program, a computation stack, and a renaming index. In an initial configuration, the computation stack contains exactly one choicepoint, consisting of a substitution which is typically set to the identity map, a decorated subgoals list containing the goal to be solved decorated by the empty stack, and a list of candidate clauses which is typically set to the whole program. The initial configuration for $\xrightarrow{i_1}$ just differs in the decoration of the goal, where now a pointer to the empty stack is held. Application of the abstraction function F to the initial configuration of $\xrightarrow{i_1}$ returns the equivalent initial configuration of $\xrightarrow{i_0}$. The correctness theorem is then formalized as follows:

$$\text{correctness} \qquad \frac{((\mathsf{db}, [(\mathsf{subst}, [(\mathsf{goal}, 0)], \mathsf{cll})], 0) \xrightarrow{i_1} \mathsf{res})}{(\mathsf{F}(\mathsf{db}, [(\mathsf{subst}, [(\mathsf{goal}, 0)], \mathsf{cll})], 0) \xrightarrow{i_0} \mathsf{res})}$$

Since this assertion cannot be proved directly, we have to show the validity of a more general theorem, holding for any given configuration. The following theorem can be proved by rule induction:

$$\text{i1\_implies\_i0} \qquad \frac{\mathsf{config\_ok\ config} \quad \mathsf{config} \xrightarrow{i_1} \mathsf{res}}{(\mathsf{F\ config}) \xrightarrow{i_0} \mathsf{res}}$$

Here, we had to introduce an additional assumption. The predicate config_ok restricts config to admissible configurations. One of the central proof tasks is to find the right restrictions. For each refinement step, several attempts were necessary to find the final solution.

Proving correctness is not sufficient to assure a really useful implementation. We could implement $\xrightarrow{i_1}$ by a never-halting function fulfilling the correctness property. Therefore, we have to verify the completeness of the development step as well, which assures that every solution computed by $\xrightarrow{i_0}$ can be found by $\xrightarrow{i_1}$:

$$\text{completeness} \qquad \frac{(\mathsf{F}(\mathsf{db}, [(\mathsf{subst}, [(\mathsf{goal}, 0)], \mathsf{cll})], 0) \xrightarrow{i_0} \mathsf{res})}{((\mathsf{db}, [(\mathsf{subst}, [(\mathsf{goal}, 0)], \mathsf{cll})], 0) \xrightarrow{i_1} \mathsf{res})}$$

Here again, a generalization of the theorem has to be proved:

$$\text{i0\_implies\_i1} \qquad \frac{\mathsf{config\_ok\ config'} \quad \mathsf{F config'} \xrightarrow{i_0} \mathsf{res}}{\mathsf{config'} \xrightarrow{i_1} \mathsf{res}}$$

This technique of defining an abstraction function F and inductively proving correctness and completeness by finding suitable restrictions was common to all refinement steps considered so far.

6 Results and Future Work

The formalization and implementation of the proofs for four development steps took seven months in total. The formalization in Isabelle comprises about 900 lines, the proofs for correctness and completeness consist of approximately 3500 user interactions. Although Isabelle offers a certain degree of automation, significant parts of the proofs have to be guided by user interaction. Better proof support by the system would facilitate the realization of complex case studies like the present one. This concerns in particular an improvement of error messages returned by the system.

As described, we decided to refrain from embedding the formalism of evolving algebras and coded the different refinement steps of an Prolog interpreter directly in higher order logic. Because of this, we were able to make intensive use of Isabelle's features concerning the treatment of inductive data structures and recursive concepts. The type class mechanism was profitably used for overloading. It is our opinion that the adaption of the formalization to higher order logic simplified the complexity of the proof invariants to a large extent. Due to that, we were able to conduct a large-scale case study like the present one: as far as we know this is one of the biggest mechanized proofs concerning operational semantics. In general this cannot be realized without careful decomposition of the proof task. Here the adaption of the refinement steps developed by Börger and Rosenzweig was essential to reduce the complexity of each step to a manageable size.

Our next steps consist in extending our formalization to the computation of multiple answer substitutions, which corresponds closer to a real Prolog interpreter. However, we do not think that proofs will become more complicated by that.

Furthermore, the development steps towards the WAM not yet considered remain to be done. The next refinement step introduces parts of the WAM instruction set: the list of clauses defining a predicate is now translated by an abstract compiler into a sequence of instructions that achieves the indexing of the clauses together with its backtracking management [Boi93]. The proofs for this step are presumed to be even more complex than the presented ones due to the formalization of suitable compiler assumptions.

Acknowledgements I wish to thank Tobias Nipkow and Franz Regensburger for helpful discussions and constructive criticism.

References

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, Cambridge, Massachusetts, 1991.

[Apt90] Krzysztof R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. Elsevier Science Publishers B.V., 1990.

[Boi93] Patrice Boizumault. *The Implementation of Prolog.* Princeton Series in Computer Science. Princeton University Press, Princeton, New Jersey, 1993.

[BR94] E. Börger and D. Rosenzweig. The WAM - Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications.* Elsevier, 1994.

[DM88] Saumya K. Debray and Prateek Mishra. Denotational and Operational Semantics for Prolog. *J. Logic Programming*, (5):61–91, 1988.

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer, 1987.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS.* Springer, 1994.

[Rus92] David M. Russinoff. A Verified Prolog Compiler for the Warren Abstract Machine. *J. Logic Programming*, (13):367–412, 1992.

[Sch95] G. Schellhorn. Von PROLOG zur WAM - Compilerverifikation mit KIV. Talk at the annual meeting of the GI section "Logic in Computer Science", Karlsruhe, Juni 1995.

[War83] D. H. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1083.

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