Journal of Universal Computer Science, vol. 3, no. 4 (1997), 377-413
submitted: 20/12/96, accepted: 21/4/97, appeared: 28/4/97 © Springer Pub. Co.

Reasoning about Abstract State Machines: The WAM Case Study

Gerhard Schellhorn
(Abt. Programmiermethodik,
Universität Ulm, 89069 Ulm, Germany
schellhorn@informatik.uni-ulm.de)

Wolfgang Ahrendt
(Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, 76128 Karlsruhe, Germany
ahrendt@ira.uka.de)

Abstract: This paper describes the first half of the formal verification of a Prolog compiler with the KIV ("Karlsruhe Interactive Verifier") system. Our work is based on [BR95], where an operational Prolog semantics is defined using the formalism of Gurevich Abstract State Machines, and then refined in several steps to the Warren Abstract Machine (WAM). We define a general translation of sequential Abstract State Machines to Dynamic Logic, which formalizes correctness of such refinement steps as a deduction problem. A proof technique for verification is presented, which corresponds to the informal use of proof maps. 6 of the 12 given refinement steps were verified. We found that the proof sketches given in [BR95] hide a lot of implicit assumptions. We report on our experiences in uncovering these assumptions incrementally during formal verification, and the support KIV offers for such 'evolutionary' correctness proofs.

1 Introduction

The Warren Abstract Machine (WAM, [War83]) today is the most popular target of Prolog compilers. Recently, a mathematical analysis of Prolog semantics and compiler correctness has become available with the work of Börger and Rosenzweig ([BR94], [BR95]).

The analysis in [BR95] starts by defining an operational semantics for Prolog in the formalism of (Gurevich) Abstract State Machines (ASMs, [Gur95]). We assume the reader to be familiar with this formalism as well as with the basic notions of Prolog (clauses with *! (cut)*, *true* and *fail*). The ASM is then refined in altogether 12 systematic steps to an ASM which executes WAM machine code. Each refinement introduces orthogonal concepts of the WAM. Parallel to the refinements, the Prolog program and the query are compiled to machine instructions. On intermediate levels the input of the ASM are machine instructions interspersed with uncompiled Prolog syntax. The compilation steps are not given as a concrete program, but specified by *compiler assumptions*. This still leaves some freedom for the implementation of a compiler, in particular several variants of the final WAM are still possible.

Based on the analysis of [BR95] and the proof sketched in [Sch94], this paper reports on our first steps towards the formal, machine-checked verification of compiler correctness with the KIV system. We will give a detailed discussion of the correctness proof for the first refinement from Prolog search trees to stacks of choicepoints, and summarize the results of the verification of the following 5 refinement steps.

Our motivations for beginning such a large case study — based on our current experience we estimate the necessary effort to develop a verified compiler to be around a person year — are the following:

- Mathematical analysis is an indispensable prerequisite for formal verification to be applicable. Nevertheless mathematical analysis will always omit details and have minor errors. These errors are due to the large complexity of correctness proofs, which is easily underestimated at first glance. The errors usually do not invalidate the analysis, but would still result in erroneous compilers. We want to demonstrate that the absence of such errors can be guaranteed by formal correctness proofs, making them a suitable counterpart to mathematical analysis.
- We want to show that Dynamic Logic (DL) as it is used in the KIV system can serve as a suitable starting point for the verification of Abstract State Machine refinements. In particular, the proof technique of commuting diagrams of Proof Maps, used informally in [BR95], can be formalized in DL.
- Finally, the requirements a system for the development of correct software must cope with are only discovered in ambitious case studies. Solving these requirements always leads to significant system improvements.

This paper is organized as follows: Section 2 introduces the first Abstract State Machine (ASM) which is used to define an operational semantics of Prolog (an "interpreter"). The first refinement towards the WAM is described. Both interpreters are explained with an example computation.

Section 3 introduces the KIV system, which was used to do the verification. Dynamic Logic (DL), the logic used in KIV to express properties of imperative programs, is given.

Section 4 translates sequential ASMs to algebraic specifications and imperative programs. The proof task of verifying the refinement between two ASMs is identified as a problem of program equivalence in DL. The deduction problem is reduced to the development of correct *coupling invariants*, which are formulas corresponding to *proof maps* in ASMs.

In Section 5 we try to give an impression of the verification process. Exemplarily we concentrate on the development of a coupling invariant for the first refinement step. As will be shown, this formula is extremely complex and can only be developed in several iterations.

Section 6 summarizes the verification of the other 5 refinement steps we have done so far.

Section 7 gives some related work and Section 8 concludes with an outlook on the continuing work on this case study.

2 The WAM Analysis of Börger and Rosenzweig

To make this exposition as self-contained as possible, the following two subsections introduce the first and second Abstract State Machine. An example will be given to explain the rules of the interpreters. We will closely follow [BR95] and deviate only in some notations. This will set up the verification task and enable us to discuss the problems we encountered in solving it (Sect. 5). The reader who knows [Sections 1.1, 1.2] from [BR95] may skip this section, and refer to it only for notational issues.

2.1 The First Interpreter: Search Trees

The two most important data structures needed to represent a *Prolog computation state* are the *sequence* of Prolog literals still to be executed and the current *substitution*. This state is modified by

1. unifying the first literal of the sequence, called *act* (activator), with the *head* of a clause
2. replacing *act* by the *body* of that clause
3. applying the unifying substitution to the resulting sequence and
4. composing the unifying substitution with the 'old' substitution.

If this leads to failure, alternative clauses have to be chosen, so backtracking is needed. Due to this the interpreter has to keep a record of the former computation states and the corresponding clause choice alternatives. This history is represented by a tree of *nodes*, connected from leaves to the root by a function *father*. Information on alternative clauses, which may be tried at a node n, is stored as a list *cands(n)* of candidate nodes. Each node in this list refers via a function *cll* to a clause line in the Prolog program. The initial *cands*-list is constructed with the help of a function *procdef*, which is assumed to return the program lines containing the candidate clauses for a given literal. The current computation state is carried by a distinguished node, the *currnode*.

The ability to handle the *cut* still requires an extension of the state representation. A *cut* causes nothing but modification of the backtracking information, updating the *father* of the current node to the *father* of that computation state whose *act* caused the introduction of the considered *cut*. For this we have to 'remember' where a *cut* has been introduced. An uniform solution is to attach the *father* of the (old) *currnode* to each clause body being introduced to the literal sequence. This attachment divides the sequence of literals into subsequents, called goals, each decorated by one node, called *cutpt* (cutpoint). The resulting sequence is called *decglseq* (decorated goal sequence).

To introduce the rules of the ASM we will now consider the evaluation of the query `?- p.` on the following Prolog program:

```
1 p :- fail.                3 q.
2 p :- q,!,true.            4 p.
```

which is stored as the value of a constant *db* (database) in the initial algebra of the ASM. Line numbers are shown explicitly in the program for explanatory purposes ([BR95] uses an abstract sort *code* for clause lines. The use of natural numbers here is only to facilitate understanding).

The query `?- p.` is stored as the *decglseq* of node A in the initial search tree depicted in Fig. 1. The two nodes (labeled $\perp$ and A) form the initial domain of a dynamic universe *node*, which is extended by the rules of the ASM. Tree structure is stored in a function *father : node* $\rightarrow$ *node*, indicated by the arrow in Fig. 1, so we have *father(A)* $= \perp$ (the father of $\perp$ is undefined). Node $\perp$ is the root node of the tree. It serves only as a marker when to finish search and does not carry information. The initial *currnode* is A, as indicated by the double circle. The (initially empty) substitution *sub(currnode)* attached to this node is not shown in the figures, since it does not matter in the example we consider.

The ASM run is controlled by two program variables (i.e. 0-ary dynamic functions) *mode* and *stop*. The value of *mode* switches between *call* and *select*, while the value of *stop* remains *run* until it finally changes to *halt*. This stops the evaluation by falsifying all rule guards.

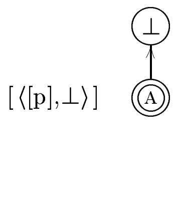


Fig. 1.

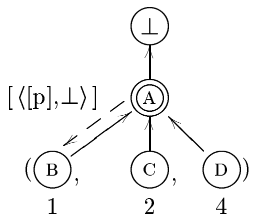


Fig. 2.

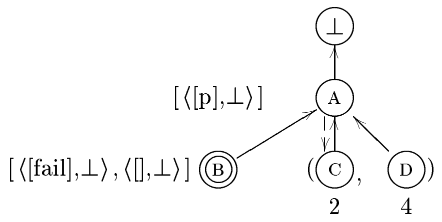


Fig. 3.

Since the functions used in the following rules are often applied to the currnode, we use the following abbreviations:

$$
\begin{array}{ll}
\text{father} & \equiv \text{father(currnode)} \\
\text{cands} & \equiv \text{cands(currnode)} \\
\text{sub} & \equiv \text{sub(currnode)} \\
\text{decglseq} & \equiv \text{decglseq(currnode)}
\end{array}
$$

Also the components of *decglseq* are abbreviated as shown in the following diagram:

$$
\mathit{decglseq} = [\, \langle \underbrace{[\, \overbrace{g_{1,1}}^{act}, g_{1,2}, \ldots, g_{1,k_1} \,]}_{goal}, \overbrace{n_1}^{cutpt} \rangle , \ldots , \langle [\, g_{m,1}, \ldots, g_{m,k_m} \,], n_m \rangle \,]
$$

$$
\mathit{cont} = [\, \langle \quad [\, g_{1,2}, \ldots, g_{1,k_1} \,], n_1 \rangle , \ldots , \langle [\, g_{m,1}, \ldots, g_{m,k_m} \,], n_m \rangle \,]
$$

The continuation *cont*, which is the *decglseq* without *act*, will later on help to describe the construction of a new *decglseq*.

In *call* mode, which is the initial mode, the *cands* information is computed (for a guard which involves *act*, we assume that decglseq $\neq$ [], goal $\neq$ []):

call rule

```
IF stop = run & is_user_defined(act) & mode = call
THEN LET [cll_1,...,cll_n] = procdef(act,db)
     EXTEND node BY temp_1,...,temp_n
     WITH
          father(temp_i) := currnode
          cll(temp_i) := cll_i
          cands := [temp_1,...,temp_n]
     ENDEXTEND
     mode := select
```

The EXTEND construct, by expanding the universe *node*, allocates one node for every clause whose head 'may unify' with the literal *act*. This list of clause lines is computed by *procdef(act,db)* and is assumed to contain at least those clauses, whose head unify with the activator, and at most those with the same leading predicate symbol as *act*. The result of the rule application is depicted in Fig. 2.

The *cands* list (of node A) is indicated by a dashed arrow to its first element and brackets around the elements. The clause lines corresponding to the candidates are attached to the new nodes via the function *cll*, as shown by numbers below the nodes. The change of the *mode* variable activates the select rule:

select rule

```
IF stop = run & is_user_defined(act) & mode = select
THEN IF cands = []
     THEN backtrack
ELSE LET clause = rename(clause(cll(first(cands))),vi)
     LET mgu = unify(act,head(clause))
     IF mgu = failure
     THEN cands := rest(cands)
     ELSE currnode := first(cands)
          decglseq(first(cands)) :=
          apply(mgu,[⟨body(clause), father⟩|cont])
          sub(first(cands)) := sub ∘ mgu
          cands := rest(cands)
          vi := vi + 1
          mode := call
```

where

```
backtrack ≡ IF father = ⊥
            THEN stop := halt
                 subst := failure
            ELSE currnode := father
                 mode := select
```

This rule causes backtracking if there are no (more) alternatives to select. Otherwise, by repeated application, it removes all candidates whose heads do not unify with *act*. When the first candidate is reached, for which a most general unifier *mgu* exists (variable index *vi* is used to rename the implicitly universal quantified clause variables to new instances), this node becomes *currnode*. A new *decglseq* is computed by replacing the activator of the old *decglseq* with the body of the 'selected clause'. As a cutpoint the *father* of the old *currnode* is attached to this new goal. The *mgu* is applied to the resulting *decglseq* and composed (with $\circ$) with the old substitution *sub*.

The result of applying the select rule in our example is shown in Fig. 3. Now mode is *call* again, but since the activator *fail* is not user defined, instead of the call rule the fail rule fires:

fail rule
IF stop = run & act = fail
THEN backtrack

It sets *currnode* to A again. Note that node B is not formally deallocated (i.e. it remains in the *node* universe). Again in *select* mode, the next candidate node of A, node C, is selected and its *decglseq* is computed as $[\langle[\text{q,!,true}],\perp\rangle, \langle[],\perp\rangle]$. Then the call rule allocates one new candidate node E for the only appropriate clause q. After selection of node E the ASM arrives at the state shown in Fig. 4.

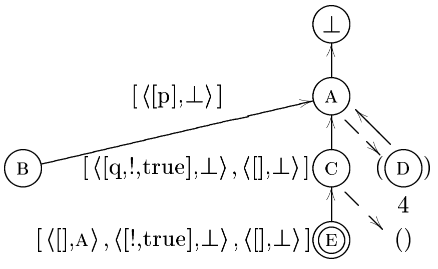


Fig. 4.

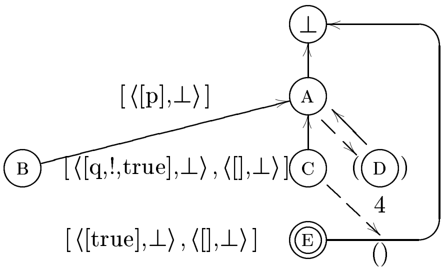


Fig. 5.

With an empty *goal* the goal success rule fires, after which the activator is a cut.

goal success rule
IF stop = run & decglseq $\neq$ [] & goal = []
THEN decglseq := rest(decglseq)

cut rule
IF stop = run & act = !
THEN father := cutpt
decglseq := cont

The cut succeeds (is removed) and the *father* function is updated to the cutpoint decorating the goal where the cut appears (see Fig. 5). This action is the only purpose for decorating goals with nodes.

Finally, the interpreter executes the following rule for the activator *true*,

true rule
IF stop = run & act = true
THEN decglseq := cont

and with another two applications of the goal success rule, *decglseq(E)* becomes empty. Since this means that the initial query is completely solved, the ASM sets the answer substitution *subst* to *sub(currnode)* (here, of course, the empty substitution).

final success rule
IF stop = run & decglseq(currnode) = []
THEN stop := halt
subst := sub

Since *stop* is no longer *run*, no more rule is applicable and the ASM halts.

In a variant of our example program, where clause `p :- q,!,true` is changed to `p :- q,!,r`, we would also arrive at the situation of Fig. 5. But now a node F would be allocated for `r` with an empty list of candidates, since no clauses for predicate `r` are given. *select* mode, finding no more alternatives, would backtrack from nodes F and E. Since the father of E is the root node $\perp$, execution would finally stop with *stop = halt* and *subst = failure*.

2.2 The Second Interpreter: Stacks of Choicepoints

Here we summarize the first refinement of the ASM described above towards the Warren Abstract Machine (WAM), following [BR95], [Section 1.2]. There are three main differences between the first and the second ASM:

First, function *father* is renamed to *b*. This change indicates that *b* now points *b*ackwards in a chain of nodes, which form a *stack*.

Second, the new ASM (ASM2) provides the registers *cllreg, decglseqreg, breg* and *subreg* corresponding to *cll, decglseq, father* and *sub* applied to the *currnode*. Thereby it avoids allocation of *currnode*.

Third, ASM2 attaches the *first* candidate directly via the *cll*-function, instead of providing a list of candidate nodes. This is possible if we assume that clauses whose head starts with the same predicate are stored in successive clause lines followed by a special marker *nil*. The ("compiled") representation of our example Prolog program for ASM2 thus has to look like

```
1 p :- fail.        3 p.      5 q.
2 p :- q,!,true.    4 nil     6 nil
```

A new *procdef'* function is needed, such that *procdef'(act,db)* now yields the first clause line whose head may unify with the activator *act*. For *act = p* we get *procdef'(p,db) = 1*, the first line of a clause with head *p*. The connection to the old *procdef* function is stated in the following *compiler assumption* about function *compile*, which is used as an axiom in correctness proofs. Let *db' = compile(db)* in

$$\begin{aligned} &\text{mapcl(procdef(act,db),db)} \\ &= \text{mapcl(clls(procdef'(act,db'),db'),db')} \end{aligned} \tag{1}$$

Here *clls* collects successive line numbers, until a *nil* is found, and *mapcl* selects the clauses at these line numbers. Instead of allocating a candidate list, ASM2 simply assigns *procdef'(act,db)* to *cllreg*. Incrementing *cllreg* then corresponds to removing a candidate from *cands*. If the clause at *cllreg* becomes *nil*, no more candidates are available. Allocation of a new node is now done only in *select* mode, when a new candidate clause is visited.

We now give the rules that have been changed.

call rule

```
IF stop = run & is_user_defined(act) & mode = call
THEN cllreg := procdef'(act,db')
     mode := select
```

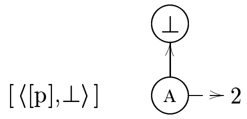
select rule

```
IF stop = run & is_user_defined(act) & mode = select
THEN IF clause(cllreg) = nil
THEN backtrack
ESLE LET clause = rename(clause(cllreg),vi)
     LET mgu = unify(act, head(clause))
     IF mgu = failure
     THEN cllreg := cllreg + 1
     ELSE EXTEND node BY temp
          WITH
               breg := temp
               b(temp) := breg
               decglseq(temp) := decglseqreg
               sub(temp) := subreg
               cll(temp) := cllreg + 1
          ENDEXTEND
          decglseqreg := apply(mgu,[⟨body(clause),ctreg⟩|cont])
          subreg := subreg ∘ mgu
          vi := vi + 1
          mode := call
```

```
backtrack ≡ IF breg = bottom
            THEN stop := halt
                 subst := failure
            ELSE decglseqreg := decglseq(breg)
                 subreg := sub(breg)
                 breg := b(breg)
                 cllreg := cll(breg)
                 mode := select
```

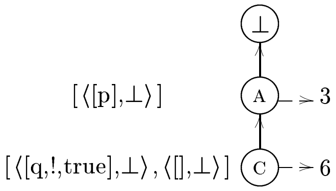
In the other rules of the previous ASM, only function *father* is renamed to *b*, and the abbreviations *decglseq*, *father* and *sub* are replaced by *decglseqreg*, *breg* and *subreg*.

In our example, ASM2 goes through the states shown in Fig. 6 and 7, which correspond to those from Fig. 3 and 4 for the first ASM.



decglseqreg = [⟨[fail],⊥⟩,⟨[],⊥⟩]
breg = A

Fig. 6.



decglseqreg = [⟨[],A⟩,⟨[!,true],⊥⟩,⟨[],⊥⟩]
breg = C

Fig. 7.

Dashed arrows now point to the *cll* of a node. The allocation of a node corresponding to B is avoided. The values which were attached to it in the first ASM are now only stored temporarily in registers. Also candidate node D is not allocated. Now, the nodes which may be visited in the future are always reachable from *breg* via the b function. They form a stack, but note that there may still be abandoned nodes in the *node* universe, which are no longer reachable. This causes problems for verification (see Sect. 5). The tuple of values *decglseq(n)*, *sub(n)*, *cll(n)* and *b(n)* attached to a stack node n is usually called a *choicepoint*.

Finally, it should be remarked that our compiler assumption (1) is weaker than the one given in [BR95], which (to avoid a compilation step) identifies databases, assuming that clauses were already grouped according to different predicates on the first level, and requires

$$\text{procdef(act,db)} = \text{clls(procdef'(act,db),db)} \tag{2}$$

But assumption (2) can not be fulfilled for definitions of the *procdef* function, which are more specific than looking only at the leading predicate symbol. In this case even our liberalized compiler assumption requires duplication of Prolog code (code will be shared again, when switching instructions are introduced).

3 KIV

The KIV system ([Rei95], [RSS95], [RS95],[RSS97]) is an advanced tool for engineering high assurance systems. It supports the entire design process from formal, algebraic specifications to executable verified code. KIV relies on first-order algebraic specifications to describe hierarchically structured systems in the style of ASL, [SW83]: Specifications are built up from elementary first-order specifications with the operations enrichment, union, renaming, parameterization and actualization. Their semantics is the class of all models (loose semantics). Generation principles (also called reachability constraints) like "nat generated by 0, succ" restrict the semantics to term-generated models. The constraints are reflected by induction principles in the calculus for theorem proving used in KIV.

Specification components can be implemented using modules which contain imperative programs. The programs contain the usual constructs found in imperative languages: Assignment $x := t$ (also parallel assignments $\underline{x} := \underline{t}$), conditional, compound, local variables, while-loops and recursive procedures with both value-and reference parameters. Although the imperative programs are written in a PASCAL-like notation, they are *abstract* programs. The operations used on the right hand sides of assignments are those given in an algebraic specification, not the concrete operations available in PASCAL.

To reason about abstract programs, KIV uses Dynamic Logic(DL). Dynamic Logic is an extension of first-order logic by formulas $\langle\alpha\rangle\,\varphi$ (read "diamond α φ"), where α is an imperative program, and φ is again a DL-formula. The informal meaning of this formula: "α terminates and φ holds afterwards" should be sufficient for the purpose of this paper. A formal definition of DL can be found in [Har84], [Gol82].

DL can be used to express total correctness of a program α with precondition φ and postcondition ψ as $\varphi \rightarrow \langle\alpha\rangle\,\psi$. $\varphi \rightarrow \neg\ \langle\alpha\rangle \neg\ \psi$ expresses partial correctness. Program inclusion (and equivalence) with respect to some program variables $\underline{\mathrm{x}}$ can be formalized as $\langle\alpha\rangle\,\underline{\mathrm{x}} = \underline{\mathrm{x}}_0 \rightarrow \langle\beta\rangle\,\underline{\mathrm{x}} = \underline{\mathrm{x}}_0$. This will be important for our case study.

To deduce properties of specifications and to verify program modules, KIV offers an advanced interactive deduction component. It combines a high degree of automation with an elaborate proof engineering environment. The proof strategy is based on induction, symbolic evaluation of programs and on simplification for first-order theories. With these "tactics", goals of the underlying sequent calculus are reduced to simpler ones until axioms are reached.

To automate proofs, KIV offers a number of *heuristics*, see [RSS95]. These can be chosen freely, and changed any time during the proof. Heuristics may be adapted to specific applications without changing the implementation. Usually, the heuristics manage to find 80 – 100 % of the required proof steps automatically.

In KIV the user can freely create, change or delete specifications, and theorems. Theorems can be proved in any order (not only bottom-up). An elaborate *correctness management* ensures that changes do not lead to inconsistencies (e.g. it prevents cycles in the hierarchy of proofs).

Since frequently the problems found in the development of correct software are not to verify proof obligations affirmatively but rather to interpret failed proof attempts, KIV offers a number of *proof engineering* facilities to support the iterative process of (failed) proof attempts, error detection, error correction and re-proof. Proof trees can be inspected using a graphical interface. Dead ends can be cut off, proof decisions may be withdrawn both chronologically and non-chronologically. Both successful and failed proof attempts are reused automatically to guide the verification after correction ([RS95]). This goes beyond proof replay (or proof scripts).

4 From Abstract State Machines to Dynamic Logic

In this section we will give a translation of sequential Abstract State Machines, as they are used in this case study, to Algebraic Specifications and Dynamic Logic. The translation is essentially one on one, because both ASM and DL feature imperative programs. Therefore no encoding of programs (as functions or relations over a state) is required. This makes DL a good starting point for verifying properties of sequential ASMs. The translation is done in three steps: First, we will give a translation of the abstract data used into an algebraic specification. The second step translates the set of rules of an ASM into an imperative program. In the third step we will identify equivalence of two ASMs as program equivalence in DL, and give a proof technique corresponding to the use of "Proof Maps". The three steps are described in the following three subsections.

4.1 Translation of Specifications

To translate the abstract data types of an Abstract State Machine into an algebraic specification, we first have to separate the *static* and the *dynamic* part of the signature. The dynamic part contains those functions and sorts, for which the set of rules contains updates. The other, static part typically contains data types like lists, numbers and suitable operations on them. These can be specified algebraically. Partial functions (present in ASMs but not in the algebraic specifications used in KIV) are usually handled using underspecification. E.g. for natural numbers, we simply do not specify the predecessor of zero. With respect to the loose semantics of algebraic specifications, we then have that *pred(0)* is an arbitrary natural number. This is sufficient, except for two cases: The first is, if we explicitly want to work with the "undefined" element, e.g. if the rules of the ASM contain definedness tests. This case does not occur in our ASMs (there *are* error elements, e.g. the result *failure* of unification, but these are *defined* elements). It would have to be handled by introducing explicit error elements.

The second exception is, when a *partial* function is defined to be the least fixpoint of recursive equations. For this latter exception, there are indeed a number of examples in our case study, namely the functions *clls* ([BR95],p. 17), F, G (p. 23f) and *chain* (p. 25 and p. 28), which all collect a list of addresses by traversing some pointer structure. But in general, recursive equations are not sufficient to characterize the intended *least* fixpoint, which is undefined on infinite (or cyclic) pointer structures and defined otherwise (the *chain* function defined on p. 28 indeed has other fixpoints).

To fix this problem in partial first-order logic ([Wir90], on which the recursive definitions in [BR95] are based) an explicit characterization of the domain of the least fixpoint would be required.

In Dynamic Logic there is an easier way to handle the problem, since we can explicitly talk about least fixpoints. Rather than specifying a first-order function, we write recursive programs for *clls*, *chain* etc., and assert (in the compiler assumption) that they terminate on all inputs delivered by the compiler.

Data types, which are not completely specified in the ASM, pose no problem for algebraic specification. E.g. on the first level nothing is said about the structure of terms. For the algebraic specification this means that the sort *term* is a parameter, which will be actualized with a concrete definition of terms at a later stage of development.

Having translated the static part, the dynamic part is somewhat more complex. The central idea here is to encode the domains of *dynamic* sorts and the semantics of *dynamic* functions as the values of (ordinary first-order) variables. Updates then are translated to assignments in DL.

Since the domains of dynamic sorts in an ASM usually are *finite* sets of elements (in our case: finite sets of nodes) a (standard) specification of finite sets is used. A variable s stores the current domain, and a sort update, which extends the domain with a new element *temp* corresponds to the two assignments *temp := new(s); s := insert(temp,s)* in DL, where function *new : set* $\rightarrow$ *node* is specified by the axiom: $\neg$ *new(s)* $\in$ *s.*

0-ary functions are simply translated to ordinary first-order variables. For other dynamic functions, the case with n $>$ 1 arguments can be reduced to the case with one argument by adding an appropriate tuple sort. For unary functions we essentially have to encode the (second-order) datatype of a *function* into a first-order datatype, so that a function can become the value of a variable. This can be achieved with the data type shown in Fig. 8, which defines functions from domain *dom* to codomain *codom*:

generic specification
parameter sorts dom, codom;
target sorts dynfun
  functions
    cf : codom $\rightarrow$ dynfun;
    . ˆ . : dynfun $\times$ dom $\rightarrow$ codom;
    . + (. / .) : dynfun $\times$ dom $\times$ codom $\rightarrow$ dynfun;
  variables f : dynfun; x, y : dom; z : codom;
  axioms
    cf(z) ˆ x = z,
    (f + (x / z)) ˆ x = z,
    x $\neq$ y $\rightarrow$ (f + (x / z)) ˆ y = f ˆ y
end generic specification

Fig. 8: Algebraic specification of dynamic functions

The data type contains a constant function *cf(z)* for every codomain-element *z*. Application of this function to any domain element x (with an apply-operation, for convenience written as an infix-circumflex) just gives *z*, as stated by the first axiom. With a suitable "dummy"-element *z*, constant functions are used as initial values. E.g. the *cands* function (now a variable) is initialized with value *cf([])*, the function delivering an empty list of candidates for every node.

A function update *f(x) := t* in the ASM-formalism becomes an assignment *f := f + (x / t)* to variable *f* in DL. It sets the new value of *f* to the result of mixfix-operation *f + (x / z)*, which is defined by the last two axioms to be the appropriately modified function.

Finally note that we did not add an extensionality axiom

$$\mathrm{f} = \mathrm{g} \leftrightarrow \forall\, \mathrm{x}.\ \mathrm{f}\ \hat{}\ \mathrm{x} = \mathrm{g}\ \hat{}\ \mathrm{x} \tag{3}$$

to the specification, in contrast to the usual methodology used in KIV to specify non-free data types. Such axioms would have allowed us to deduce equalities between functions like *f = f + (x / f ˆ x)*. Since such (higher-order) equalities are not expressible in the ASM-formalism, we expected not to need them in the translated version either. And indeed, there was no need for equations between functions in verification. Also no induction principle on dynamic functions like 'dynfun **generated by** cf, . + (. / .)' was needed.

The specification can be viewed as an abstract version of a store structure. It could be implemented e.g. by association lists. In our case, where the domain is pointers, in fact the final implementation in the WAM will be a part of computer memory.

Putting the translation of the static and dynamic part together we get a structured specification composed of about 40 subspecifications for the first refinement. Many of the specifications (lists, pairs, etc.) could be retrieved from the library, together with a lot of simplification rules useful for verification.

A first version of the specification was written within some hours and needed only minor corrections.

4.2 Translation of Programs

Given the translation of the static and dynamic part of the ASM from the first two translation steps, it remains to translate the set of rules to an imperative program. The result, shown for the first interpreter, is procedure *ASM1#*:

ASM1#(db, query; var subst)	BODY1#(var $\underline{\mathrm{x}}$)
begin	**begin**
var $\underline{\mathrm{x}}$:= $\underline{\mathrm{t}}$ **in**	**if** {test of rule_1} **then** {updates of rule_1} **else**
while stop = run	...
do BODY1#($\underline{\mathrm{x}}$)	**if** {test of rule_n} **then** {updates of rule_n}
end	**end**

The inputs of *ASM1#* are the Prolog program *db* and the *query*. The reference parameter *subst* is used as the result value for the answer substitution. *ASM1#* starts by initializing the variables $\underline{\mathrm{x}}$ = [stop, subst, decglseq, father, ...] with a vector $\underline{\mathrm{t}}$ of suitable initial values. Then it enters a while loop with test *stop = run* and body *BODY1#*. *BODY1#* has the variables $\underline{\mathrm{x}}$ as reference parameters, and uses them as input and output. It consists of a case analysis, which selects an applicable rule and executes its updates.

Translation of the ASM rules somewhat increases their size, because abbreviations have to be expanded using variable declarations. The translated code of ASM1 and ASM2 is each about 120 lines of PASCAL-like statements.

4.3 From Proof Maps to Coupling Invariants

Correctness and completeness of interpreter refinement is formalized in DL as the assertion of the following program equivalence:

$$\begin{array}{ll} & \langle \mathrm{ASM1\#(db,query;subst_1)} \rangle\, \mathrm{subst_1} = \mathrm{val} \\ \leftrightarrow & \langle \mathrm{ASM2\#(compile(db),query;subst_2)} \rangle\, \mathrm{subst_2} = \mathrm{val} \end{array} \tag{4}$$

This formula states that if and only if the first interpreter *ASM1#* terminates, then so does the second *ASM2#* with the same answer substitution.

Function *compile* relates the two Prolog programs. The compiler assumption, which specifies *compile*, is given in the axioms of the specification (see formula (1) in Sect. 2.2), over which the program equivalence must be proved. Variable *val* is used to store the common result value of both interpreters (this variable is not modified by *ASM1#* and *ASM2#*).

The notions of *Correctness* and *Completeness* from [BR95] directly correspond to the implication from right to left and from left to right.

Proof maps $\mathcal{F}$ are defined in [BR95] to map static algebras of a 'concrete' ASM to algebras of an 'abstract' ASM. They are used to sketch correctness proofs for the equivalence of two ASMs. The basic argument is as follows: The three proof obligations

PO1: The initial states (algebras) of the two ASMs are related via $\mathcal{F}$
PO2: Fig. 9 commutes for every corresponding pair of rules R and R' of the first and second interpreter (with $\mathcal{A}_0$ and $\mathcal{B}_0$ being the results of rule application to $\mathcal{A}$ and $\mathcal{B}$)
PO3: Two final states which are related via $\mathcal{F}$ store the same answer substitution

imply (by induction on the number of rule applications) that both ASMs are equivalent.

$$\begin{tikzcd} \mathcal{A} \arrow[r, "\mathrm{R}"] & \mathcal{A}_0 \\ \mathcal{B} \arrow[u, "\mathcal{F}"] \arrow[r, "\mathrm{R'}"] & \mathcal{B}_0 \arrow[u, "\mathcal{F}"'] \end{tikzcd}$$

Fig. 9.

$$\begin{tikzcd} \underline{\mathrm{x}} \arrow[r, "\mathrm{R}"] \arrow[d, leftrightarrow, "\mathrm{INV}"'] & \underline{\mathrm{x}}_0 \arrow[d, leftrightarrow, "\mathrm{INV}"'] \\ \underline{\mathrm{x}}' \arrow[r, "\mathrm{R'}"] & \underline{\mathrm{x}}'_0 \end{tikzcd}$$

Fig. 10.

This informal argument can be formalized in DL to prove (4) as follows: The (dynamic parts of the) algebras involved in a computation have been replaced by the values of the vector of program variables. If we name the program variables, *ASM1#* and *ASM2#* compute on, differently, say $\underline{\mathrm{x}}$ = [stop, subst, decglseq, vi, father,...] and $\underline{\mathrm{x}}$' = [stop', subst', decglseq', vi', b, cll, ...], then the direct translation of a proof map is a function, which maps a tuple of values for $\underline{\mathrm{x}}$' to a tuple of values for $\underline{\mathrm{x}}$. Since we found no need for the connection between $\underline{\mathrm{x}}$ and $\underline{\mathrm{x}}$' to be a function, we allow it to be an arbitrary *relation*, which we describe by a (DL-)formula INV($\underline{\mathrm{x}}$,$\underline{\mathrm{x}}$'), which involves the free variables $\underline{\mathrm{x}}$ and $\underline{\mathrm{x}}$'. We call this formula a *coupling invariant.* To use this formula in the proof, we split (4) into two goals, one for each direction of the biimplication. Since the following steps are the same for both directions, we concentrate on the one from right to left (correctness). This implication can be simplified to the following statement about the two while loops involved:

$$\begin{array}{l} \quad\ \underline{\mathrm{x}} = \underline{\mathrm{t}} \wedge \underline{\mathrm{x}}' = \underline{\mathrm{t}}' \\ \ \ \wedge \langle \mathbf{while}\ \mathrm{stop}' = \mathrm{run}\ \mathbf{do}\ \mathrm{BODY2\#}(\underline{\mathrm{x}}')\rangle\, \mathrm{subst}' = \mathrm{val} \\ \rightarrow \langle \mathbf{while}\ \mathrm{stop} = \mathrm{run}\ \mathbf{do}\ \mathrm{BODY1\#}(\underline{\mathrm{x}})\rangle\, \mathrm{subst} = \mathrm{val} \end{array} \tag{5}$$

Now the basic idea of our proof will be an induction on the number i of iterations, the loop executes BODY2#($\underline{\mathrm{x}}$'). Technically, induction over the number of while loop iterations is possible using the *Omega-Axiom* of Dynamic Logic:

$$\langle \textbf{while } \varepsilon \textbf{ do } \alpha\rangle \varphi \leftrightarrow \exists\, \mathrm{i}.\ \langle \textbf{loop } (\textbf{if } \varepsilon \textbf{ then } \alpha) \textbf{ times } \mathrm{i}\rangle (\varphi \wedge \neg\, \varepsilon) \tag{6}$$

In this axiom, i is a natural number (we have an induction principle), and the loop program **loop** α **times** i indicates execution of α i times. The two axioms for the loop-construct in DL therefore are:

$$\begin{aligned}
&\langle \textbf{loop } \alpha \textbf{ times } 0\rangle \varphi \leftrightarrow \varphi \\
&\langle \textbf{loop } \alpha \textbf{ times } \mathrm{i} + 1\rangle \varphi \leftrightarrow \langle \alpha\rangle \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i}\rangle \varphi
\end{aligned} \tag{7}$$

The axiom (6) intuitively says that a formula φ holds after execution of a while-loop, iff it holds after sufficiently many iterations of **if** ε **then** α and the test ε is false afterwards. Application of (6) on both while-loops we can then generalize our goal (5) using the coupling invariant, resulting in the following three goals:

$$\mathrm{INV}(\underline{\mathrm{t}},\underline{\mathrm{t}}') \tag{8}$$

$$\mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \rightarrow \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{subst} = \mathrm{subst}' \tag{9}$$

$$\begin{aligned}
&\quad \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \\
&\wedge\ \langle \textbf{loop if } \mathrm{stop} = \mathrm{run} \textbf{ then } \mathrm{BODY2\#}(\underline{\mathrm{x}}') \textbf{ times } \mathrm{i}\rangle\, \underline{\mathrm{x}}' = \underline{\mathrm{x}}'_0 \\
&\rightarrow \exists\, \mathrm{j}.\ \langle \textbf{loop if } \mathrm{stop}' = \mathrm{run} \textbf{ then } \mathrm{BODY1\#}(\underline{\mathrm{x}}) \textbf{ times } \mathrm{j}\rangle\, \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}'_0)
\end{aligned} \tag{10}$$

The first goal states that the coupling invariant holds before execution of the two while loops, corresponding to PO1. The second goal says that the coupling invariant implies that both while stop at the same time with the same answer substitution (PO2). These two goals are usually rather trivial. The complexity of verification is buried in finding an invariant INV such that the last goal (10) is provable. This last goal states that for every number i of rules the second interpreter executes there is a number j of rule applications of the first interpreter such that Fig. 11 commutes.

$$\begin{tikzcd}
\underline{x}'_0 \arrow[r, "\mathrm{R}'_1"] \arrow[d, leftrightarrow, "\mathrm{INV}"'] & \underline{x}'_1 \arrow[r, "\mathrm{R}'_2"] & \cdots \arrow[r, "\mathrm{R}'_\mathrm{i}"] & \underline{x}'_\mathrm{i} \arrow[dr, leftrightarrow, "\mathrm{INV}"] & \\
\underline{x}_0 \arrow[r, "\mathrm{R}_1"] & \underline{x}_1 \arrow[r, "\mathrm{R}_2"] & \cdots \arrow[r] & \cdots \arrow[r, "\mathrm{R}_\mathrm{j}"] & \underline{x}_\mathrm{j}
\end{tikzcd}$$

Fig. 11

Since in the verification of the first refinement the correspondence between the two sets of rules is one to one (for some other refinements a generalization of the proof technique is required, see section 6) it suffices to choose $j = i$ and to induce on i. The induction step (the base case with i = 0 is trivial) reduces to

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \wedge \mathrm{stop}' = \mathrm{run} \wedge \langle \mathrm{BODY2\#}(\underline{\mathrm{x}}')\rangle\, \underline{\mathrm{x}}' = \underline{\mathrm{x}}'_0 \\ \rightarrow \langle \textbf{if}\ \mathrm{stop} = \mathrm{run}\ \textbf{then}\ \mathrm{BODY1\#}(\underline{\mathrm{x}})\rangle\, \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}'_0) \end{array} \tag{11}$$

which is the formalization of fig. 9, as shown in fig. 10. Having a closer look at goal (11), we find that having proved it, we not only have shown *correctness*, but also solved the problem of *completeness*. This is true, since proving the direction from right to left in (4) only exchanges the roles of the interpreters, and doing the same proof steps as before we will end up with the following goal dual to (11) in the induction step:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \wedge \mathrm{stop} = \mathrm{run} \wedge \langle \mathrm{BODY1\#}(\underline{\mathrm{x}})\rangle\, \underline{\mathrm{x}} = \underline{\mathrm{x}}_0 \\ \rightarrow \langle \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then}\ \mathrm{BODY2\#}(\underline{\mathrm{x}}')\rangle\, \mathrm{INV}(\underline{\mathrm{x}}_0,\underline{\mathrm{x}}') \end{array} \tag{12}$$

Both goals differ only in the way they treat termination of the two loop bodies. (11) claims that termination of *BODY2#* implies termination of *BODY1#*, (12) asserts the reverse implication. But since both loop bodies just apply one rule, they are flat programs. To show their termination is trivial. Therefore, having proved (11), using it as a lemma in (12) will finish the proof immediately. The proof of (11) splits into 7 cases, one for each corresponding pair of rules.

5 Verification of the First Refinement Step

In this section we will describe the verification of the first refinement step in detail. Unfortunately we cannot avoid to confront the reader with a lot of details, which were uncovered during the verification. Only the consideration of these details leads to the detection of hidden assumptions, which ultimately *guarantee* the correctness of the refinement. The reader who is not interested in the details may just have a look at the 9 initial properties, as they were given in [BR95] at the beginning of the following subsection, and compare them to final coupling invariant shown at the end of subsection 5.2.4. This should give an impression about the work needed to translate an informal mathematical argument to a complete, formal proof.

5.1 The Initial Coupling Invariant

As was discussed in the previous section, the critical point for a successful formal proof is to find a coupling invariant $\mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}')$, such that goal (11) is provable. Some rough indication how such an invariant might look like is already given in [BR95], p.17f. There an auxiliary function F is suggested, which maps the nodes in the stack (built by the second ASM) to corresponding nodes in the tree (built by the first ASM) (see Fig. 12.).

[Sch94] pointed out that F cannot be given statically, but has to be defined by induction on the number of rule applications. That is, in terms of our ASM1# procedure mentioned above, induction on the number of loop iterations. This requires a formalism, where a dynamic function can be updated by proof steps.

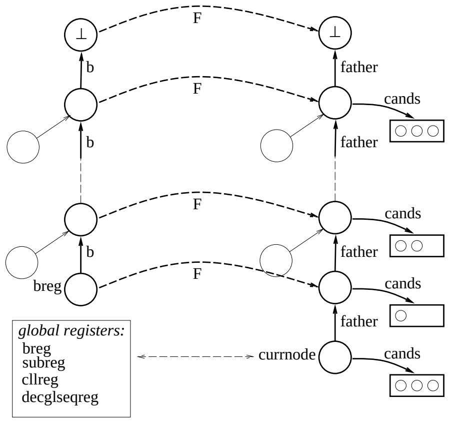


Fig. 12.

In DL, the answer comes for free since we made *dynamic* functions available as a datatype (see specification 'Dynfun', Sec. 4.1). Let F be of sort *dynfun*, which means it is a data structure and therefore can be (first order) quantified. Our coupling invariant then asserts the *existence* of a suitable function F for every two corresponding interpreter states. F then gets updated by *instantiation*. Based on this dynamic function the properties listed on p.17f of [BR95] translate to the following conjuncts in our invariant (in ambiguous cases the variables of the second interpreter are primed):

$\exists$ F:
1 decglseq ^ currnode = decglseqreg
2 sub ^ currnode = subreg
3a mapcl(map(cll, cands ^ currnode),db) = mapcl(clls(cllreg,db'),db')
3b every(father,cands ^ currnode, currnode)
4 father ^ currnode = F ^ breg
5 decglseq ^ (F ^ n) = decglseq' ^ n
6 sub ^ (F ^ n) = sub' ^ n
7a mapcl(map(cll, cands ^ (F ^ n)),db) = mapcl(clls(cll' ^ n,db'),db')
7b every(father, cands ^ (F ^ n), (F ^ n))
8 father ^ (F ^ n) = b ^ n
9 F ^ $\perp$ = $\perp$

Here *every(father,cands ^ n, n)* means that *n* is the *father* of all its *cands*.

The equations 1 and 5 actually do not hold. Although the goals of the decglseq(reg)s are identical, the incorporated cutpoints are not related by identity but by F. Due to this 1 and 5 were replaced by:

1 decglseq ˆ currnode = F_d(F, decglseqreg)
5 decglseq ˆ (F ˆ n) = F_d(F, decglseq' ˆ n)

where F_d applies F to all cutpoints of its second argument. In [Sch94] this was added to the coupling invariant together with the equations:

10 stop = stop' $\wedge$ mode = mode' $\wedge$ vi = vi'

Formulas 1 – 10 formed the first version of the coupling invariant INV($\underline{x}$,$\underline{x}$') when we began verification with the KIV system.

Up to here, INV concentrates on the dependencies between the two interpreters (the only exceptions are 3b and 7b). The reason is that at the beginning one might believe that invariant properties of single abstract machines (if at all needed for the proof) come for free. But they do not, as we will show below.

5.2 Development of the Correct Coupling Invariant

This first version of the coupling invariant was not sufficient. The completion of the coupling invariant took much more time than proving the finally valid version. Without going too much in details, we give a rough overview of this *search* rather then describing the logical deduction. We explain how hidden assumptions were detected (if the proof needed them explicitly) and how proving these new formulas leaves new gaps and so on. We take this proof-historical point of view to emphasize the evolutionary nature of solving the given problem.

5.2.1 Injectivity of F

After only 5 min. (and 6 interactions) of proving we reached the unprovable goal:

$$\text{F} \ \hat{}\ \text{breg} = \text{F} \ \hat{}\ \perp \rightarrow \text{breg} = \perp \tag{13}$$

This formula holds (compare Fig. 12), but how to deduce it? A short look at the visualized proof tree shows that this proof situation arose by trying to guarantee that in the backtracking case ASM2 stops (with failure) if *and only if* ASM1 stops! The "if" direction is trivial but for the "only if" direction we must prove (13).

What we need is the *injectivity* of F, and although that seems obvious (see Fig. 12.), we have to add it explicitly to INV:

11 F ˆ n = F ˆ n_1 $\rightarrow$ n = n_1

Thereby, on the one hand, we make it available for all proof situations. On the other hand it is now necessary to prove injectivity itself inductively!

5.2.2 Characterization of the Stack

Unfortunately, this was too rough. The proof attempt fails with a goal where injectivity of F + *(new(s')/currnode)* is asserted. We are not able to guarantee that the select rule preserves injectivity of F. It can not be proved because it is not true! Fig. 13. shows a situation where two different nodes of the (ASM2) stack are mapped to the same node of the (ASM1) tree.

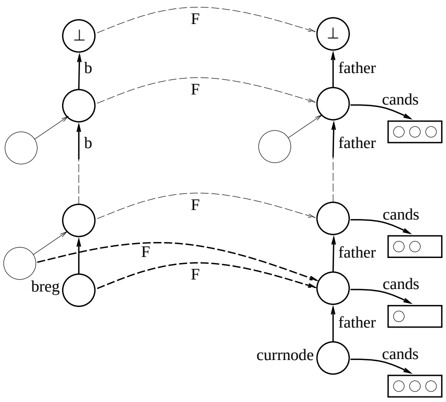


Fig. 13.

The problem arises because there are abandoned nodes that are no longer in the stack (i.e. reachable following the function *b* up from *breg*) but still present in the universe of nodes. The function F is still defined on such nodes, violating injectivity. But in the restricted context of stack nodes injectivity holds. (These reachable nodes are really what is meant to be the stack.) What we need is a logical characterization of the stack (i.e. of reachability). Then we can restrict injectivity as well as the other properties of F to the stack.

This restriction is also necessary to close another open goal in the same proof:

$$(\text{cands} + (\text{currnode} \;/\; \text{x}))\ \hat{}\ (\text{F}\ \hat{}\ \text{n}) = \text{cands}\ \hat{}\ (\text{F}\ \hat{}\ \text{n}) \tag{14}$$

This means that updating the dynamic function *cands* at *currnode* does not affect nodes in the range of F. Here we need:

12 $\text{F}\ \hat{}\ \text{n} \neq \text{currnode}$

But this is not true in general, as can be seen in Fig. 14, a snapshot of a situation after backtracking. What is true is that *currnode* is not in the range of the stack under F.

A correct approach to characterize the list of stack nodes is to define a program B-LIST# (its termination characterizes structures without cycles):

B-LIST#(n, b; **var** stack)
begin
if n = ⊥ **then** stack := nil **else**
  begin B-LIST#(b ˆ n, b; stack); stack := cons(n, stack) **end**
end

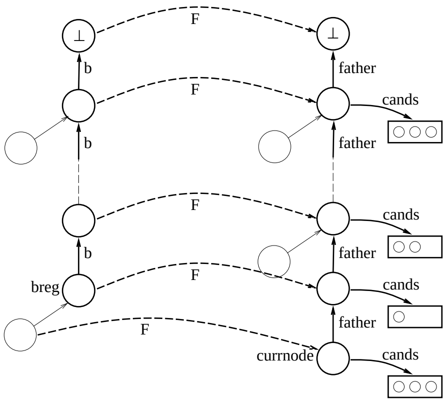


Fig. 14.

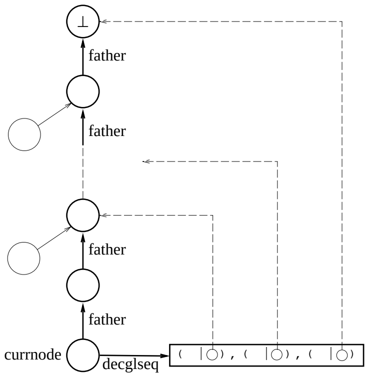


Fig. 15.

Now let ψ(n) be the conjunction of all subformulas, which depend on the selected node n (5 to 8 and 11) and φ the conjunction of the remaining subformulas (1 to 4, 9, 10 and 12). Then the coupling invariant INV gets the form:

$$\exists \text{ F: } \varphi \wedge \langle \text{B-LIST\#(breg, b; stack)} \rangle \ (\forall \text{ n. n} \in \text{stack} \rightarrow \psi(\text{n})) \tag{15}$$

This means that (for a suitable F) φ holds and that B-LIST# terminates with output *stack*, such that ψ holds for all elements of the *stack*.

5.2.3 Cutpoints

Proving equivalence between the two cut rules with this version of INV shows another difficulty: ψ must be guaranteed for the new stack shortened by execution of the cut. This stack is given by B-LIST# applied to the new *breg*, which is the first cutpoint of the current decorated goal sequence. Now, of course, the new stack would inherit ψ from the old one, if we knew that it is a part of the old one! But this is not deducible with the current INV. Here we need to assert that the cutpoints in the current decorated goal sequence are elements of the current stack. They may not point elsewhere. Therefore we define a new predicate called *cutptsin* and assert:

$$\text{decglseqreg cutptsin stack} \tag{16}$$

In the first version, the definition of *cutptsin* simply checked whether all cutpoints of the first argument are element of the second. Because the decorated goal sequence of every node in the stack can potentially become the decglseqreg (by backtracking), we also have to add

$$\text{(decglseq' \^{} n) cutptsin substack} \tag{17}$$

where substack is the output of ⟨B-LIST#(b ^ n, b; substack)⟩. With these additions the coupling invariant (15) changes to:

$$
\begin{array}{l}
\exists\ \text{F.}\quad \varphi \\
\qquad \wedge\ \langle\text{B-LIST\#(breg, b; stack)}\rangle \\
\qquad\qquad (\quad \text{decglseqreg cutptsin stack} \\
\qquad\qquad\ \ \wedge\ (\forall\ \text{n.}\quad \text{n} \in \text{stack} \\
\qquad\qquad\qquad\qquad \rightarrow\quad \psi(\text{n}) \\
\qquad\qquad\qquad\qquad\quad \wedge\ \langle\text{B-LIST\#(b \^{} n, b; substack)}\rangle \\
\qquad\qquad\qquad\qquad\qquad\quad \text{(decglseq' \^{} n) cutptsin substack))}
\end{array} \tag{18}
$$

Again, this is not strong enough. The proof fails because *cutptsin* so far does not care about any ordering. Executing the cut rule (which means that *breg* is changed to point to the first cutpoint of *decglseqreg*) shortens the stack like some pop operations would do (compare Fig. 5. in section 2.1). After that we have to prove that the (unchanged) cutpoints of decglseqreg are elements of that *shortened* stack. This holds only because the cutpoints point into the stack *in the right ordering* (see Fig. 15), so that *decglseqreg cutptsin stack* remains true with the new *breg*.

For this we have to strengthen the definition of *cutptsin*, leaving INV syntactically unchanged. In this special case no proof gets invalid (and this is checked by the correctness management of the KIV system!) because so far we used only lemmas about *cutptsin* that remain true in spite of the changed specification.

5.2.4 More Properties

The coupling invariant is still not complete. Several further proof attempts revealed the necessity to make some other tree properties explicit, which are only guaranteed by the rules, not by the data structure. Some of these properties are (informally): no candidate is in the range of *F*, no candidate list has duplicates, the intersection of different candidate lists is empty, and so on. Just to give a feeling for the complexity of the searched formula, the final coupling invariant is shown at the end of this section. Please recognize that all properties listed were actually needed to complete the proof (so it is not a arbitrary accumulation of properties!).

Summarizing, our general experience was that every time one finds INV to be insufficient and therefore adds new properties, this again causes unprovable goals. To discharge these new goals INV has to be improved again, leading to an evolutionary process of improving INV by verification attempts. We claim that for problems like the given one it is impossible to state all properties in a first proof attempt or to find them all in a pencil-and-paper proof. Therefore we use a proof system that offers good support for the evolutionary verification process sketched above.

$$
\begin{array}{l}
\exists\ \mathrm{F.}\quad \mathrm{stop} = \mathrm{stop'} \wedge \mathrm{mode} = \mathrm{mode'} \wedge \mathrm{vi} = \mathrm{vi'} \wedge \mathrm{subreg} = \mathrm{sub}\ \hat{}\ \mathrm{currnode} \\
\quad \wedge\ \mathrm{F}\ \hat{}\ \bot = \bot \wedge \mathrm{F}\ \hat{}\ \mathrm{breg} = \mathrm{father}\ \hat{}\ \mathrm{currnode} \wedge \bot \neq \mathrm{currnode} \\
\quad \wedge\ \mathrm{F}_d(\mathrm{F},\ \mathrm{decglseqreg}) = \mathrm{decglseq}\ \hat{}\ \mathrm{currnode} \\
\quad \wedge\ \bot \in \mathrm{s'} \wedge \bot \in \mathrm{s} \wedge \mathrm{currnode} \in \mathrm{s} \\
\quad \wedge\ (\quad \mathrm{mode} = \mathrm{select} \\
\qquad \rightarrow \qquad \mathrm{mapcl}(\mathrm{clls}(\mathrm{cllreg},\ \mathrm{db'}),\ \mathrm{db'}) \\
\qquad\qquad = \mathrm{mapcl}(\mathrm{map}(\mathrm{cll},\ \mathrm{cands}\ \hat{}\ \mathrm{currnode}),\ \mathrm{db}) \\
\qquad\quad \wedge\ \mathrm{every}(\mathrm{father},\ \mathrm{cands}\ \hat{}\ \mathrm{currnode},\ \mathrm{currnode}) \\
\qquad\quad \wedge\ \neg\ \mathrm{currnode} \in \mathrm{cands}\ \hat{}\ \mathrm{currnode} \wedge \neg\ \bot \in \mathrm{cands}\ \hat{}\ \mathrm{currnode} \\
\qquad\quad \wedge\ (\mathrm{cands}\ \hat{}\ \mathrm{currnode}) \subseteq \mathrm{s} \wedge \mathrm{nodups}(\mathrm{cands}\ \hat{}\ \mathrm{currnode})) \\
\quad \wedge\ \langle \mathrm{B\text{-}LIST\#}(\mathrm{breg},\ \mathrm{b};\ \mathrm{stack}) \rangle \\
\qquad\quad (\quad \mathrm{decglseqreg\ cutptsin\ stack} \wedge \mathrm{candsdisjoint}(\mathrm{F},\ \mathrm{cands},\ \mathrm{stack}) \\
\qquad\quad \wedge\ (\forall\ \mathrm{n}_1,\ \mathrm{n}_2.\ \mathrm{n}_1 \in \mathrm{stack} \wedge \mathrm{n}_2 \in \mathrm{stack} \wedge \mathrm{F}\ \hat{}\ \mathrm{n}_1 = \mathrm{F}\ \hat{}\ \mathrm{n}_2 \rightarrow \mathrm{n}_1 = \mathrm{n}_2) \\
\qquad\quad \wedge\ \mathrm{nocands}(\mathrm{F},\ \mathrm{cands},\ \mathrm{stack}) \wedge \mathrm{stack} \subseteq \mathrm{s'} \\
\qquad\quad \wedge\ \forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\quad \rightarrow \quad \mathrm{sub'}\ \hat{}\ \mathrm{n} = \mathrm{sub}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n}) \wedge \mathrm{F}\ \hat{}\ (\mathrm{b}\ \hat{}\ \mathrm{n}) = \mathrm{father}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n}) \\
\qquad\qquad\qquad \wedge\ \mathrm{F}_d(\mathrm{F},\ \mathrm{decglseq'}\ \hat{}\ \mathrm{n}) = \mathrm{decglseq}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n}) \\
\qquad\qquad\qquad \wedge\quad \mathrm{mapcl}(\mathrm{clls}(\mathrm{cll'}\ \hat{}\ \mathrm{n},\mathrm{db'}),\mathrm{db'}) \\
\qquad\qquad\qquad\quad = \mathrm{mapcl}(\mathrm{map}(\mathrm{cll},\ \mathrm{cands}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n})),\ \mathrm{db}) \\
\qquad\qquad\qquad \wedge\ \mathrm{every}(\mathrm{father},\ \mathrm{cands}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n}),\ \mathrm{F}\ \hat{}\ \mathrm{n}) \\
\qquad\qquad\qquad \wedge\ \mathrm{F}\ \hat{}\ \mathrm{n} \neq \mathrm{currnode} \wedge (\mathrm{F}\ \hat{}\ \mathrm{n}) \in \mathrm{s} \wedge \mathrm{nodups}(\mathrm{cands}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n})) \\
\qquad\qquad\qquad \wedge\ (\mathrm{cands}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n})) \subseteq \mathrm{s} \wedge \neg\ \mathrm{currnode} \in \mathrm{cands}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n}) \\
\qquad\qquad\qquad \wedge\ (\quad \mathrm{mode} = \mathrm{select} \\
\qquad\qquad\qquad\quad \rightarrow \quad \neg\ (\mathrm{F}\ \hat{}\ \mathrm{n}) \in \mathrm{cands}\ \hat{}\ \mathrm{currnode} \\
\qquad\qquad\qquad\qquad \wedge\ \mathrm{disjoint}(\mathrm{cands}\ \hat{}\ (\mathrm{F}\ \hat{}\ \mathrm{n}),\ \mathrm{cands}\ \hat{}\ \mathrm{currnode})) \\
\qquad\qquad\qquad \wedge\ \langle \mathrm{B\text{-}LIST\#}(\mathrm{b}\ \hat{}\ \mathrm{n}, \mathrm{b}; \mathrm{substack}) \rangle (\mathrm{decglseq'}\ \hat{}\ \mathrm{n})\ \mathrm{cutptsin\ substack})
\end{array}
$$

5.3 Statistics

All in all it took 12 proof attempts to reach a correct coupling invariant for the first refinement step. The verification work was done by the two authors in one month. In contrast, verification of the final correct version only took two days. 1416 proof steps were necessary, 378 of them were given interactively. The rest were found automatically by the heuristics of the KIV system. In addition, we needed about 300 first order lemmas. About half of these were already proved in the library, the other half was shown easily (in most cases, 0–2 interactions). One of the more complex is e.g.

$$
\begin{array}{l}
\qquad \mathrm{decglseqreg\ cutptsin\ stack} \wedge \bot \notin \mathrm{stack} \\
\quad \wedge\ \mathrm{n} \neq \bot \wedge \mathrm{n} \notin \mathrm{stack} \wedge \mathrm{F}\ \hat{}\ \bot = \bot \\
\rightarrow \mathrm{F}_d(\mathrm{F} + \mathrm{n}\ /\ \mathrm{n'},\ \mathrm{decglseqreg}) = \mathrm{F}_d(\mathrm{F},\ \mathrm{decglseqreg})
\end{array}
\tag{19}
$$

After the work on this refinement, the KIV system was improved from the experiences we learned (see [RSS97]). Most notable improvements were to the heuristics for unfolding procedures, for loops, and for quantifier instantiation. Also an additional heuristic for the elimination of selectors similar to the one in NQTHM ([BM79]) was added. Efficiency of rewriting was improved by using compiled discrimination nets (see [Kap87],[Gra96]).

With the improved system Harald Vogt, a student, who had previously learned about KIV only in a one term practical course, and did not have any prior knowledge of the WAM, redid the case study in 80 hours of work. This result gives an impression of the time it takes to learn to productively use the KIV system. The improvements of KIV saved about 1/3 of the necessary interactions (now 246).

6 Verification of Other Refinement Steps

This section summarizes our work on the verification of the refinement steps from ASM2 to ASM7 as described in [BR95], which we have proven correct so far. We will use the abbreviation *i/j* to mean the refinement from ASM*i* to ASM*j*.

Subsection 6.1 gives the optimizations made in the refinements from ASM2 to ASM4. The set of rules for the ASM4 is given, and the main verification problems are discussed.

Subsection 6.2 is concerned with the verification of the first proper compilation step from ASM4 to ASM5. Chronologically, verification of this refinement step was done before 2/3 and 3/4, since we wanted to find out if the different type of refinement would pose new problems. Verification of the first compilation step uncovered an error, which is present in the rules of all ASMs from ASM3 on.

Subsection 6.3 sketches some of the problems in verifying the full compilation of backtracking structure including switching instructions. [BR95] gives this refinement in two steps, introducing ASM6 and ASM7. Unfortunately, from the verification of 5/6 we learned that splitting verification into two parts complicates the work instead of simplifying it. Therefore the equivalence proof between ASM5 and ASM7 had to be given directly. This proof is the most complex proof we have done so far in this case study. The coupling invariant for it covers 3 pages and is 5 times the size of the coupling invariant for the first refinement as it was shown at the end of section 5.2.4. The complexity of the work makes it impossible for us to keep this section self-contained. Therefore we only sketch the main compilation idea and line out the main correctness arguments. For the rule set of ASM7, and to understand some of the technical remarks on the verifications problems we must refer the reader to [BR95].

The two main results of verifying 5/7 were the addition of backtracking to the switching instructions, and a precise definition of the *s-chain*-function which is used in the compiler assumption, given at the end of the section.

The final subsection 6.4 gives some statistics.

6.1 The Third and Fourth Interpreter: Optimizations

Although the second interpreter allocates fewer nodes than the first, there are still two possibilities for improvements, which are exploited in ASM3 and ASM4.

To see the first one, let us assume, that for some activator *act* ASM2 has to try several candidate clauses. When trying the first one, select rule will allocate a new node *temp*, and set the values *decglseq(temp)*, *sub(temp)* and *cll(temp)* of the new choicepoint.

If the first alternative does not lead to a solution, the interpreter will finally execute a *backtrack* instruction, which removes the node *temp* from the stack. Thereby the whole choicepoint will become inaccessible. The subsequent select rule for the second alternative will then push a new choicepoint on the stack, which is exactly the same as the one for the first alternative, except that *cll(temp)* has been incremented.

The optimization done in ASM3 avoids deallocation and reallocation of choicepoints. Instead it *reuses* the existing choicepoint. The optimization is achieved by replacing the removal of a choicepoint in the else-branch of backtracking with the assignment *mode := retry*, which activates a new rule, retry rule. This rule combines the effects of the else-branch of *backtrack* and *select*. It is executed instead of select rule for every alternative except the first. Its action is to remove a choicepoint (i.e. to set *breg* to *b(breg)*) only on execution of the last alternative. Otherwise it reuses the old choicepoint as required by incrementing *cll(breg)*.

The old select rule, which allocates a new choicepoint is now only called for the first alternative clause. It is therefore renamed to try rule. To avoid duplication of interpreter code, the common parts of try and retry rule (computing a most general unifier and applying it to *subreg*) are moved to a new enter rule.

The second place for improvement is addressed in interpreter 4. It is the allocation of choicepoints with empty lists of alternatives. Such a *useless* choicepoint is created e.g. for queries with just one alternative in the try rule of the third interpreter (resp. in select rule of the second, an example node is C in fig. 7). Such a choicepoint is useless, since it will immediately be discarded by backtracking when visited. Its creation can be avoided altogether by suitable *look ahead* guards in the try- and retry rule of ASM4.

With both optimizations, the set of rules for ASM4 looks as follows:

```
final success rule
    IF stop = run & decglseqreg = []
    THEN stop := halt
         subst := subreg

goal success rule
    IF stop = run & goal = []
    THEN decglseqreg := rest(decglseqreg)

call rule
    IF stop = run & is_user_defined(act) & mode = call
    THEN IF clause(procdef'(act,db')) = nil
    THEN backtrack
    ELSE cllreg := procdef'(act,db')
         ctreg := breg
         IF clause(procdef'(act,db')+1, db') ≠ nil
         THEN mode := try
         ELSE mode := enter

try rule
    IF stop = run & mode = try
    THEN mode := enter
         EXTEND STATE BY temp
         WITH
              breg := temp
              b(temp) := breg
              decglseq(temp) := decglseqreg
              sub(temp) := subreg
              cll(temp) := cllreg + 1
         ENDEXTEND
```

```
enter rule
      IF stop = 0 & mode = Enter
      THEN LET clause = rename(clause(cllreg),vi)
               LET mgu = unify(act, hd(clause))
               IF mgu = nil
               THEN backtrack
               ELSE  decglseqreg := apply(mgu,[<bdy(clause),ctreg> |cont])
                        subreg := subreg ∘ unify
                        vi := vi + 1
                        mode := Call

    where

backtrack ≡ IF breg = ⊥
                THEN stop := halt
                         subst := failure
                ELSE mode := retry

retry rule
      IF stop = run & mode = retry
      THEN decglseqreg := decglseq(breg)
               subreg := sub(breg)
               cllreg := cll(breg)
               ctreg := b(breg)
               mode := Enter
               IF clause(cll(breg) + 1) ≠ nil
               THEN cll(breg) := cll(breg) + 1
               ELSE breg := b(breg)

true rule
      IF stop = run & act = true
      THEN decglseqreg := cont

fail rule
      IF stop = run & act = fail
      THEN backtrack

cut rule
      IF stop = run & act = !
      THEN father := cutpt
               decglseqreg := cont
```

An additional register *ctreg* is now set in call and retry rule, to have the right *cutpt* available in the enter rule. Also some care has to be taken in call and try rule to make ASM4 work correctly in case only one or no alternative has to be tried for some query literal.

Verification of the refinements from ASM2 to ASM4 is easier (done in 3 weeks) than the verification of 1/2 and 4/5 done before, since coupling invariants sufficient for a correctness proof are much easier to find. Although some additional properties to the ones given in [BR95] are still necessary, (e.g. the nodes mentioned in them have to be restricted to stack nodes), they provide a good starting point.

The main problem in the equivalence proof of 2/3 is how to generalize the proof technique as described in sect. 4.3 to cases, where m rule applications of ASM2 correspond to n rule applications of ASM3 (here there are cases with m:n = 2:3, 1:2). A general solution, which guarantees that correctness and completeness can be shown in *one* proof, will be described elsewhere.

In the verification of the refinement from 3/4 we (like [BR95] too) use a coupling invariant, which relates computation states that have the same useful choicepoints with nonempty clause list.

Unfortunately, with this coupling invariant proof obligation PO3 from section 4.3 (goal 9), which states that both interpreters must terminate at the same time, is not provable. ASM4 may already have stopped with result *failure*, while ASM3 still has to backtrack from useless choicepoints. This situation of *nonsimultaneous termination* has to be considered carefully in the coupling invariant. An additional argument is needed to guarantee that for two states of ASM3 and ASM4 related by the coupling invariant, removing useless choicepoints by executing rules of ASM3 will keep the invariant and eventually lead to a state where *stop = stop'* holds again. A generalized version of PO3 then becomes provable with this argument.

6.2 The fifth Interpreter: Compilation of Backtracking Structure

The first three refinement steps can be viewed as an optimization of the first ASM, which do not change the representation of the Prolog program. In contrast, the refinement from ASM4 to ASM5 compiles the predicate structure of Prolog. For the first time instructions are introduced, which will also be present in the final WAM.

The general idea of the refinement step is to move control over the rule to be executed from the *mode*-Variable to the actual code. To do this, *cllreg* no longer points to the line of a clause, but to an address, where *instructions* are stored. *cllreg* becomes the program counter of a CPU, and is therefore renamed to *pcreg*. Similarly the clause line *cll(n)* stored in choicepoints becomes a code pointer *pc(n)*. Checks for the value of *mode* are replaced by checks on the type of the instruction stored at *pcreg*. Possible instructions may at this stage still be clauses (they are replaced by finer-grained instructions later on), but additionally control instructions are introduced.

As an example, the following clauses for a predicate `p` in a Prolog program

```
p(X)    :- body1.
p(f(X)) :- body2.
p(g(X)) :- body3.
p(g(X)) :- body4.
```
(20)

are translated to the code fragment (labels L1 – L4 are symbolic addresses):

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: retry_me_else(L4)
    p(g(X)) :- body3.
L4: trust_me
    p(g(X)) :- body4.
```
(21)

On a query `?- p(X)`, call rule of ASM5 (now called when *pcreg* is at a special *start* address) will set *pcreg* to address L1.

call rule
IF stop = 0 & is_user_defined(act) & pcreg = start
THEN ctreg := breg
IF clause(procdef$_5$(act,db$_5$)) = nil
THEN backtrack
ELSE pcreg := procdef$_5$(act,db$_5$)

with

backtrack ≡ IF breg = ⊥
THEN stop := halt;
subst := failure
ELSE pcreg := pc(breg)

Execution of the `try_me_else(L2)` instruction at address `L1` will have the same effect, as try rule of ASM4 had.

try_me rule
IF stop = run & code(pcreg) = try_me_else(N)
THEN EXTEND STATE BY temp
WITH
breg := temp
b(temp) := breg
decglseq(temp) := decglseqreg
sub(temp) := subreg
pc(temp) := N
ENDEXTEND
pcreg := pcreg + 1

The address for alternative clauses stored in the choicepoint is `L2`. Similarly, execution of a clause at `L2 + 1` or `L4 + 1` will execute a rule with the same effect as enter rule of ASM4.

enter rule

```
IF stop = run & is_user_defined(act) & is_clause(code(pcreg))
THEN LET clause = rename(clause(pcreg),vi)
     LET mgu = unify(act, hd(clause))
     IF mgu = nil
     THEN backtrack
     ELSE  decglseqreg := apply(mgu,[<bdy(clause),ctreg> |cont])
           subreg := subreg ∘ unify
           vi := vi + 1
           pcreg := start
```

When *pcreg* = `L3` or *pcreg* = `L5` rules will be executed that correspond to the then- and the else-branch of retry rule of ASM4:

retry_me rule

```
IF stop = run & code(pcreg) = retry_me_else(N)
THEN decglseqreg := decglseq(breg)
     subreg := sub(breg)
     ctreg := b(breg)
     pc(breg) := N
     pcreg := pcreg +1
```

trust_me rule

```
IF stop = run & code(pcreg) = trust_me
THEN decglseqreg := decglseq(breg)
     subreg := sub(breg)
     breg := b(breg)
     pcreg := pcreg +1
```

In general, the list of clauses for one predicate given in the original program is compiled to a code fragment stored in the memory of ASM5, which starts with a `try_me_else` instruction and consist of the list of clauses separated by `retry_me_else` instructions, except the last, which is separated by a `trust_me` instruction. Such a code fragment is called a *linear chain.* The requirement, that all code fragments must be linear chains is reflected in the compiler assumption for the refinement from interpreter 4 to 5. Formally we have:

$$\begin{aligned} &\text{mapcl(clls(procdef'(act,db'),db'),db')} \\ = \; &\text{l-chain}(\text{procdef}_5(\text{act},\text{db}_5),\text{db}_5) \end{aligned} \tag{22}$$

where *procdef'* and *db'* are the *procdef*-function and the Prolog-Program as used in the ASMs 2, 3 and 4. $procdef_5$ is the new *procdef*-function for ASM5 and db_5 is the compiled Prolog program. The partial function *l-chain* terminates, if the code fragment stored at $procdef_5(act,db_5)$ is a linear chain, and delivers the clauses contained in it.

Proving the refinement correct has about the same complexity as the verification of the first refinement. It was done by the second author in about a month. The most interesting point in the verification was, how the compiler assumption must be transferred to properties in the invariant (see [Ahr95]).

Verification uncovered two problems in [BR95]. The first problem was found during formalization of the *l-chain* function which is used in the compiler assumption. The formal definition of the chain function (p. 25, which also allows nested chains, see the remark in 6.3) obviously does not guarantee that `try_me_else`, `retry_me_else`, `trust_me` instructions in the compiled Prolog program are only used in that order, as is stated correctly in the text.

A more important problem was found in the rule system itself and was revealed during formal verification. We found that all ASMs starting from ASM3 introduced in [Section 1.3] contained an unintended indeterminism between two rules.

To see the problem, consider again the fail rule from ASM4, as it was shown in Sect. 6.1. The obvious *intention* of the rule is that retry rule should be executed afterwards.

Now it seems to be obvious that the only rule that is applicable at all after execution of the fail rule is indeed retry rule. But our correctness proofs revealed that fail rule does not invalidate its own guard, so it may be executed again, leading to an infinite loop. The rule system is therefore indeterministic (or following the terminology of [Gur95], inconsistent), and does no longer correctly implement a Prolog interpreter.

Although the error is easy to correct (the conjunct $mode = call$ must be added to the guard of fail rule), we think this is a typical error that is very difficult to find even by intensive inspection (and, of course, we *had* to inspect the code thoroughly before we could make an attempt to define a coupling invariant). A reader will always unconsciously resolve the indeterminism in the intended way. Nevertheless, an implementation is blind for intentions, and will possibly resolve the conflict in the wrong way (and ours did!).

6.3 The Sixth and Seventh Interpreter: Switching

Until ASM5, the problem how to select clauses whose head 'may unify' with an activator has been delayed by using an abstract (underspecified) *procdef*-function. The problem is now addressed in the two refinement steps from interpreter 5 to 7 which restrict the possible implementations of the *procdef*-function.

Two extremes have to be considered:

- A simple implementation, in which *procdef(act,db)* depends on the leading predicate symbol *pred(act)* only. This solution is inefficient, since it leads to a lot of (expensive) failed unification attempts (consider e.g. a collection of facts $\texttt{p(c}_1\texttt{), ..., p(c}_n\texttt{)}$ in a database).
- A complex solution, which selects only the clauses unifiable with *act*. Such a solution would require to encode the whole unification process already in the clause selection.

The solution adopted in the WAM is a compromise between these two extremes. It uses the simple *procdef*-function as described above and additionally *switching instructions* in the compiled code, which jump („switch") to relevant "blocks" of clauses according to leading function symbols of the subterms of *act*. E.g. for a predicate `p` with two arguments, successive clauses for `p` in which the second subterm of the head starts with the same function symbol `f` may be grouped together, and a switching instruction may try these clauses if and only if the activators second subterm has leading function symbol `f` too.

Before switching instructions can be introduced in ASM7, blocks of similar clauses have to be grouped together in ASM6 (otherwise backtracking and switching would not be compatible). This is done by allowing *nested chains* instead of linear chains in interpreter 6: At every position, where a *linear* chain contains a clause, a *nested* chain may again contain a (nested) chain. The clauses contained in a subchain then form a "block of similar clauses". Taking up again the example program of the previous section as given in (20) and (21), the last two clauses could be grouped together, resulting in a subchain starting at `L4`:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: trust_me                                                    (23)
L4: try_me_else(L5)
    p(g(X)) :- body3.
L5: trust_me
    p(g(X)) :- body4.
```

In Interpreter 7, arbitrary switching instructions can be put in front of chains or subchains.They come in three types: `switch_on_term(i,Lv,Lc,Ll,Ls)` jumps to address `Lv`, `Lc`, `Ll` or `Ls` depending on whether *arg(act,i)* (the ith argument of *act*) is a variable, a constant, a list or a structure. `switch_on_struct(i,N,T)` assumes a list of N triples (f,j,L) stored at address `T` in memory (or some other datastructure from which this information can be retrieved). It jumps to `L`, if *arg(act,i)* has leading function symbol f and arity j. Similarly a list of N pairs (c,L) at address `T` is assumed for `switch_on_const(i,N,T)`. The instruction jumps to `L` if *arg(act,i)* is the constant c. In our example we could e.g. add the following switching instructions at `L4`:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: trust_me
L4: switch_on_term(L7,failcode,failcode,L6)                     (24)
L6: switch_on_struct(1,1,T)
L7: try_me_else(L5)
    p(g(X)) :- body3.
L5: trust_me
    p(g(X)) :- body4.
```

Address T would contain a list with the single element (g,1,L7). *failcode* is a special *code*-value, which should point to the backtracking routine. This call had to be added to the interpreter code given in Appendix 2 of [BR95]. We note that the call to the backtracking routine is present in the rules of the WAM as given in [AK91] for switch_on_struct and switch_on_const, but only implicitly assumed for the switch_on_term instruction.

Let us remark that this code layout is not optimal (for a systematic treatment of the variants "one-level" and "two-level" switching see [AK91]). It is given only for demonstration purposes.

The compiler assumptions

$$\begin{aligned} &\text{l-chain}(\text{procdef}_5(\text{act},\text{db}_5),\text{db}_5) \\ = \ &\text{n-chain}(\text{procdef}_6(\text{act},\text{db}_6),\text{db}_6) \end{aligned} \tag{25}$$

and

$$\begin{aligned} &\text{n-chain}(\text{procdef}_6(\text{act},\text{db}_6),\text{db}_6) \\ = \ &\text{s-chain}(\text{act},\text{procdef}_7(\text{pred}(\text{act}),\text{db}_7),\text{db}_7) \end{aligned} \tag{26}$$

for the two refinement steps are similar to the one given in the previous section for 4/5. The partial function *n-chain* (*s-chain*) characterizes nested chains (with switching). Note that the compiler assumptions do not prescribe some concrete switching scheme (variants of the WAM use different ones), but only restrict the possible code layouts to reasonable ones.

At this point we like to make some remarks on small deviations from [BR95]:

- With the introduction of subchains executing *one* query-literal will no longer create (at most) *one* choicepoint. *ctreg* is no longer sufficient to have the right *cutpt* available in enter rule. Two solutions for this *restore_cutpoint* problem are indicated in [BR95]: Either retry and trust instructions have to be marked with the depth of the corresponding subchain (indicating the depth in the stack where the correct *cutpt* is to be found) or *ctreg* has to be saved and restored as part of choicepoints. We have adopted the latter solution, which according to [AK91] is the standard one.
- [BR95] does not distinguish between ASM5 and ASM6, since they have the same set of rules (except that ASM6 requires a solution to the restore_cutpoint problem) and differ only in the restriction on the layout of code (*l-chain* vs. *n-chain*). We have separated the two levels, since we wanted to study the problems, which arise due to the change of datastructures in the compilation from ASM4 to ASM5 in isolation, without interference of problems created by optimization steps. In retrospective this was a good decision, since ASM6 does not occur in the final verification work (see below).
- *pcreg* and *pc* are called *p*, partial functions *l-chain*, *n-chain* and *s-chain* are uniformly called *chain* in [BR95].

An informal argument for the equivalence of ASM5, ASM6 and ASM7 is that they all "execute the same candidate clauses". More precisely, they invoke the enter rule with the same activators *act*, and the same candidate clauses stored at *pcreg*. Unfortunately, this informal argument is not sufficient for a formal proof. The reason is that the clause considered in the next activation of enter rule may currently be stored in a choicepoint *anywhere* in the stack, due to intervening applications of the cut rule. Therefore we have to set up a relation between the choicepoints of the two stacks involved. The complexity of verification depends mainly on the complexity of this relation.

For the equivalence proof of 5/6 we have to map one stack node from ASM5 to a list of "similar" stack nodes from ASM6. This can be done by a dynamic function H (existentially quantified like function F in 1/2) in the coupling invariant. Appending the lists *H(n)* for every n from the stack of ASM5 gives the stack of ASM6.

The difficulty in stating the invariant correctly is to formalize "similarity" between node n and nodes *n'* $\in$ *H(n)*. The most delicate point in the definition must relate the clauses contained in the (end piece of a) linear chain reachable from the code pointer *pc(n)* to those reachable in the nested chains from every *pc(n')* for *n'* $\in$ *H(n)*. A precise description is very lengthy and complicated. Also the choicepoint stored in the register set *regs* := *decglseqreg,subreg,cllreg* of ASM5 may correspond to the register set *and additionally* some varying list *nl* of choicepoints of ASM6.

Altogether the invariant derived for 5/6 is about two times the size of the invariant shown at the end of section 5.2 for the first refinement step. The correct invariant was found in 2 weeks with 8 iterations.

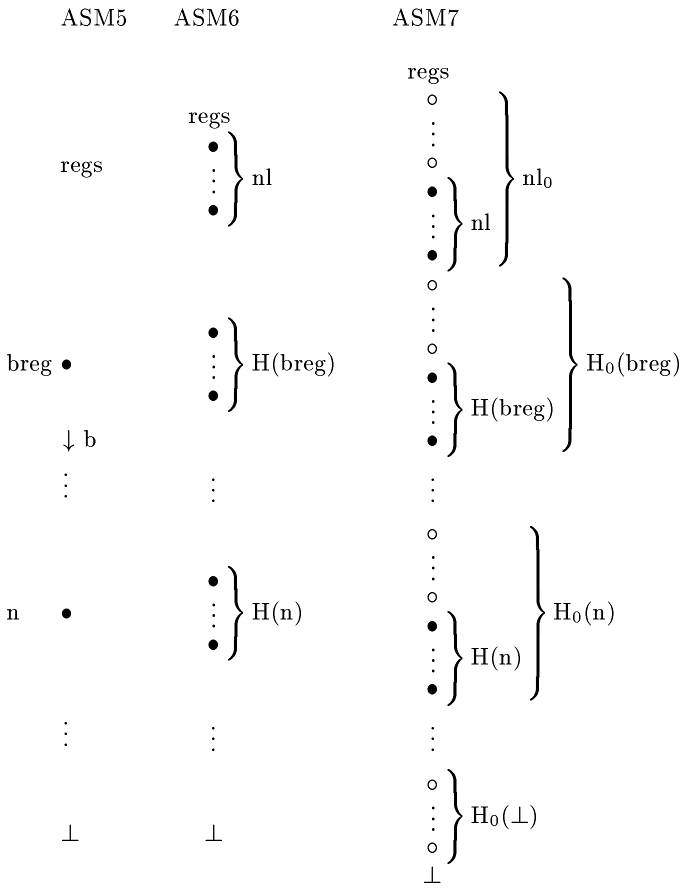


Fig. 16.

The equivalence proof of ASM6 and ASM7 would require to define a relation that maps a *varying* number of nodes of the stack of ASM6 to a *varying* number of nodes of the stack of ASM7. Also it would involve the two complex definitions of *n-chain* and *s-chain*. Therefore we decided that it should be simpler to verify the refinement from ASM5 to ASM7 directly, where like in the equivalence proof of 5/6 *one* choicepoint of interpreter 5 has to be related to a list of choicepoints of interpreter 7.

Still, the relation between the two stacks of ASM5 and ASM7 is much more complex than that between ASM5 and ASM6. The main reason is that in ASM7 there may be code fragments, which when executed with some specific query, call no clause at all, but fail in a switching statement. An example is the code fragment starting at `L3` in (24), when the activator is `p(f(X))`. Such code fragments give rise to what we call *empty* choicepoints with the same property ($cp \,\hat{}\, n =$ `L3`, $decglseq \,\hat{}\, n$ starts with `p(f(X))`). An empty choicepoint of ASM7 has no corresponding choicepoint in ASM5.

Fig. 16 compares the three stacks of ASM5, ASM6 and ASM7. Empty choicepoints are indicated with a '$\circ$'. Fig. 16 also shows that our coupling invariant for 5/7 uses an additional dynamic function H_0 and an additional list nl_0 to overcome the described problem.

Some further problems not present in the refinement from ASM5 to ASM6 are:

- Empty choicepoints on the bottom of the stack of ASM7 cause the problem of nonsimultaneous termination, as it was already found in 3/4.
- To formulate a coupling invariant, which holds *any* time during the computation requires a horrible number of cases and *must* be avoided with similar techniques than those required in 2/3.
- Termination arguments no longer work by arguing on the finite length of clause lists, but by induction on the recursion depth of the nested chain.

The above mentioned problems caused the first author to think about an initial coupling invariant for two weeks before he made a first proof attempt (typically, typing in a first guess of a coupling invariant was otherwise done in some hours). To derive the final coupling invariant 17 proof attempts were necessary in another 6 weeks of work.

In most of the proof attempts only the coupling invariant was modified. Several modifications had to be done to the *s-chain*-function which characterizes nested chains with switching.

Our final definition of the partial *s-chain*-function given in the notation of p. 28 in [BR95] is as follows (**abort** means nontermination, the arguments db_7 and *act* which are actually present in our recursive program have been dropped for better readability):

s-chain(Ptr) =
  if code(Ptr) = failcode then []
  else s-ch-rec(Ptr)

```
s-ch-rec(Ptr) =
  if is_clause(code(Ptr)) then [clause(code(Ptr))] else
  if code(Ptr) = try_me(N) then append(s-ch-rec(Ptr+),s-ch-retry_me(N)) else
  if code(Ptr) = try(N) then append(s-ch-rec(N), s-ch-retry(Ptr+)) else
  if code(Ptr) = switch_on_term(i,Lv,Lc,Ll,Ls)
  then if ¬ is_struct(act) ∨ arity(act) < i then abort else
         if is_var(arg(act,i)) then s-chain(Lv) else
         if is_const(arg(act,i)) then s-chain(Lc) else
         if is_list(arg(act,i)) then s-chain(Ll) else
         if is_struct(arg(act,i)) then s-chain(Ls) else abort
  else
  if code(Ptr) = switch_on_struct(i,N,T)
  then if ¬ is_struct(act) ∨ arity(act) < i then abort else
         if is_struct(arg(act,i))
         then s-chain(hash_s(T,N,funct(xi),arity(xi))) else abort
  else
  if code(Ptr) = switch_on_const(i,N,T)
  then if ¬ is_struct(act) ∨ arity(act) < i then abort else
         if is_const(arg(act,i)) then s-chain(hash_c(T,N,xi))
         else abort
  else abort
```

```
s-ch-retry_me(Ptr) =
  if code(Ptr) = retry_me(N)
  then append(s-ch-rec(Ptr+),s-ch-retry_me(N)) else
  if code(Ptr) = trust_me then s-ch-rec(Ptr+) else abort
```

```
s-ch-retry(Ptr) =
  if code(Ptr) = retry(N) then append(s-ch-rec(N),s-ch-retry(N)) else
  if code(Ptr) = trust(N) then s-ch-rec(N) else abort
```

Two changes were made to the actual interpreter compared to the one shown in [Appendix 2] of [BR95]. The first one was to add backtracking to the switching instructions as already mentioned. This was done before verification. The second was, that we replaced the test, whether *code(procdef(pred(act),db)) = nil* in call rule of ASM7 by *procdef(pred(act),db) = failcode*. Thereby the *nil*-instruction could be removed, which simplifies the *s-chain*-function.

The most complex proof ensures that backtracking of ASM7 preserves the coupling invariant. We have split this proof into three cases, one for the else-branch of backtracking and two for the then-branch. Still the more complex proof of the two then-cases (in which the node removed is the only element of the list *nl*) required 193 interactions and created a proof tree with 2504 nodes.

6.4 Statistics

The following table gives the number of proof steps, the number of interactively given proof steps and the number of theorems for the final version of the proof of each refinement. Also the time it took to specify and verify each refinement and the number of iterations needed to find the final invariant are given:

	1/2	2/3	3/4	4/5	5/6	5/7
Proof Steps	1475	4425	3988	2936	6190	20085
Interactions	246	450	580	237	666	1970
Theorems	16	32	31	32	53	72
Iterations	12	8	5	9	8	17
Verif. time	2 months	2 weeks	1 week	1 month	2 weeks	2 months

Since usually we do not keep old proof attempts, we cannot determine the total number of proof steps exactly. A rough estimate should be a quarter of the number of iterations that were needed multiplied by the number of final proof steps.

The size of the interpreters starts with 120 lines of (PASCAL-)code and reaches 240 lines for ASM7. The algebraic specifications of all datatypes used is composed of about 90 subspecifications with altogether 454 axioms. 580 first-order lemmas are used, 249 of them were from the library. The remaining 331 were proved with 441 interactions and 1462 proof steps.

7 Related Work

Work on compiler verification in general (or even more general: data refinement and other refinement relations) is so numerous that we will not even attempt to give an overview.

From the work on formal system-supported verification of compilers we exemplarily want to mention the work with NQTHM on the formal verification of a compiler for an imperative language ([Moo88], [You88]). This work is based on the notion of "interpreter equivalence" which is quite similar to our notion of equivalence of ASMs. It also contains a lot of references to related work.

Of specific work on the formal verification of a Prolog compiler we are aware only of the parallel work of C. Pusch in Munich. She also verified some refinement steps verified with the Isabelle system ([Pus96]).

The formalism used in Isabelle are inductively defined relations on the tuple of variables, which correspond to the semantics of our imperative programs as relations over their values. Pattern matching notation and polymorphism as used in functional languages allow to write the rules of an ASM in a more concise notation than our notation as PASCAL programs.

In contrast to our approach, which starts from a Prolog semantics based on search trees and tries to model the ASM approach as faithfully as possible, verification in Isabelle started from an operational Prolog semantics which is already based on stacks. With the knowledge of our invariant for the refinement from ASM1 to ASM2, and its complexity due to the use of pointer structures, stacks were modeled as simple lists, which allows to avoid our B-LIST#-procedure.

Instead of our refinement from ASM1 to ASM2, which has no counterpart in Isabelle, two other refinement steps were verified: In the first, the representation of cutpoints as (separate) stacks is refined to a representation, where cutpoints are positions in the main stack. In the second step, the initial *procdef*-function, which delivers all clauses is replaced by one which delivers only clauses with the same leading predicate symbol. Refinements 3 and 4 as verified in Isabelle are the same as our refinements from ASM2 to ASM4 (in Isabelle, Prolog-constructs *true* and *fail* were not considered, therefore the error we found in the interpreters 3 and 4 was not present in the case study).

The verification effort for the four refinement steps as given in [Pus96] was 6 person months and 3500 interactions. These numbers are about two times the numbers we got for the verification of the three refinements to reach ASM4. We suspect that this is largely due to the use of an asymmetric proof technique using proof maps, which requires two separate large proofs for correctness and completeness for each refinement step instead of one symmetric proof.

8 Conclusion

We have presented a framework for the formal verification of the Prolog to WAM compilation as given in [BR95]. The framework is based on the translation of sequential Abstract State Machines to imperative programs over algebraic specifications. With this translation correctness and completeness of the refinement between two ASMs is expressible as program equivalence in Dynamic Logic.

We introduced a proof technique, based on coupling invariants, which corresponds to the use of proof maps over ASMs. We have found that the correct coupling invariants, which are needed to show correctness and completeness of refinement steps, are far too complex to be stated correctly in a first attempt. The incremental development of a correct version takes much more time than the verification of the correct solution. Therefore, besides the pure power of the theorem prover, the 'proof engineering' support offered by the verification system (explicit proof trees, correctness management, reuse of proofs etc.) is crucial for the feasibility of the case study.

Verification showed that [BR95] is indeed an excellent analysis of the compilation problem from Prolog to WAM. Nevertheless an unintended indeterminism in one of the ASMs had to be removed (6.2), and minor corrections were also necessary on the formalization of the compiler assumptions. These results show, that to *guarantee* compiler correctness, mathematical analysis should be followed by formal verification.

Let us conclude with an outlook on the continuing work on this case study. The next two of the remaining 6 refinement steps are concerned with the compilation of single clauses ([Section 3] in [BR95]). Their correctness should be easy to show and require no new proof techniques. New problems will have to be overcome to verify that Prolog-Terms can be represented by pointer structures (the final [Section 4] in [BR95]). Finally it would remain to verify a compiler built on the basis of the compiler assumptions.

Although we are currently only about half the way from Prolog to the WAM, verification of the first levels has confirmed our belief that verification of the WAM is a feasible, but challenging task.

Acknowledgements

We thank our colleagues Wolfgang Reif, Kurt Stenzel and Matthias Ott for their valuable comments on drafts of this paper.

References

[Ahr95] Wolfgang Ahrendt. Von PROLOG zur WAM — Verifikation der Prozedurübersetzung mit KIV. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, December 1995.

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

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.

[BR94] Egon Börger and Dean Rosenzweig. A mathematical definition of full PROLOG. *Science of Computer Programming*, 1994.

[BR95] Egon Börger and Dean Rosenzweig. The WAM—definition and compiler correctness. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence.* North-Holland, Amsterdam, 1995.

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* Springer LNCS 130, 1982.

[Gra96] P. Graf. *Term Indexing.* Springer LNCS 1053, 1996.

[Gur95] M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[Kap87] S. Kaplan. A compiler for conditional term rewriting systems. In *2nd Conf. on Rewriting Techniques anf Applications. Proceedings.* Bordeaux, France, Springer LNCS 256, 1987.

[Moo88] J Moore. Piton: A Verified Assembly Level Language. Technical report 22, Computational Logic Inc., 1988. available at the URL: http://www.cli.com.

[Pus96] Cornelia Pusch. Verification of Compiler Correctness for the WAM. In *Proc. of the 1996 Intern. Conf. on Theorem Proving in Higher Order Logics*, Springer LNCS, 1996.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.

[RS95] W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In J. Köhler, editor, *Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs.* Montreal, Quebec, 1995.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Tenth Annual Conference on Computer Assurance*, IEEE press. NIST, Gaithersburg (MD), USA, 1995.

[RSS97] W. Reif, G. Schellhorn, and K. Stenzel. Proving System Correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*, LNCS. Townsville, Australia, Springer, 1997. to appear.

[Sch94] Peter H. Schmitt. Proving WAM compiler correctness. Interner Bericht 33/94, Universität Karlsruhe, Fakultät für Informatik, 1994.

[SW83] D. T. Sanella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Coll. on Foundations of Computation Theory*, Springer LNCS 158. Linköping, Sweden, 1983.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical note 309, Artificial Intelligence Center, SRI International, 1983.

[Wir90] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, 1990.

[You88] W. D. Young. A Verified Code Generator for a Subset of Gypsy. Technical report 33, Computational Logic Inc., 1988. available at the URL: http://www.cli.com.