GERHARD SCHELLHORN AND WOLFGANG AHRENDT

THE WAM CASE STUDY: VERIFYING COMPILER CORRECTNESS FOR PROLOG WITH KIV

1. INTRODUCTION

This chapter describes the first half of the formal, machine-supported verification of a Prolog compiler with the KIV system.

Our work is based on the mathematical analysis given in (Börger and Rosenzweig, 1995), where an operational semantics (an "interpreter") for Prolog is defined as an Abstract State Machine (ASM). This interpreter is then transformed in 12 systematic refinements to an ASM which executes machine code of the Warren Abstract Machine (WAM).

The goal of our case study was to formalize ASMs and the proof techniques given in (Börger and Rosenzweig, 1995), and to give machine-checked correctness proofs for the correctness of the refinements. So far we have verified the first 6 refinements, and we will give a detailed account on the problems we found in verification.

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 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 (Börger and Rosenzweig, 1995), 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 chapter is organized as follows: Sect. 2 introduces the formalism of Abstract State Machines (ASMs,(Gurevich, 1995)).

Sect. 3 shortly described 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.

Sect. 4 defines a general translation of 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.

Sect. 5 first introduces the Abstract State Machine (ASM) which is used to define an operational semantics of Prolog. We assume the reader to be familiar with the core of Prolog: clauses with *! (cut)* and *fail* (see e.g. (Sterling and Shapiro, 1986)). Then the 6 refinements towards the Warren Abstract Machine (WAM) we verified so far are given. 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. Sect. 5 closely follows (Börger and Rosenzweig, 1995). It deviates only in some minor notational issues, and will give a precise formalization of the *chain*-function to be used in one of the compiler assumptions.

In Sect. 6 we give an overview over the verification. Problems specific to the verification of each refinement will be discussed. The coupling invariant for the first compilation step (the 4th refinement) will be shown. This formula is very complex and we will describe how it was developed by iterated proof attempts with the KIV system.

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

2. ABSTRACT STATE MACHINES

A (Gurevich) Abstract State Machine (ASM, also known as ‘Evolving Algebra’) basically consists of a number of rules which are applied to transform an initial state to a final one. The possible states of an ASM are defined to be the class of algebras over a fixed first-order signature SIG.

A rule is given by its applicability test ε (a ground boolean expression over SIG) and a set of function updates. A function update is of the form

$$f(\mathrm{t}_1,\ldots,\mathrm{t}_\mathrm{n}) := t,$$

where f is a function symbol from SIG, $t_1,\ldots,t_n$ and t are ground terms. A rule is applicable in an algebra $\mathcal{A}$, if the applicability test holds ($\mathcal{A} \models \varepsilon$). Firing a rule executes all function updates in parallel resulting in a new algebra (a "successor state"). Execution of a function update like the one above changes the semantics of f at $(t_1,\ldots,t_n)$ to be t. The case n = 0 is admitted, and we will talk of updating a "0-ary function" f then, since to call f a "constant" would be rather misleading. The updates of a rule are always assumed to be consistent, i.e. no two updates of a rule change the same function at the same argument.

Repeated execution of rules results in traces of 'evolving' algebras ($\mathcal{A}_0$, $\mathcal{A}_1$, $\ldots$), where each $\mathcal{A}_{\mathrm{i}+1}$ is the result of firing an applicable rule in $\mathcal{A}_\mathrm{i}$. *Computations* are defined to be traces starting in an algebra taken from some predefined set of initial algebras. A computation is either infinite (non-terminating computation) or terminates in a state in which no further rule is applicable. The semantics of an ASM is defined to be the set of all computations.

There are several extensions of the basic formalism and the semantics e.g. for parallel execution of rules (see (Gurevich, 1995)). For our case study we will only need *sort updates* for many-sorted signatures,

extend s **by** c

This sort update extends the domain of sort s with a new element, which is assigned to the 0-ary function c. Sort updates can be used just like function updates in rules (in an unsorted setting, where sorts are boolean valued functions, a sort update can be viewed as a function update, see again (Gurevich, 1995)).

In our application, the definition of a Prolog interpreter, an initial algebra will contain a Prolog program and a query as the value of two predefined constants. Execution of the rules of the first ASM will build up Prolog search trees, as they can be found in many Prolog textbooks (e.g. (Sterling and Shapiro, 1986)). If computation terminates, the answer substitution (which may be *failure*) can be read off as the value of a 0-ary function.

Since Prolog interpreters are deterministic, at most one rule of the ASM should be applicable in every state, so there will be exactly one computation of the ASM for every initial state.

3. KIV

The KIV system ((Reif, 1995), (Reif et al., 1995), (Reif and Stenzel, 1995), (Reif et al., 1997)) 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. Details on syntax and semantics are given in Chapter I.3.13.

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 programs are written in a PASCAL-like notation, they are *abstract* programs. The operations used in them are those given in an algebraic specification, not the concrete ones available in PASCAL.

To reason about abstract programs, KIV uses Dynamic Logic (DL). DL 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 chapter. A formal definition of DL can be found in (Harel, 1984), (Goldblatt, 1982).

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

To deduce properties of specifications and to verify program modules, KIV offers an advanced interactive theorem prover. Details on this prover can again be found in Chapter I.3.13. 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, see (Reif and Stenzel, 1995).

4. From Abstract State Machines to Dynamic Logic

In this section we will give a translation of sequential Abstract State Machines 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.

The central idea for translating the dynamic part 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

extend s **with** c

corresponds to the two assignments

c := new(s); s := s $\cup$ {c}

in DL, where for function *new : set* $\rightarrow$ *node* the axiom: $\neg$ *new(s)* $\in$ *s* is given.

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. 1, which defines functions from domain *dom* to codomain *codom*.

```
generic specification
parameter sorts dom, codom;
target sorts dynfun;
      functions cf           : codom                  → dynfun;
                . ^ .        : dynfun × dom           → codom;
                f . + ( . / . ) : dynfun × dom × codom → dynfun;
      variables f : dynfun; x, y : dom; z : codom;
      axioms cf(z) ^ x = z,
             (f + (x / z)) ^ x = z,
             x ≠ y → (f + (x / z)) ^ y = f ^ y
end generic specification
```

Fig. 1: 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.

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 / t)*, which is defined by the last two axioms to be the appropriately modified function.

Finally note that we did not add an extensionality axiom

$$f = g \leftrightarrow \forall x.\ f \mathbin{\hat{}} x = g \mathbin{\hat{}} x$$

to the specification, in contrast to the usual methodology used in KIV to specify non-free data types. Such an axiom would have allowed us to deduce (higher order) equalities between functions like *f = f + (x / f ^ x)*, but such equations did not show up in verification.

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 study the domain will be pointers (addresses, locations), and the final implementation of the dynamic functions in the WAM will be a part of computer memory.

4.2. *Translation of Programs*

Given the translation of the static and dynamic part of the ASM, it remains to translate the set of rules to an imperative program. For simplicity of the following presentation, we will assume that the test, whether any ASM rule is applicable, is *stop* = *run* (this can always be achieved by requiring *stop* to be initialized with *run*, and by adding a suitable final rule, which sets *stop* to *halt*, if the conjunction of all other rule tests is false). Then the result of the translation is the procedure *ASM* shown in Fig. 2.

```
ASM(in; var out)          BODY(var x)
begin                     begin
var x := t in             if {test of rule_1} then {updates of rule_1} else
    while stop = run      ...
    do BODY(x)            if {test of rule_n} then {updates of rule_n}
end                       end
```

Fig. 2: ASM as imperative program

In our case study, the inputs *in* of the first ASM will be the Prolog program and the query, the output value will be the answer substitution. The variable declarations $\underline{x} = \underline{t}$ initialize the variables $\underline{x}$ the interpreter computes on with suitable values $\underline{t}$. They correspond to the definition of the initial algebra in the ASM. Application of rules is done in a while loop. The body has been put in a separate procedure *BODY* to have a suitable abbreviation for the formulas below. *BODY* is called with reference parameters $\underline{x}$, which are used as input and output. It consists of a case analysis, which selects an applicable rule and executes its updates.

4.3. *From Proof Maps to Coupling Invariants*

Correctness and completeness of the refinement of one ASM to another (ASM') is formalized in DL as the assertion of the following program equivalence between the two corresponding procedures *ASM* and *ASM'*.

$$\begin{aligned}&\text{CompAssum}(\underline{\text{in}},\underline{\text{in'}})\\ \rightarrow\ &(\langle \text{ASM}(\underline{\text{in}};\text{out})\rangle\, \text{out=val} \leftrightarrow \langle \text{ASM'}(\underline{\text{in'}};\text{out'})\rangle\, \text{out'=val})\end{aligned} \tag{1}$$

This formula states that if the two interpreters are given inputs related by the compiler assumption *CompAssum*, then the first interpreter *ASM* terminates, if and only if the second *ASM'* terminates with the same answer substitution. Variable *val* is used to store the common result of both interpreters (this variable is not modified by *ASM* and *ASM'*).

The compiler assumption relates the two inputs of the interpreter. For optimization steps, the relation is just identity ($\underline{in} = \underline{in'}$), for compilation steps, the assumption gives sufficient criteria for the correct compilation of the Prolog program. The notions of *Correctness* and *Completeness* from (Börger and Rosenzweig, 1995) directly correspond to the implication from right to left and from left to right.

Proof maps $\mathcal{F}$ are defined in (Börger and Rosenzweig, 1995) 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 both ASMs are related via $\mathcal{F}$
PO2: Fig. 3 commutes for every corresponding pair of rules R and R’ of the two interpreters (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.

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

Fig. 3.

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

Fig. 4.

This informal argument can be formalized in DL to prove (1)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, *ASM* and *ASM’* compute on, differently, say $\underline{x}$ and $\underline{x}$’, then the direct translation of a proof map is a function, which maps a tuple of values for $\underline{x}$’ to a tuple of values for $\underline{x}$. Since we found no need for the connection between $\underline{x}$ and $\underline{x}$’ to be a function, we allow it to be an arbitrary *relation*, which we describe by a (DL-)formula *INV($\underline{x}$,$\underline{x}$’)*, with free variables $\underline{x}$ and $\underline{x}$’. We call this formula a *coupling invariant*.

Induction over the number of rule applications is replaced by induction on the number of iterations of the while loop (for details of the formalization, see (Schellhorn and Ahrendt, 1997)), and it can be proved, that the following three goals are sufficient to prove goal (1).

$$\text{CompAssum}(\underline{\text{in}},\underline{\text{in}}') \rightarrow \text{INV}(\underline{\text{t}},\underline{\text{t}}') \tag{2}$$

$$\text{INV}(\underline{\text{x}},\underline{\text{x}}') \rightarrow \text{stop} = \text{stop}' \wedge \text{out} = \text{out}' \tag{3}$$

$$\begin{array}{l} \quad \text{INV}(\underline{\text{x}},\underline{\text{x}}') \wedge \text{stop}' = \text{run} \wedge \text{stop} = \text{run} \\ \rightarrow \langle \text{BODY}'(\underline{\text{x}}')\rangle\, \langle \text{BODY}(\underline{\text{x}})\rangle\, \text{INV}(\underline{\text{x}},\underline{\text{x}}') \end{array} \tag{4}$$

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 loops 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 (4)is provable. This last goal corresponds to PO3 and formalizes commutativity of Fig. 4. The proof of (4) splits into one case for each corresponding pair of rules.

5. The WAM Analysis of Börger and Rosenzweig

To make this chapter as self-contained as possible, the following section introduces some of the Abstract State Machines of our case study. We will closely follow [Sections 1, 2] from (Börger and Rosenzweig, 1995) 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 (see Sect. 6). We will use the abbreviation *i/j* to mean the refinement from ASM*i* to ASM*j*.

5.1. *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 by backtracking. 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*.

To handle the *cut*, an extension of the state representation is required. A *cut* updates 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. The resulting (decorated goal) sequence *decglseq* looks as follows

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

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

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

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,!.                   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.

The query `?- p.` is stored as the *decglseq* of node *A* in the initial search tree depicted in Fig. 6. The two nodes (labeled ⊥ 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* → *node*, indicated by the arrow in Fig. 6, so we have *father(A)* = ⊥ (the father of ⊥ is undefined). Root node ⊥ 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. Computed substitutions (attached to the nodes with the *sub* function) are not shown in the figures, since they do not matter in the example.

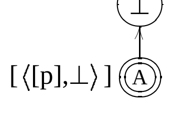


Fig. 6.

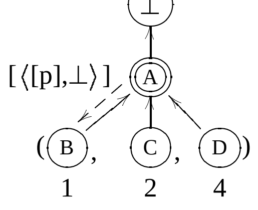


Fig. 7.

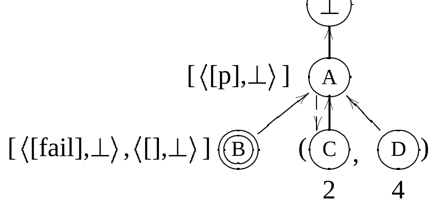


Fig. 8.

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.

In *call* mode, which is the initial mode, the candidate nodes are 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 $tmp_1$,...,$tmp_n$
   WITH father($tmp_i$) := currnode
         cll($tmp_i$) := $cll_i$
         cands := [$tmp_1$,...,$tmp_n$]
   ENDEXTEND
   mode := select

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

select rule

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

In this rule, the computed candidates for currnode, *cands(currnode)*, are abbreviated with *cands*, and we will use similar abbreviations for the functions *father*, *decglseq* and *sub* in the following rules.

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. 7.

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. 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. 8. 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 $\wedge$ 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 $[\langle[q,!],\perp\rangle,\langle[],\perp\rangle]$. 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. 9.

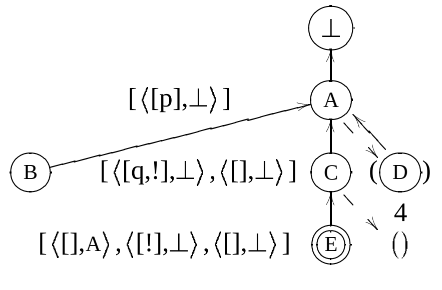


Fig. 9.

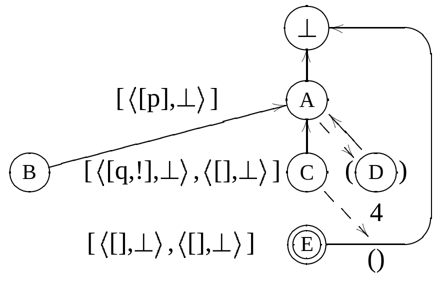


Fig. 10.

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

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

cut rule
IF stop = run $\wedge$ 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. 10). This action is the only purpose for decorating goals with nodes.

Finally, 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 $\wedge$ decglseq(currnode) = [] THEN stop := halt
                                                        subst := sub
```

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

5.2. *The Second Interpreter: Stacks of Choicepoints*

Here we summarize the first refinement of the ASM described above towards the Warren Abstract Machine (WAM), following (Börger and Rosenzweig, 1995), [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, instead of providing a list of candidate nodes, ASM2 attaches the *first* candidate directly via the *cll*-function. 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,!.         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

$$\begin{aligned}&\mathrm{mapcl(procdef(act,db),db)}\\ =\ &\mathrm{mapcl(clls(procdef'(act,db'),db'),db')}\end{aligned} \tag{5}$$

Here *clls* collects successive line numbers, until a *nil* is found and *mapcl* selects the clause at each line number. 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. The changed rules are

call rule

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

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

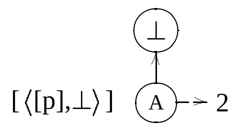
select rule

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

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

In our example, ASM2 goes through the states shown in Fig. 11 and 12, which correspond to those from Fig. 8 and 9 for the first ASM. Dashed arrows now point to the *cll* of a node. Since the former values attached to the *currnode* are now stored in registers, allocation of nodes corresponding to *B* and *D* is avoided. On the other hand, when node *A* is visited by backtracking (by executing the fail instruction in the state shown in Fig. 6), its choicepoint is moved to registers, and the following select instruction allocates the new, similar choicepoint *A'*.

In ASM2, 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 (here *A*). This causes one of the problems in the verification of the refinement of ASM1 to ASM2. The tuple of values *decglseq(n), sub(n), cll(n)* and *b(n)* attached to a stack node *n* is usually called a *choicepoint*.

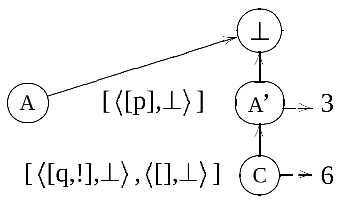


decglseqreg =

[⟨[fail],⊥⟩,⟨[],⊥⟩]

breg = A

Fig. 11.

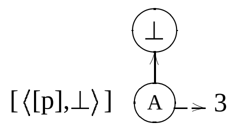


decglseqreg =

[⟨[],A'⟩,⟨[!],⊥⟩,⟨[],⊥⟩]

breg = C

Fig. 12.



decglseqreg =

[⟨[],A⟩,⟨[!],⊥⟩,⟨[],⊥⟩]

breg = A

Fig. 13.

5.3. *The Third and Fourth Interpreter: Optimizations*

Although the second interpreter allocates fewer nodes than the first, there are two possibilities for improvements, which are exploited in ASMs 3 and 4.

The first one is again visible in our example. When the first clause for activator p is tried, select rule allocates a new node *A*, and set the values *decglseq(A)*, *sub(A)* and *cll(A)* of the new choicepoint.

Since the first alternative does not lead to a solution, the interpreter executes a *backtrack* instruction, which removes the node *A* from the stack. Thereby the whole choicepoint becomes inaccessible. The subsequent select rule for the second alternative then pushes the new choicepoint *A'* on the stack, which is exactly the same as the one for the first alternative, except that *cll(A')* 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, is renamed to try rule. To avoid code duplication the common parts of try and retry rule 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 call- and retry rule of ASM4.

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

```
call rule
IF   stop = run ∧ mode = call
   ∧ is_user_defined(act)
THEN
    IF clause(procdef'(act,db')) = nil
    THEN backtrack
    ELSE
    cllreg := procdef'(act,db')
    ctreg := breg
       /* look ahead guard */
    IF clause(procdef'(act,db')+1, db')
       ≠ nil
    THEN mode := try
    ELSE mode := enter

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

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

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

try rule
IF stop = run ∧ mode = try
THEN
    mode := enter
    EXTEND node BY tmp
    WITH breg := tmp
           b(tmp) := breg
           decglseq(tmp) := decglseqreg
           sub(tmp) := subreg
           cll(tmp) := cllreg +1
    ENDEXTEND

retry rule
IF stop = run ∧ mode = retry
THEN
    decglseqreg := decglseq(breg)
    subreg := sub(breg)
    cllreg := cll(breg)
    ctreg := b(breg)
    mode := Enter
       /* look ahead guard */
    IF clause(cll(breg) +1) ≠ nil
    THEN cll(breg) := cll(breg) +1
    ELSE breg := b(breg)

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

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

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

An additional register *ctreg* is now set in call and retry rule, to have the right *cutpt* available in enter rule. By the two optimizations, the state of ASM4 corresponding to the one of ASM2 from Fig. 12 is now given by Fig. 13.

5.4. *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 basic idea behind this refinement step is to move control of rule selection from the *mode* variable to the compiled code. In addition to the clauses, ASM5 code contains instructions, which control the stack manipulation as the mode did before.

The former register for the current clause line, *cllreg*, is therefore renamed to a program counter *pcreg*. Consequently, the continuation addresses for backtracking, attached to each choicepoint *n* by *cll(n)*, are now represented by *pc(n)*. Selection of a clause at a clause line (function *clause*) becomes selection of the code (clause or instruction) at an address (function *code*). Checks for the value of *mode* are replaced by checks on the type of the code stored at *pcreg*. As an example, the following clauses for a predicate `p` in a Prolog program

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

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.
```

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

```
call rule
IF   stop = run
∧ is_user_defined(act)
∧ pcreg = start
THEN
    ctreg := breg
    IF code(procdef$_5$(act,db$_5$)) = nil
    THEN backtrack
    ELSE pcreg := procdef$_5$(act,db$_5$)

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

Execution of the `try_me_else(L2)` instruction at address `L1` has a similar effect as the try rule of ASM4 had. The continuation address for alternative clauses stored in *pc(n)* now is `L2`. By incrementing *pcreg* the next clause considered is the one at `L1 +1`. Similarly, execution of a clause at `L2 +1` or `L4 +1` executes a rule with the same effect as enter rule of ASM4.

```
try_me rule
IF   stop = run
   ∧    code(pcreg,db$_5$)
      = try_me_else(N)
THEN
    EXTEND node BY tmp
    WITH breg := tmp
             b(tmp) := breg
             decglseq(tmp) :=
                 decglseqreg
             sub(tmp) := subreg
             pc(tmp) := N
    ENDEXTEND
    pcreg := pcreg +1
```

```
enter rule
IF   stop = run
   ∧ is_clause(code(pcreg,db$_5$))
THEN
    LET clau = rename(code(pcreg),vi)
    LET mgu = unify(act, hd(clau))
    IF mgu = nil
    THEN backtrack
    ELSE
    decglseqreg :=
        apply(mgu,[<bdy(clau),ctreg> |cont])
    subreg := subreg ∘ mgu
    vi := vi +1
    pcreg := start
```

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

```
retry_me rule
IF   stop = run
   ∧ code(pcreg,db$_5$) = 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,db$_5$) = 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. The fragment 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`. 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 4/5. Formally we have

$$\begin{aligned}&\text{mapcl(clls(procdef'(act,db'),db'),db')}\\ &= \text{chain}(\text{procdef}_5(\text{act},\text{db}_5),\text{db}_5)\end{aligned} \tag{6}$$

where *procdef'* and *db'* are the *procdef*-function and the Prolog program from 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 *chain* terminates, if the code fragment stored at *procdef*$_5$*(act,db*$_5$*)* is a linear chain, and delivers the clauses contained in it. A look at (Börger and Rosenzweig, 1995) shows a problem with the formal definition[1]. It 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 in the text. A correct formalization of linear chains as a recursive program is (*cons* adds an element to the front of a list, instr $\equiv$ code(Ptr,db$_5$) and next_instr $\equiv$ code(Ptr +1,db$_5$))

```
chain(Ptr,db_5) =
   if instr = try_me_else(N) then chain-try-me(Ptr,db_5)
   if is_clause(instr) then [instr] else
   if instr = nil then [] else undef
chain-try-me(Ptr,db_5) =
   if instr = try_me_else(N) ∧ is_clause(next_instr)
   then cons(next_instr,chain-retry-me(N,db_5)) else undef
chain-retry-me(Ptr,db_5) =
   if instr = retry_me_else(N) ∧ is_clause(next_instr)
   then cons(next_instr,chain-retry-me(N,db_5)) else
   if instr = trust_me ∧ is_clause(next_instr)
   then [next_instr] else undef
```

Non-termination of this program can be caused either by a cyclic pointer structure or by an attempt to compute **undef**. Therefore termination of *chain* is required in compiler assumption (6) above.

5.5. *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 addressed in the two refinement steps from interpreter 5 to 7. ASM6 groups clauses together by introducing nested chains (at

[1] The definition is given for nested chains (our ASM6), while we consider only linear chains in ASM5, but the problem is the same.

positions, where linear chains contain clauses, a nested chain may contain another chain), and ASM7 introduces switching instructions. These restrict the possible implementations of the *procdef*-function. E.g. for an activator *act* = `p(X,g(Y))`, switching instructions may select the clauses with leading predicate symbol `p`, whose second argument is either a variable or starts with function symbol `g`. A detailed description of switching instructions is beyond the scope of this chapter, we refer the reader to (Börger and Rosenzweig, 1995) and (Aït-Kaci, 1991).

6. VERIFICATION

This section describes our work on the verification of the refinement steps from ASM1 to ASM7, which we have proven correct so far.

The following subsections will discuss the problems we found in the verification of each of the individual refinements. Before, let us give some impression on the verification process in general. As was discussed in Sect. 4, the critical point for a successful formal proof always is to find a coupling invariant INV($\underline{x}$,$\underline{x}$'), such that the diagram of Fig. 4 from Sect. 4.3 commutes for every corresponding pair of rules.

Verification of the refinements revealed, that it is impossible to state INV in a first proof attempt or to find all properties listed in INV in a pencil-and-paper proof. Depending on the complexity of the refinement step, between 6 and 17 iterations were needed to find an invariant, for which the proof goes through.

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. The good support KIV offers for this process (correctness management, the possibilities to inspect and modify proof trees and the reuse of proofs) were very important for minimizing the overhead of the iterations.

Sect. 6.1 gives some impression of the verification problems of the first refinement. Verification of the optimizations 2/3 and 3/4 is shortly discussed in Sect. 6.2.

Sect. 6.3 is concerned with the verification of the first proper compilation step 4/5. We will try to give an impression of the evolutionary development of the coupling invariant for this refinement, by stating the initial coupling invariant, and by indicating, what changes were necessary during iterated proof attempts. We will also show, how an error was uncovered, which is present in the rules of all ASMs from ASM3 on. The error was found during the verification of 4/5, since 4/5 was done before 2/3 and 3/4. The reason was that we wanted to find out if the different type of refinement (compilation, not optimization) would pose different new problems.

The equivalence proofs for the refinements from ASM5 to ASM7 were the most complex verification problems we have tackled so far in this case study. Verification uncovered some problems in the compiler assumptions as well as a missing backtracking-clause in the switching instructions. For more details on this verification the interested reader is referred to (Schellhorn and Ahrendt, 1997).

The final subsection 6.4 gives some statistics.

6.1. *From Trees to Stacks: Verifying the First Refinement*

Verification of the first refinement started from the 9 properties as given in (Börger and Rosenzweig, 1995), and a proof sketch given in (Schmitt, 1994), which added 3 more and made some minor corrections. These properties just formalize our description (see 5.2) of the changes that were made in the first refinement. Since ASM1 and ASM2 allocate different sets of nodes, a mapping F from nodes of ASM2 to corresponding nodes of ASM1 is used (which e.g. maps *breg* to *currnode*), just as in (Börger and Rosenzweig, 1995).

As already (Schmitt, 1994) pointed out, F cannot be given statically, but has to be defined by induction on the number of rule applications. Therefore we use an existentially quantified dynamic function for this purpose (note, that without our representation of dynamic functions as data structures, the quantification would not be first-order).

During verification we had to learn that the initially given properties were far from being sufficient for a formal proof. Another 20 properties had to be added in 12 iterated proof attempts. Some major ones were the characterization of the *stack* nodes of ASM2 (those nodes, which are reachable from *breg* via the *b*-function), the injectivity of the mapping F for those stack nodes, and a precise characterization of the search tree structure (all candidate nodes must be different and disjoint from the stack nodes, all cutpoints stored in decorated goal sequences must be stack nodes etc.).

A detailed discussion on how we found these new properties during proof attempts in two months of work is given in (Schellhorn and Ahrendt, 1997).

6.2. *Verifying the Optimizations*

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 (Börger and Rosenzweig, 1995) are necessary, (e.g. the nodes mentioned in them must be 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 (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 in (Schellhorn, 1998).

In the verification of the refinement from 3/4 we (like (Börger and Rosenzweig, 1995) too) use a coupling invariant, which relates computation states that have the same “useful” choicepoints with non-empty clause list.

Unfortunately, with this coupling invariant proof obligation PO3 from section 4.3 (goal (3)), 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 *non-simultaneous 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.

6.3. *Verifying Compilation of Backtracking Structure*

At a first glance, the refinement 4/5 seems to be easy to verify, since the mapping between rules is 1:1 (one rule of ASM4 corresponds to one of ASM5), and both interpreters allocate the same set of nodes, so no function *F* (like in 1/2) is required.

Nevertheless, proving the refinement of 4/5 poses some new problems, since this refinement is the first, in which a compiler assumption plays an important role. To demonstrate the complexity (we needed 9 iterations), we will try to give an impression of the development process necessary to find a correct coupling invariant (see also (Ahrendt, 1995)). 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 all hidden assumptions, which are necessary to ultimately *guarantee* the absence of errors in the refinement. And indeed, the attempt to verify the refinement revealed an error in one of rules shown in Sect. 5.4.

Let us start with our first attempt to define a coupling invariant. According to the description in Sect. 5.4, most of the corresponding registers from ASM4 and ASM5 should be identical. This gives the following properties (using the convention, that variables of ASM5 are written with a prime).

1. vi = vi’
2. stop = stop’
3. subreg = subreg’
4. breg = breg’
5. decglseqreg = decglseqreg’
6. ctreg = ctreg’
7. sub ^ n = sub’ ^ n
8. b ^ n = b’ ^ n
9. decglseq ^ n = decglseq’ ^ n

Application of dynamic functions is written with infix ^, the notation used in KIV (see Sect. 4.1). The node *n* in Properties 7–9 must be universally quantified over the set *ns* of currently allocated nodes, except the root node $\perp$, and we adopt this convention whenever a node *n* is mentioned in the following. The universal quantification causes a first problem. All relevant nodes must be guaranteed to be in this set (which is the same for both interpreters). We additionally need

10. breg $\in$ ns
11. ctreg $\in$ ns
12. decglseqreg $\in_{\text{ctp}}$ ns
13. decglseq ^ n $\in_{\text{ctp}}$ ns
14. b ^ n $\in$ ns
15. ns = ns’

where *dg* $\in_{\text{ctp}}$ *ns* is defined to mean “all cutpoints contained in *dg* (a decorated goal sequence) are contained in *ns* (a set of nodes)”.

Now it remains to define the relation between the clause lines stored in *cllreg* and *cll ^ n* in ASM4 and those stored in *pcreg* and *pc ^ n* in ASM5.

Starting with *cllreg* and *pcreg*, a comparison of the rule guards shows, that the *type* of the code, *pcreg* points to in ASM5, corresponds to the value of *mode* in ASM4. Therefore the definition will be by case distinction on the *mode*. Moreover, the invariant must guarantee that the *content* of the code(-sequent), *pcreg* points to, must always be equal to the clause(-list), *cllreg* points to.

mode = call: In this case, the content of *cllreg* is irrelevant (it will be overwritten by the call rule), and the value of *pcreg* is start.

16. mode = call $\rightarrow$ pcreg = start

mode = enter: Here, *pcreg* must point to a clause, in particular to the same as *cllreg* does.

17. mode = enter $\rightarrow$ clause(cllreg,db’) = code(pcreg,db_5)

mode = try: Two properties must be guaranteed. First, *pcreg* must point to a try_me_else instruction, and second the clauses contained in the following instructions must be the same as those following *cllreg*. The second property is formalized by

18. mode = try $\rightarrow$ mapcl(clls(cllreg,db’)) = chain-try-me(pcreg,db_5)

But note that the first property also follows from 18, since the definedness of chain-try-me implies that *pcreg* points to a try_me_else instruction. Actually, the check for the instruction at *pcreg* to be a try_me_else instruction was introduced just for this reason. It would not have been necessary, if chain-try-me were only used as an auxiliary function for chain in the compiler assumption, since chain contains the same check already. This is a general result for the definition of compiler assumptions. Design auxiliary procedures (or assumptions) in such a way that they are suitable to describe intermediate execution states of the ASM.

mode = retry: In this mode, *pcreg* has been set to *pc ^ breg'*:

19. mode = retry $\rightarrow$ pcreg = pc ^ breg'

This property is sufficient to ensure that *pcreg* points to a *retry_me_else* or *trust_me*-instruction, provided that we specify this property for all stack nodes *pc ^ n*:

20. mapcl(clls(cll ^ n,db')) = chain-retry-me(pc ^ n,db$_5$)

Again, definedness of *chain-retry-me* guarantees that *pc ^ n* indeed points to a retry_me_else instruction.

This completes the description of the initial invariant. What we have shown so far is the process of looking for a first version of an invariant, *before* starting the first proof attempt with the system. We demonstrated that doing this is not a trivial task at all. But, in our experience of verifying refinements, each hour invested in a good starting point can save days of verification time.

Now we like to describe the process of completing the invariant by iterated proof attempts with KIV until it suffices for the inductive proof. We give a rough overview of this *search* rather then describing the logical deduction, explaining how hidden assumptions were detected (if the proof needed them explicitly) and how proving these new formulas leads to new gaps an so on. We take this proof-historical point of view to emphasize the evolutionary nature of solving the given problem.

With the current invariant we tried to prove that the diagram shown in Fig. 4 (Sect. 4.3) commutes for each corresponding pair of rules. The first case we tried was the one with the two cut rules (since this was one of the most problematic proofs in the verification of 1/2). After some minutes of proving, we arrived at the subgoal, in which property 19 had to be proved to hold after rule application. The property holds trivially, since the cut rule of ASM4 is always called in *call* mode, and does not change it. But our invariant was too weak to allow a proof of this property. Taking into account, that fail rule is similar, we added (note again the convention, that guards of rules, which mention *act*, implicitly contain the conjunct decglseqreg $\neq$ [] $\wedge$ goal $\neq$ [])

21. decglseqreg=[] $\vee$ goal=[] $\vee$ $\neg$ is_used_defined(act) $\rightarrow$ mode = call

to the coupling invariant. With the new invariant the case for the cut rules succeeds. The case for the retry rules revealed another problem caused by the formula we just introduced. In retry rule *decglseqreg* is loaded with *decglseq ^ breg* from the stack. If this decorated goal sequence were empty, we would have mode = retry, violating property 21. To resolve this problem, we have to ensure, that the activator of every decorated goal sequence stored on the stack always is user defined (we abbreviate the goal and the activator of decglseq ^ n with goal ^ n and act ^ n).

22. decglseq ^ n $\neq$ [] $\wedge$ goal ^ n $\neq$ [] $\wedge$ is_user_defined(act ^ n)

A further attempt to prove the case of retry rules uncovers a new problem. We cannot guarantee, that retry rule of ASM4 is executed with breg $\neq$ $\perp$ (which would load the registers with the undefined values at the root node). Therefore we strengthen property 19 to

19a. mode = retry $\rightarrow$ pcreg = pc ^ breg' $\wedge$ breg $\neq$ $\perp$

With this invariant, the proof for most of the cases goes through. One of the last rules we considered was the fail rule (assuming this proof should be one of the easy ones . . .). But a proof attempt finally gives the goal

$\neg$ (decglseqreg = [] $\vee$ goal = [] $\vee$ $\neg$ is_user_defined(act))

which is obviously not provable. Analysis of the proof branch shows, that this goal arose from trying to prove

decglseqreg=[] $\vee$ goal=[] $\vee$ $\neg$ is_user_defined(act)
$\rightarrow$ retry=call

which is itself an instance of property 20 after fail rule. This means, that our assumption, that fail rule would be called only in call mode (like cut rule) was wrong. But if it is wrong, then there is something wrong not only with our invariant, but also with fail rule!

To see the problem, look again at the fail rule from ASM4, as it was shown in Sect. 5.3. The obvious *intention* of the rule is that retry rule should be executed afterwards.

Now it seems to be obvious that the only rule applicable after execution of the fail rule is indeed retry rule. But our correctness proofs reveals 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 newer terminology of (Gurevich, 1995), 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!).

Correction of the fail rule still gives *act* = fail, and *mode* = retry after its application. Therefore property 21 must be changed to

21a. (decglseqreg=[] ∨ goal=[] ∨ act = !) → mode = call

Unfortunately, still one detail is incorrect in our coupling invariant. In the case of the proof, which considers the two try rules, we find that we cannot prove that the choicepoint pushed on the stack is user defined. We have to add

23. mode = try → is_user_defined(act)

Finally, the equivalence proof of 4/5 succeeds with the invariant

```
  vi = vi’ ∧ stop = stop’ ∧ subreg = subreg’ ∧ breg = breg’ ∧ ctreg = ctreg’
∧ decglseqreg = decglseqreg’ ∧ breg ∈ ns ∧ ctreg ∈ ns ∧ decglseqreg $\in_{ctp}$ ns
∧ ns = ns’ ∧ (mode = call → pcreg = start)
∧ (mode = enter → clause(cllreg,db’) = clause(pcreg))
∧ (    mode = try
   → is_user_defined(act) ∧ mapcl(clls(cllreg,db’)) = chain-try-me(pcreg,$db_5$))
∧ (mode = retry → pcreg = pc ^breg’∧ breg ≠ ⊥)
∧ ((decglseqreg = [] ∨ goal = [] ∨ act = !) → mode = call)
∧ (∀ n.    n ∈ ns ∧ n ≠ ⊥
        →    sub ^n = sub’ ^n ∧ b ^n = b’ ^n
           ∧ decglseq ^n = decglseq’ ^n ∧ b ^n ∈ ns
           ∧ decglseq ^n ≠ [] ∧ goal ^n ≠ []
           ∧ is_used_defined(act ^n) ∧ decglseq ^n $\in_{ctp}$ ns
           ∧ mapcl(clls(cll ^n,db’)) = chain-retry-me(pc ^n,$db_5$))
```

6.4. *Statistics*

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

The size of the interpreters starts with 120 lines of (PASCAL-)code and reaches 240 lines for ASM7. The algebraic specifications of all datatypes is composed of about 90 subspecifications with altogether 454 axioms. A first version of the specifications and the interpreters was written within a day. Only minor corrections were necessary. 580 first-order lemmas are used, 249 of them were from the library. The remaining 331 were added on demand during the proofs of the main equivalence theorems. Their proofs needed 441 interactions and 1462 proof steps.

	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
Size of INV	20	25	25	14	53	97

As the statistic shows, the proofs for 5/7 were extraordinary complex. *All* of the problems we encountered in the refinements of ASM1 to ASM5 are present in the equivalence proof of ASM5 and ASM7: stack nodes must be characterized (like in 1/2), rules do not correspond 1:1 (like in 2/3), the interpreters may terminate non-simultaneously (like in 3/4), and the compiler assumption must be carefully translated to assumptions about intermediate interpreter states (like in 4/5). Also the verification of 5/6 showed that splitting verification of 5/7 into two parts complicates the work instead of simplifying it. Therefore we constructed the equivalence proof between ASM5 and ASM7 directly.

The statistic also shows a drastic increase of productivity. The main reason (besides increasing familiarity with the topic) are improvements in the KIV system, which were done from the experience we had gained from verifying refinements 1/2 and 4/5. A lot of heuristics (most notably the ones for quantifier instantiation, unfolding recursive procedures and for loops) were improved, as well as the efficiency of the simplifier (see Chapter I.3.13).

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 ((Moore, 1988), (Young, 1988)). 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 with the Isabelle system (Pusch, 1996).

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. Stacks were modeled as simple lists, which allows to avoid the characterization of stack nodes.

Instead of our refinement from ASM1 to ASM2 two other refinement steps were verified. Refinements 3 and 4 as verified in Isabelle are the same as our refinements from ASM2 to ASM4. In Isabelle, the Prolog-construct *fail* was 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 (Pusch, 1996) was 7 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 (Börger and Rosenzweig, 1995). 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 (Börger and Rosenzweig, 1995) 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 (6.3), and minor corrections were 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 (Börger and Rosenzweig, 1995)). 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 (Börger and Rosenzweig, 1995)). 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 chapter.

REFERENCES

Ahrendt, W. (1995). Von PROLOG zur WAM — Verifikation der Prozedurübersetzung mit KIV. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe. (in German).

Aït-Kaci, H. (1991). *Warren's Abstract Machine. A Tutorial Reconstruction.* MIT Press.

Börger, E. and Rosenzweig, D. (1995). The WAM—definition and compiler correctness. In Beierle, C. and Plümer, L., editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, Amsterdam.

Goldblatt, R. (1982). *Axiomatising the Logic of Computer Programming*. Springer LNCS 130.

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

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

Moore, J. (1988). PITON: A Verified Assembly Level Language. Technical report 22, Computational Logic Inc. available at the URL: http://www.cli.com.

Pusch, C. (1996). Verification of Compiler Correctness for the WAM. In *Proc. of the 1996 Int. Conf. on Theorem Proving in Higher Order Logics*, LNCS 1125.

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

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

Reif, W., Schellhorn, G., and Stenzel, K. (1997). Proving System Correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*, pages 69 – 72. Townsville, Australia, Springer LNCS 1249.

Reif, W., Schellhorn, G., Stenzel, K., and Balser, M. (1998). Structured Specification and Interactive Proofs with KIV. In Bibel, W. and Schmitt, P., editors, *Automated Deduction — A Basis for Applications*, volume I, 3. Kluwer. to appear.

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

Schellhorn, G. (1998). A general proof technique for ASM refinements. to appear as a Technical Report of the University of Ulm.

Schellhorn, G. and Ahrendt, W. (1997). Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*. available at the URL: http://hyperg.iicm.tu-graz.ac.at/jucs/.

Schmitt, P. H. (1994). Proving WAM compiler correctness. Interner Bericht 33/94, Universität Karlsruhe, Fakultät für Informatik.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MIT Press.

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