Verification of a Prolog Compiler – First Steps with KIV*

Gerhard Schellhorn
Abt. Programmiermethodik
Universität Ulm, D-89069 Ulm, Germany
email: schellhorn@informatik.uni-ulm.de

Wolfgang Ahrendt
Inst. für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, D-76128 Karlsruhe, Germany
email: ahrendt@ira.uka.de

Abstract

This paper describes the first steps of the formal verification of a Prolog compiler with the KIV system. We build upon the mathematical definitions given by Börger and Rosenzweig in [BR95]. There an operational semantics of Prolog is defined using the formalism of Evolving Algebras, and then transformed in several systematic steps to the Warren Abstract Machine (WAM). To verify these transformation steps formally in KIV, a translation of deterministic Evolving Algebras to Dynamic Logic is defined, which may also be of general interest. With this translation, correctness of transformation steps becomes a problem of program equivalence in Dynamic Logic. We define a proof technique for verifying such problems, which corresponds to the use of proof maps in Evolving Algebras. Although the transformation steps are small enough for a mathematical analysis, this is not sufficient for a successful formal correctness proof. Such a proof requires to explicitly state a lot of facts, which were only implicitly assumed in the analysis. We will argue that these assumptions cannot be guessed in a first proof attempt, but have to be filled in incrementally. We report on our experience with this 'evolutionary' verification process for the first transformation step, and the support KIV offers to do such incremental correctness proofs.

1 Introduction

The Warren Abstract Machine (WAM, [War83]) today is *the* standard for the implementation of Prolog compilers. Recently, a mathematical analysis of Prolog semantics and compiler correctness has become available with the papers of Börger and Rosenzweig ([BR94], [BR95]).

Based on this analysis and the proof sketched in [Sch94], this paper reports on our first steps towards the formal, machine-checked verification of the development described in [BR95] with the KIV system. 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 between 1-2 person years — are the following:

- Although the necessary effort is quite large for a universitary project, we want to demonstrate that the currently available technology for formal software development *is* capable of handling the complexity of compiler verification.
- 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 Evolving Algebras (EA's), at least in the

*This research was partly sponsored by the German Research Foundation (DFG).

deterministic case. In particular, the proof technique of commuting diagrams of Proof Maps, used informally in [BR95], can be formalized in DL.

- Currently the proof techniques in KIV are tailored to the verification of hierarchical, modular software systems ([Rei93]). Compiler verification is of a different type: It focusses on the transformation of a program, not on hierarchical implementation of a specification. Therefore our goal is to find out how proof techniques (tactics, heuristics and proof engineering techniques), which were developed for the first type of software development, behave in this new application area.
- Finally, in our experience, many of the requirements a system for the development of correct software must cope with to be suited for practical applications are only found in ambitious case studies. Solving these requirements always leads to significant improvements in the verification system.

This paper is organized as follows: Section 2 gives an introduction to the semantics of Prolog, the first transformation step and the formalism of *Evolving Algebras* (EA). We assume the reader to be familiar with the basic constructs of Prolog (clauses, including the cut) and their (informal) semantics. The introductory section closely follows [BR95].

Section 3 gives an introduction to the formalism used in KIV, namely *Algebraic Specifications* and *Dynamic Logic* (DL). We will assume that the reader is familiar with the basic notions of first-order logic and algebraic specifications. In section 4 the EA's from [BR95] are translated to programs and specifications of DL. Section 5 describes, how the equivalence proofs of EA's using 'commuting diagrams of Proof Maps' are formalized in DL. We will show that correctness and completeness of a transformation step in the sense of [BR95] can be reduced to the development of a *coupling invariant*, which is a DL-formula that corresponds to *proof maps* in EA's.

In Section 6 we develop a coupling invariant for the first transformation step. As will be shown, this formula is extremely complex and can only be developped in several iterations.

Section 7 concludes with an outlook on the continuing work on this case study.

2 A Prolog Semantics based on Evolving Algebras

Informal introductions to Prolog usually describe the semantics of the programming language operationally with the help of a search tree (e.g. [SS86]). To formalize this operational approach, an interpreter must be given, which builds up (and later on reduces) a search tree. The input of the interpreter is a Prolog program and a query. In case the interpreter terminates, it will give an answer substitution (which may be the special value 'failure'). In [BR95] such an interpreter has been formalized for the core constructs of Prolog: Clauses including `!`, `true` and `fail`. This interpreter is then transformed in alltogether 12 systematic steps to an interpreter of WAM machine code, with the idea that the role of the final interpreter is taken over by a processor executing assembler instructions. Parallel to the transformation of the interpreter, the Prolog program and the query are compiled to machine instructions. On intermediate levels the input of the interpreter 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.

Transformation in several steps is necessary, since the interpreter for machine code works completely different from the interpreter realizing the operational semantics of Prolog. To show the equivalence between the interpreters in one step would just be infeasible.

Splitting the transformation into several steps is also helpful to get insight in the basic steps of compilation. With orthogonal transformation steps it becomes possible to see how the various components of the WAM fit together.

Many of the transformations are optimization steps, e.g. the first transformation introduces registers and changes the search tree to a stack-like data structure. The first proper compilation step is step 5, where the predicate structure of Prolog is compiled.

2.1 Evolving Algebras

All interpreters are given using the formalism of Evolving Algebras (EA's; for a detailed introduction see [Gur95]). Evolving Algebras (EA's) can be viewed as a general method to write 'pseudocode over abstract data' ([BR95], p. 4). In the case of the interpreter of the first level, the abstract data describe different states of the search tree during the interpretation of a Prolog program. These states are formalized to be first-order, partial algebras $\mathcal{A}$ (in the sense of [Wir90]) over a fixed many sorted signature[1] SIG. The domain of a sort `node` is used to describe the currently allocated nodes of the search tree. They are related via a function `father`, which gives for every node its father (and is undefined on the root of the tree). To make the algebras 'evolve', an interpreter (the pseudocode) is given by a set of rules, which change the state (i.e. the algebra) under consideration, by allocating new nodes and by modifying the meaning of the `father` function.

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

$$\texttt{f}(\texttt{t}_1,\ldots,\texttt{t}_n)\ \texttt{:=}\ \texttt{t}, \tag{1}$$

where `f` is a function (or constant, if n = 0) from SIG, $\texttt{t}_1,\ldots,\texttt{t}_n$ and `t` are ground terms. A sort update[2] is of the form

extend s **by** c, (2)

where `s` is a sort and `c` is a constant of this sort. A rule is applicable in an algebra $\mathcal{A} \in$ Alg(SIG), if ε holds in $\mathcal{A}$. Applying a rule means executing all updates *in parallel*. Execution of a function update in an algebra $\mathcal{A}$ changes (or sets, if it was undefined) the value of `f` at $(\texttt{t}_1,\ldots,\texttt{t}_n)$ to `t`. Execution of a sort update adds a new element to the universe of sort `s`, and assigns it to the constant `c`. Note that the term 'constants' for 0-ary functions is rather misleading here, since they may change their value in function updates. Therefore we will avoid to call 0-ary functions, which are modified by rules of an EA 'constants' in the future, and use the term 'program variables' instead (since we will only encounter ground terms in Evolving Algebras, there is no risk to confuse program variables with ordinary variables).

"Evolution" of algebras is then defined by repeated indeterministic selection of an applicable rule and parallel execution of its updates[3]. Choosing an initial Algebra $\mathcal{A}_0$, which contains the Prolog program bound to a predefined fixed constant `db` (the "database") and an initial search tree of one node (standing for the initial query), we get traces $(\mathcal{A}_0, \mathcal{A}_1, \ldots)$ of Algebras over SIG, representing the state changes of the search tree. Reflecting the fact that Prolog is a deterministic language, we have at most one applicable rule in every algebra, which means that every trace $(\mathcal{A}_0, \mathcal{A}_1, \ldots)$ is determined by the initial algebra $\mathcal{A}_0$. If it is finite, it will lead to a final state $\mathcal{A}_n$, where the answer substitution (which may be also be the special value `fail`) can be read off by inspecting the value of the program variable `subst`. In case of an infinite trace execution of the query does not terminate.

2.2 The first interpreter

We will now describe the search tree and the first interpreter in some more detail. To illustrate the work of the interpreter we use the following example program:

[1] The general framework does not use predefined sorts, but introduces them via characteristic functions. This gives some extra freedom, but the Prolog-to-WAM compiler does not use notions, which go beyond many sorted logic.

[2] The generalization **extend** s **by** $c_1, \ldots c_n$ **with** *updates* **endextend**, as defined in [BR95] can obviously be simulated.

[3] There are other execution models, which execute *all* applicable rules in parallel, but we will not consider them.

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

Line numbers are explicitly written in front of the Prolog clauses for explanatory purposes[4]. The program is stored in a database `db` (a constant) in the algebra the interpreter starts with. The query `?- p.` is stored in an initial search tree depicted in Fig. 1. The figure contains a tree with two nodes, labelled `r` and `a`. The tree structure is stored in a function `father : node` $\rightarrow$ `node`, indicated by the arrow in Fig. 1, so we have `father(a) = r`. Node `r` is the *r*oot node of the tree. It serves only as a marker when to finish search.

The actions of the interpreter always work on a selected node, the "current node" `currnode`. As indicated in Fig. 1 by the double circle around `a`, we initially have `currnode = a`.

The relevant information to do the search is attached to the nodes via three functions `decglseq` ("decorated goal sequence"), `cands` ("candidates") and `sub`[5] ("substitution"). The `decglseq` of a node basically contains all literals, which have to be resolved at this point. The first of these literals is called the "activator" `act`. To handle the cut instruction, the list of decorated goals is divided into sublists, each sublist corresponding to a part of some clause body or to the query. Each sublist is paired with a node, called "cutpoint". Initially, the `decglseq` of node `a` (shown to its left in Fig. 1) is a list containing one element, the pair $\langle$`(p),r`$\rangle$ of query and root node. The `decglseq` of the root node is undefined.

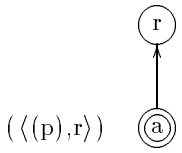

Fig. 1.

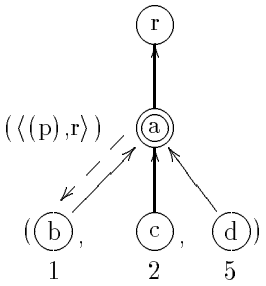

Fig. 2.

The second information used in the search is the `cands` function. This function is initially undefined on all nodes. It is used to store information about the clauses, which can be possibly used to solve the activator `act` in `decglseq(currnode)`.

The third function `sub` stores the answer substitution computed so far. Initially `sub(a)` is the empty substitution. It is not shown in the figures, since it does not matter in the example we consider.

Now the interpreter works in two modes, `call` mode and `select` mode. The mode is stored in a program variable `mode`, and there are different rules for each of the two modes.

In `call` mode, which is the initial mode, the `cands` information is computed. This is done by allocating a node for every clause (by expanding the universe of nodes), whose head may unify with the activator. This list of clause lines, which 'may unify' with a given literal `act` are specified as the result of a function `procdef`. Given a literal `act` and the Prolog program as stored in the database `db`, `procdef(act,db)` is assumed to return at least the lines of the clauses whose heads unify with `act`, and at most the ones which start with the same predicate as `act`.

[4] Our specification as well as the one in [BR95] uses an abstract sort `code` for clause lines. The use of natural numbers here is only to facilitate understanding.

[5] This function is called `s` in [BR95].

For the initial state, we have `act = p` and `procdef(p,db)` returns `(1,2,5)`, the lines of the three clauses with head `p`. Therefore, a list of three nodes `b,c,d` is allocated, and we have `cands(a) = (b,c,d)`. Fig. 2 indicates the `cands` list (of node `a`) with a dashed arrow to its first element and brackets around the elements. The clause line corresponding to the candidate node is attached to every candidate node via a function `cll`, i.e. we have `cll(b) = 1, cll(c) = 2, cll(d) = 5`, as shown by numbers below the candidate nodes in Fig. 2.

After the allocation of candidate nodes the interpreter switches to `select` mode. In this mode it selects the first candidate of `currnode` (here the node `b`). This is done by removing it from the `cands` list and making it the new `currnode`. Then the interpreter computes the decorated goal sequence `decglseq` for the new `currnode`, by removing the activator from the `decglseq` of the old `currnode` and replacing it with the body of clause at `cll(b)`. The `father` of the old `currnode` becomes the cutpoint for this clause body. Also the unifying substitution of activator and the clause head at `cll(b)` is applied to the new `decglseq` and stored in `sub(currnode)`[6]. Finally the interpreter switches back to `call` mode. The resulting search tree of this second step is shown in Fig. 3.

Now search continues again in `call` mode. Since now the activator is the special (always failing) predicate `fail`, the interpreter backtracks by setting `currnode := father(currnode)`, i.e. `currnode` is set to `a` again. Although this abandons node `b`, it will be kept in the search tree, since it 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 `(<(q,!,s),r>,<(),r>)`. Then, `call` mode with activator `q` allocates one new candidate node `e` for the clause `q.` in line 3, and selecting it the interpreter arrives at the state shown in Fig. 4.

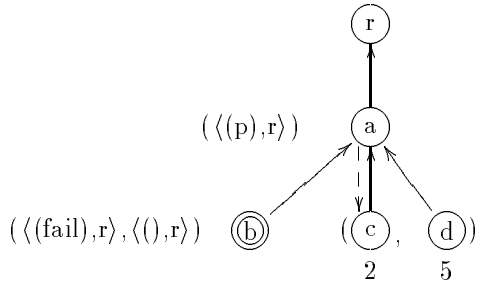

Fig. 3.

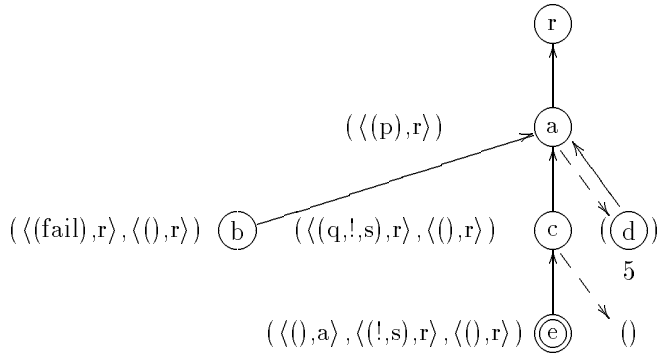

Fig. 4.

A 'goal success rule' now removes the empty body of the `q.` clause from the decorated goal sequence of `e` (together with cutpoint `a`), retaining `call` mode. Now the interpreter has to execute the cut instruction. According to the intuitive meaning of the cut, this should throw away all alternatives at nodes `a` and `c`. This could be done by setting `cands(a)` and `cands(c)` to the empty list (actually `cands(c)` is already empty), but there is an easier way, by updating `father(currnode)` to the cutpoint `r`, which is attached to the list of literals `(!,s)`. The result is shown in Fig. 5.

[6] Actually, unification is done with a copy of the clause with new variables. New variables are created with the help of a renaming index `vi`, which is incremented after each unification.

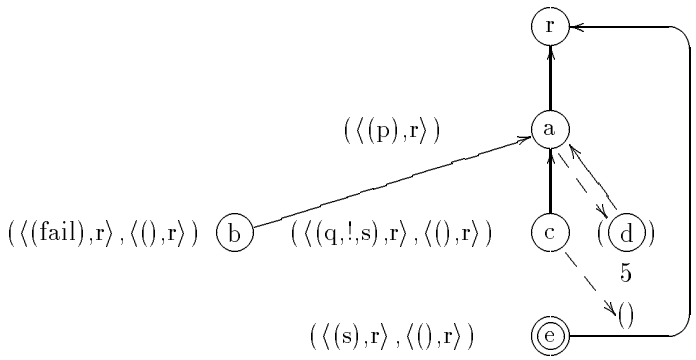

Fig. 5.

Finally, the interpreter allocates a node `f` for clause `s.`, selects it, and with two applications of goal success rule, `decglseq(f)` becomes empty. Since this means that we have completely solved the goal, the interpreter sets the answer substitution `subst` to `sub(currnode)` (here, of course, the empty substitution). Then it stops by modifying constant `stop` from `run` to `success`. Then no rule is any longer applicable, since all rule tests include the conjunct `stop = run`.

If we consider a variant of our example program, where clause `s.` is missing, the interpreter would also arrive at the situation shown in Fig. 5. But now an empty list of candidates would be allocated in `call` mode, and `select` mode, finding no more alternatives, would backtrack by setting `currnode := father(currnode)`. Since in this case `currnode` would become the root node `r`, the interpreter would stop by setting `subst:= fail; stop := failure`.

2.3 The second interpreter

With the move from the first to the second interpreter, we make a first step towards the Warren Abstract Machine (WAM). In this step, *registers* are introduced to store the currently important data, and the search tree of the first interpreter is transformed to a *stack* structure. In detail, the differences between the first and second interpreter are the following:

- sort `node` is renamed to `state` and function `father` is renamed to `b`. This change indicates that `b` now points *b*ackwards in a chain of nodes, which forms a stack. Note that in spite of the sort renaming, we will still call elements of sort `state` "nodes", to avoid confusion with the computation "state" (an algebra) of an interpreter.
- Instead of a list of candidate nodes, which all have a clause line attached by the `cll`-function, the second interpreter attaches the candidates directly via the `cll`-function. This is possible, if it is assumed that clauses, whose head start with the same predicate, are stored in successive clause lines, with a special marker (called `nil`) at the end. The representation of our example Prolog program for the second interpreter thus has to look like

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

A new `procdef'` function is needed, such that `procdef'(act,db)` now yields the first clause line, starting with `act`. So 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` (used as an axiom in correctness proofs):

$$\begin{aligned}&\text{mapclause(procdef(lit,db),db) =}\\&\text{mapclause(clls(procdef'(lit,compile(db)),compile(db)),compile(db))}\end{aligned} \tag{3}$$

Here `clls`[7] collects successive line numbers, until a `nil` is found, and `mapclause` selects the clauses at these line numbers. This assumption is weaker than the one given in [BR95], which identifies databases and requires

$$\text{procdef(lit,db) = clls(procdef'(lit,db),db)} \tag{4}$$

Here (to avoid a compilation step) it is assumed that clauses were already grouped according to different predicates on the first level. But even under this assumption, (4) can not be implemented for definitions of the `procdef` function, which are more specific than only looking at the leading predicate symbol. In this case even our liberalized compiler assumption requires code duplication (which does not matter, since the code is shared again in the compilation step, which introduces "switching instructions" described on pages 27ff in [BR95]).

Using the new `procdef'` function, instead of allocating a candidate list `cands(a) = (b,c,d)` in *call* mode, the second interpreter simply assigns `cll(a) = 1` (`= procdef'(p,db)`) in `call` mode. Incrementing `cll(a)` then corresponds to removing a candidate from `cands`. If the clause at `cll(a)` should become `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. With this change the nodes of the second interpreter, which may be visited in the future, always are the ones reachable from `breg` via the `b` function. They form a stack, but note that there may still abandoned nodes in the `state` universe, which are no longer reachable. As we will see in section 6, this causes problems for verification.

- The second interpreter provides register for keeping the data, which were attached to the `currnode` of the first interpreter. This allows to avoid the allocation of `currnode` altogether. Instead of `cll(currnode)`, `decglseq(currnode)`, `father(currnode)` and `sub(currnode` there are now registers `cllreg, decglseqreg,breg` and `subreg`[8].

By these changes, the situations corresponding to figures 3 and 4 on level 1 now become

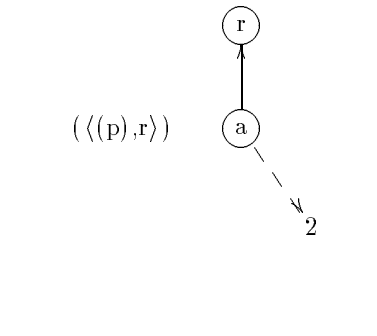

Fig. 6.

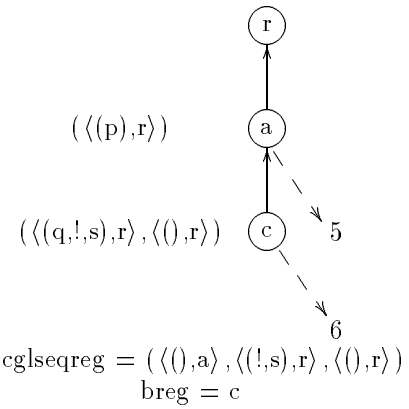

Fig. 7.

[7] Compared to [BR95] we have added an argument `db` to the function `clls` and some other (dereferencing) functions that retrieve a value stored at an address in the database.

[8] [BR95] uses overloading and calls the registers as well as the functions `cll`,`decglseq`,`b` and `s`.

In these diagrams the dashed arrows now point to the `cll` of the node, instead of his list of candidates. Since the `cllreg` does not matter in `call` mode (it would be the old value, which is now stored in `cll(breg)`), it is not shown in the figures.

Note that allocation of node `b` is avoided in the second interpreter, since the values, which were attached to this node in the first interpreter (when executing `p :- fail`), are now kept in the registers and are never pushed on the stack. Also candidate node `d` is not allocated in the second interpreter.

3 Dynamic Logic and Algebraic Specifications

The KIV system uses another formalism to describe 'pseudocode over abstract data': Imperative Programs over Algebraic Specifications. To prove properties over these programs, we use Dynamic Logic (DL,[Har79],[Gol82]). DL is an extension of (in our case many-sorted) first-order logic by formulas $\langle\alpha\rangle\psi$ (read "diamond α ψ") and $[\alpha]\psi$ ("box α ψ") , where α is an imperative program, and ψ is again a DL-formula. The intuitive meaning of the first formula is "α terminates and afterwards φ holds", the second means "if α terminates, then φ holds afterwards". The logic allows to state the total correctness of a program α with precondition φ and postcondition ψ as $\varphi \rightarrow \langle\alpha\rangle\psi$, and its partial correctness as $\varphi \rightarrow [\alpha]\psi$. Program inclusion with respect to some program variables $\underline{x}$ is also expressible as $\langle\alpha\rangle\underline{x} = \underline{x}_0 \rightarrow \langle\beta\rangle\underline{x} = \underline{x}_0$.

The imperative programs (written in a PASCAL-like notation) contain the usual imperative constructs: Assignment `x := t`, where the program variables are ordinary first-order variables (also parallel assignments $\underline{x} := \underline{t}$), conditional, compound, while-loops and recursive procedures with both value- and reference parameters. With random assignments `x := ?` there is also the possibility to write indeterministic programs.

The semantics of programs is given as a relation $[\![\alpha]\!]$ on valuations **v**. Valuations, as usual in first-order logic, assign values from an algebra $\mathcal{A}$ to the variables. For an algebra $\mathcal{A}$ and a valuation **v**, $\langle\alpha\rangle\psi$ holds (in short: $\mathcal{A}$, **v** $\models \langle\alpha\rangle\psi$) iff there is **v'** with **v** $[\![\alpha]\!]$ **v'**, such that $\mathcal{A}$, **v'** $\models \psi$ is true. $[\alpha]\varphi$ is equivalent to $\neg\ \langle\alpha\rangle\neg\ \varphi$.

The algebras, which are used to describe the possible values of variables, are specified by algebraic specifications. Algebraic specifications in KIV are built up from elementary specifications with the usual operations enrichment, union, renaming, parameterization and actualization. Elementary specifications are theories over Dynamic Logic (in most cases, we only use first-order axioms). Their (loose) semantics is the whole class of models. It can be restricted by generation principles (sometimes also called reachability constraints) of the form `S generated by F`[9], which assure that the sorts in `S` are generated by the constructors (constants or function symbols) in `F`.

4 From Evolving Algebras to Dynamic Logic

In this section we will give a translation of deterministic Evolving Algebras, as they are used in this case study, to Algebraic Specifications and Dynamic Logic. The translation is essentially one on one, because both EA and DL feature imperative programs, and 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 deterministic EA's. The translation is done in two steps: First, we will give a translation of the abstract data used (including the set of initial states) into an algebraic specification. In a second step we will translate the set of rules of an EA into an imperative program. The two steps are described in the following two subsections.

4.1 Translation of Specifications

To translate the abstract data types of an Evolving Algebra into an algebraic specification, we first have to separate the *static* and the *dynamic* part of the signature. The dynamic part contains those

[9] `S generated by F` can be expressed as an axiom in DL (see [Rei93]), but for convenience, specifications use this shortcut

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 EA's 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 EA contain definedness tests. This case does not occur in the Prolog-to-WAM-compiler (there *are* error elements, e.g. the result `fail` of the substitution function, 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 the Prolog-to-WAM-compiler, namely the functions `clls` ([BR95],p. 17), `F`, `G` (p. 23f) and `chain` (p. 25), which all collect a list of addresses by traversing some pointer structure. Although it is provable that the functions mentioned above all have a unique fixpoint, 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. With only minor changes (a case where the result is the empty list of pointers would do) the uniqueness of the fixpoint would be lost.

To fix this problem in partial first order logic requires an explicit characterization of the domain of the least fixpoint. 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. This avoids the use of an error element as well as an explicit characterization of the domain.

Data types, which are not completely specified in the EA, pose no problem for algebraic specification. E.g. on the first level nothing is said about the structure of terms. In terms of 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 essential idea here is to code dynamic functions and the carrier of dynamic sorts as the state (i.e. the value) of some (program) variables.

For dynamic functions, we have to separate the case of program variables (0-ary functions). These are simply translated to ordinary variables. Modification of a 0-ary function then becomes modification of a variable by an assignment in DL.

For dynamic functions f, the case with $n > 1$ arguments can be reduced to the case with one argument by adding an appropriate tuple-sort for the arguments (the Prolog-to-WAM-compiler uses only unary functions). For a unary function, we essentially have to code the (second-order) datatype of a *function* into a first-order datatype, with an explicit *apply*-operation. The modification operation of functions thereby becomes a first-order operation.

The resulting datatype is depicted in Fig. 8. It specifies functions from domain `dom` to codomain `codom` (both sorts are parameters to be actualized). The datatype contains constant functions `cf(z)` for every codomain-element `z`. Application of this function to any domain element `x` (with the `apply`-operation, for convenience written as an infix-circumflex '`^`') just gives `z`, as stated by the first axiom. Modification of function `f` at `x` by `z` is done with the mixfix-operation `f + (x / z)`.

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 (elements of sort `codearea`) in fact the final implementation will be a part of computer memory.

Every dynamic function then is turned into a program variable. Its sort is an instance (actualization) of the datatype `Dynfun` with the appropriate domain and codomain. For the initial state we usually use the constant function `cf(d)`, where `d` is a suitable "dummy"-element in the codomain. E.g. the `cands` function is initialized to `cf(nil)`, the function delivering an empty list of candidates for every node. A function update `f(x) := t` in the EA-formalism becomes an assignment `f := f + (x / t)` to variable `f`.

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

Fig. 8: Algebraic specification of dynamic functions

Finally note that we did not add an extensionality axiom

$$f = g \leftrightarrow \forall\, x.\ f\ \hat{}\ x = g\ \hat{}\ x \tag{5}$$

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 ^ x)`. Since such (higher-order) equalities are not expressible in the EA-formalism, we expected not to need them in the translated version either. And indeed, there was no need for equations between functions in verification.

The last problem we have to consider are dynamic sorts. We handle them by storing the current domain of a sort `so` in a variable `s`. The necessary generic specification `Set` is shown in Fig. 9.

In this specification, $\emptyset$ is the empty set, $\in$ tests for membership. `+s` and `-s` add resp. delete an element from a set. Equality on sets is characterized by an extensionality axiom. The `generated by` principle characterizes the sets to be finite. `new(s)` delivers a new element, which is not already in the set `s`. This operation is used to translate the sort update

extend so **with** c (6)

(where `so` is a sort and `c` a program variable of this sort in the EA) to the two assignments

$$c := \text{new}(s);\ s := s +_s c \tag{7}$$

where now `s` is a variable of sort `set` with elements `elem` actualized to `so`.

In the case of Prolog-to-WAM, all levels contain only dynamic sorts, which are initialized with a finite domain. E.g. for the first level it is initialized to {`root,currnode`}, where `root = new(`$\emptyset$`)` and `currnode = new(`$\emptyset$ `+s root)`. If the initial domain were not finite (we do not know of any case studies on EA's where an infinite initial domain is used), we would have to add a constant for this initial domain to the `Set` specification, and to include it in the `generated by` clause.

A somewhat unsatisfactory property of the `Set` specification is that actualizations of the parameter `elem` with a datatype with *finite* domain lead to an inconsistent specification, since we can

Set =
generic specification
parameter sorts elem;
target
sorts set
constants $\emptyset$: set;
functions
. +s . : set × elem → set;
. −s . : set × elem → set;
new : set → elem;
predicates
. ∈ . : elem × set;
variables s, s_1, s_2 : set; e, e_1, e_2 : elem;
axioms
set **generated by** $\emptyset$, +s;
¬ e ∈ $\emptyset$,
$e_1 \in s +s\ e_2 \leftrightarrow e_1 = e_2 \vee e_1 \in s$,
$e_1 \in s -s\ e_2 \leftrightarrow e_1 \neq e_2 \wedge e_1 \in s$,
$s_1 = s_2 \leftrightarrow (\forall\ e.\ e \in s_1 \leftrightarrow e \in s_2)$,
¬ new(s) ∈ s
end generic specification

Fig. 9: Algebraic specification of sets

deduce from the specification that the elements `new(∅), new(∅ +s new(∅)), new(∅ +s new(∅ +s new(∅))), ...` are all different. In the terminology of algebraic specification such a specification is said to lack the property of being *freely extendible* ([Rei93]). To regain this property, we could specialize the admissible parameters to be those with infinite domain, e.g. by adding a partial order < on `elem` together with an axiom $\forall\ e_1.\ \exists\ e_2.\ e_1 < e_2$.

Putting the translation of the static and dynamic part together we get the specification shown completely in appendix A. Many of the specifications (lists, pairs, etc.) could be retrieved from the library, together with a lot of simplification rules useful for verification. Some of the (admittedly large) size of the specification is due to the renaming of sort `node` to `state`, which causes a lot of duplicates. Also some of the specification length could be avoided by making overloading of operations available in KIV. Nevertheless a first version of the specification was written within some hours and needed only minor corrections.

4.2 Translation of Programs

Given the algebraic specifications used in our case study, and the translation of function and sort updates to assignments, we can now translate the rules of the interpreters to imperative programs. The main program realizing the first interpreter is procedure `EVAL1#` (by convention, procedure names end with a # in KIV) with the following structure (written in PASCAL-like notation):

EVAL1#(db, goal; var subst)
begin
var $\underline{x}$:= $\underline{t}$ **in**
while stop = run **do** BODY1#($\underline{x}$)
end

BODY1#(var $\underline{x}$)
begin
if { test of rule_1 } **then** { updates of rule_1 } **else**
if { test of rule_2 } **then** { updates of rule_2 } **else**
...
if { test of rule_n } **then** { updates of rule_n }
end

The inputs of `EVAL1#` are the database `db`, containing the Prolog program and the query `goal`. The reference parameter `subst` is used as the result value for the answer substitution. `EVAL1#` starts by initializing the program variables $\underline{x}$ = `stop, subst, decglseq, father, ...` with a vector $\underline{t}$ of suitable initial values. Then it enters a while loop with test `stop = run` and body `BODY1#`. An extra routine for the loop body is used simply to have a suitable abbreviation in the following formulas. `BODY1#` has the program variables $\underline{x}$ 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.

To structure the interpreter, there are subroutines for the different rules. The routine for the rule for `call` mode in the first interpreter is recursive, because it has to allocate a new node for every candidate of `currnode` (see sect. 2). Abbreviations are handled using variable declarations. Although DL allows parallel assignments, we transformed them into sequential ones, since we wanted to stay as close as possible to usual programming languages like PASCAL or C. In retrospective, this was not a very good idea, since it introduced the only real error (apart from typing errors) in selection rule of the first interpreter.

Translation of the rules of EA's somewhat increases their size, because of the expansion of abbreviations. The translated code of each interpreter is about 120 lines of PASCAL-Code. It is shown in appendices B and C.

5 Compiler Correctness as Program Equivalence

Correctness and completeness of the transformation of one interpreter into another is formalized in DL as the assertion that the following program equivalence holds:

$$\begin{aligned} & \langle \text{EVAL1\#(db,goal;subst)} \rangle \text{subst} = \text{subst}_0 \\ \leftrightarrow\ & \langle \text{EVAL2\#(compile(db),goal;subst)} \rangle \text{subst} = \text{subst}_0 \end{aligned} \tag{8}$$

Here `EVAL1#` and `EVAL2#` are the two interpreter programs and variable `subst` is their answer substitution (which may be the result `fail`). subst_0 is another variable, which is used to store the result value (this variable is not modified by `EVAL1#` and `EVAL2#`). The notions of *Correctness* and *Completeness* from ([BR95], p. 8) now directly correspond to the implication from right to left and from left to right.

The purpose of this section now will be to describe how the notion of a *proof map* $\mathcal{F}$ in evolving algebras translates to a formula of Dynamic Logic, which will be used in the formal correctness proof. In the context of Evolving Algebras, a *proof map* is defined to map algebras and rules of a 'concrete' level to algebras and rules of an 'abstract' level such that the following diagram 10 commutes for every rule R (cf. [BR95], p. 8):

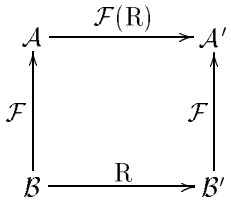

Fig. 10.

In the context of Dynamic Logic, the (dynamic parts of the) algebras involved in a computation have been replaced by the states of the vector of program variables. If we name the program variables, `EVAL1#` and `EVAL2#` compute on, differently, say $\underline{x}$ = [stop,subst,decglseq,vi,father,...] and $\underline{x}$' = [stop',subst',decglseq',vi',b,cll...], then the direct translation of a proof map would be a function, which would map 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($\underline{x}$,$\underline{x}$'), which involves the free variables $\underline{x}$ and $\underline{x}$'. We call this formula a *coupling invariant*. To use this formula in the proof, we split (8) into two goals, one for each direction of the implication. 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:

$$\begin{array}{l} \underline{x} = \underline{t} \wedge \underline{x}' = \underline{t}' \wedge \langle \textbf{while } \mathrm{stop}' = \mathrm{run} \textbf{ do } \mathrm{BODY2\#}(\underline{x}') \rangle \, \mathrm{subst}' = \mathrm{subst}_0 \\ \rightarrow \langle \textbf{while } \mathrm{stop} = \mathrm{run} \textbf{ do } \mathrm{BODY1\#}(\underline{x}) \rangle \, \mathrm{subst} = \mathrm{subst}_0 \end{array} \tag{9}$$

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

$$\langle \textbf{while } \varepsilon \textbf{ do } \alpha \rangle \, \varphi \leftrightarrow \exists \, \mathrm{i}. \; \langle \textbf{loop if } \varepsilon \textbf{ then } \alpha \textbf{ times } \mathrm{i} \rangle \, (\varphi \wedge \neg \, \varepsilon) \tag{10}$$

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:

$$\begin{array}{l} \langle \textbf{loop } \alpha \textbf{ times } 0 \rangle \, \varphi \leftrightarrow \varphi \\ \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i}+1 \rangle \, \varphi \leftrightarrow \langle \alpha \rangle \, \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i} \rangle \, \varphi \end{array} \tag{11}$$

The axiom (10) 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. Note that (assuming some fixed input) the number of iterations, we substitute for the quantified variable `i`, may not be the *exact* number of iterations, the while loop does. It can be any greater number, since iterations of the loop, with ε being false, do nothing.

Application of (10) on both while-loops we can then generalize our goal (9) using the coupling invariant, resulting in the following three goals:

$$\mathrm{INV}(\underline{t},\underline{t}') \tag{12}$$

$$\mathrm{INV}(\underline{x},\underline{x}') \wedge \mathrm{stop} \neq \mathrm{run} \rightarrow \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{subst} = \mathrm{subst}' \tag{13}$$

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \langle \textbf{loop if } \mathrm{stop} = \mathrm{run} \textbf{ then } \mathrm{BODY2\#}(\underline{x}') \textbf{ times } i \rangle \underline{x}' = \underline{x}'_0 \\ \rightarrow \exists\, j.\ \langle \textbf{loop if } \mathrm{stop}' = \mathrm{run} \textbf{ then } \mathrm{BODY1\#}(\underline{x}) \textbf{ times } j \rangle \mathrm{INV}(\underline{x},\underline{x}'_0) \end{array} \tag{14}$$

The first goal states that the coupling invariant holds before execution of the two while loops. The second goal says that from the coupling invariant and the fact that the first while loop stops, we must be able to infer that the second while loop also stops with the same answer substitution. These two goals are usually rather trivial, the complexity of verification is buried in finding an invariant INV such that the last goal (14) is provable. This last goal states that for every number `i` of rules the first interpreter executes there is a number `j` of rule applications of the second interpreter such that the Fig. 11 commutes.

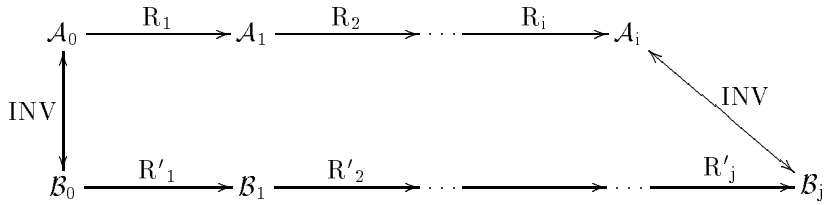

Fig. 11

It is proved by choosing `j = i` and induction on `i`. The induction step (the base case with i = 0 is trivial) reduces to showing:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \mathrm{stop}' = \mathrm{run} \wedge \langle \mathrm{BODY2\#}(\underline{x}') \rangle \underline{x}' = \underline{x}'_0 \\ \rightarrow \langle \textbf{if } \mathrm{stop} = \mathrm{run} \textbf{ then } \mathrm{BODY1\#}(\underline{x}) \rangle \mathrm{INV}(\underline{x},\underline{x}'_0) \end{array} \tag{15}$$

which now is the formalization of figure 10 (with INV replacing $\mathcal{F}$) in DL.

Having a closer look at goal (15), 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 (8) 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 (15) in the induction step:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \mathrm{stop} = \mathrm{run} \wedge \langle \mathrm{BODY1\#}(\underline{x}) \rangle \underline{x} = \underline{x}_0 \\ \rightarrow \langle \textbf{if } \mathrm{stop}' = \mathrm{run} \textbf{ then } \mathrm{BODY2\#}(\underline{x}') \rangle \mathrm{INV}(\underline{x}_0,\underline{x}') \end{array} \tag{16}$$

Both goals differ only in the way they treat termination of the two loop bodies. (15) claims that termination of `BODY2#` implies termination of `BODY1#`, (16) asserts the reverse implication. But since both loop bodies just apply one rule, they are flat programs[10]. To show their termination is trivial. Therefore, having proved (15), using it as a lemma in (16) will finish the proof immediately.

Due to this we now concentrate on the proof of (15). The goal is divided in as many subgoals as there are rules in the Evolving Algebra of interpreter2. The resulting seven cases corresponds to the tests of the rules $\mathrm{rule}'_1, \ldots \mathrm{rule}'_7$ in BODY#2. So for n = 1,2,...,7 we prove separately:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \mathrm{stop}' = \mathrm{run} \wedge \{ \text{ test of } \mathrm{rule}'_n \} \wedge \langle \mathrm{BODY2\#}(\underline{x}') \rangle \underline{x}' = \underline{x}'_0 \\ \rightarrow \langle \textbf{if } \mathrm{stop} = \mathrm{run} \textbf{ then } \mathrm{BODY1\#}(\underline{x}) \rangle \mathrm{INV}(\underline{x}'_0,\underline{x}') \end{array} \tag{17}$$

[10] except the recursive allocation of nodes in the `call` rule of the first interpreter

These seven lemmas are called query-success-21, goal-success-21, call-21, select-21, true-21, fail-21 and cut-21. Note that selection of the right case inside BODY2# and BODY1# is not done explicitly, but delayed to symbolic execution of the programs in the proof. Of course, case selection is trivial for BODY2# but important for BODY1#. To show that, when interpreter2 executes rule n, so does interpreter1, we inavoidably get the goal

$$\text{INV} \rightarrow (\{ \text{ test of rule'}_n \ \} \wedge \text{stop'} = \text{run} \rightarrow \{ \text{ test of rule}_n \ \} \wedge \text{stop} = \text{run}) \tag{18}$$

as one proof obligation.

6 Verification

Verification is done with the proof strategy of the KIV-System. This proof strategy works with a sequent calculus for DL (a sequent has the form $\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$ and is equivalent to $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \psi_1 \vee \ldots \vee \psi_m$). Goals (i.e. premises of a proof tree) are reduced to simpler ones by applying tactics (either manually or by built in heuristics), until we arrive at axioms. The verification strategy is based on induction and symbolic execution of programs. Symbolic execution is used to eliminate assignments, compounds, conditionals (where we get two subgoals), variable declarations, and non-recursive procedures. Induction is used for recursive programs and also for while loops. The proof strategy is described in [RSS95], so we will not go into the technical details of the tactics but only give the main intermediate steps of our proof.

6.1 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 INV(x,x'), such that goals (17) are provable. Some rough indication, how such an invariant might look like, is already given in ([BR95], p.17f). There an auxiliary function F: `state` $\rightarrow$ `node` is suggested, which maps the nodes of interpreter2 to the corresponding ones in interpreter1 (see Fig. 12.).

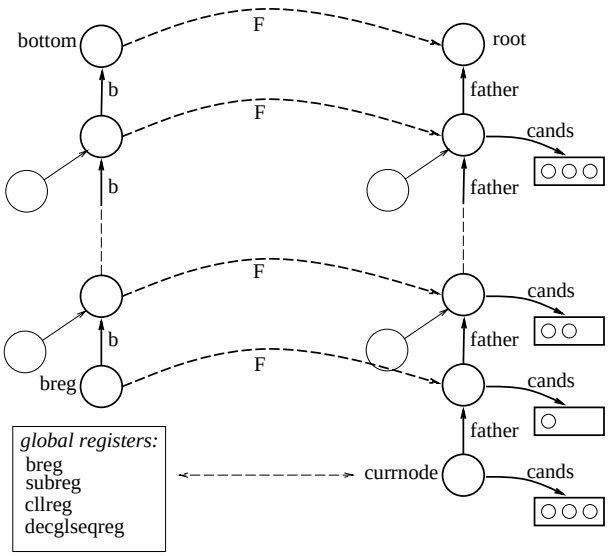

Fig. 12.

In the diagrams of section 2 this map is indicated by giving corresponding nodes the same label, but note that this was only done for clarity, the interpreters do not assign labels. Now the first problem we found with the definition of F is that it depends on the computation states

the two interpreters are in. A static definition would require to analyze the dynamic behavior of the two interpreters in allocating nodes. Of course this works only in the trivial case, where the interpreters allocate the same nodes synchronously, and F can be defined to be the identity. In our case it is not even possible, as can be seen from the injectivity problem for F discussed in subsection 6.2.2.

[Sch94] pointed out that F has to be defined by induction on the number of rule applications. That is, in terms of our EVAL# procedure mentioned above, induction on the number of loop iterations. So F is constructed by an inductive proof. The crucial question is about the formalism where a function can be updated not by an evolving algebra but by proof steps.

In our translated version a solution is easy: Simply let F be a *dynamic* function in the sense of section 4.1, 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 computation states. Based on this dynamic function the properties listed on p.17f of [BR95] translate to the following conjuncts in our invariant (remember that non-primed/primed variables refer to the first/second interpreter):

$\exists$ F:

1 decglseq ^ currnode = decglseqreg'

2 sub ^ currnode = subreg'

3a mapclause(map(cll, cands ^ currnode)) = mapclause(clls(cllreg'))

3b every(father,cands ^ currnode,currnode)

4 father ^ currnode = F ^ breg'

5 decglseq ^ (F ^ n) = decglseq' ^ n

6 sub ^ (F ^ n) = sub' ^ n

7a mapclause(map(cll, cands ^ (F ^ n))) = mapclause(clls(cll' ^ n))

7b every(father,cands ^ (F ^ n), (F ^ n))

8 father ^ (F ^ n) = father' ^ n

9 F ^ bottom = root

Here the predicate `every(fun, li, res)` is true iff for all elements `el` $\in$ `li` the equation `fun ^ el = res` holds. So `every(father,cands ^ currnode,currnode)` means that `currnode` is the `father` of all its `cands`. The formulas 3a+b, 7a+b are weakened versions of the equations `cands(currnode) = mk_cands(currnode, cll)` and `cands(F(n)) = mk_cands(F(n), cll(n))` given in [BR95], p.17f. This reflects the fact that in section 2.3 we used a (weakened) compiler assumption (3) instead of (4).

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

1 decglseq ^ currnode = fd(F, decglseqreg')

5 decglseq ^ (F ^ n) = fd(F, decglseq' ^ n)

where fd maps the first argument to the cutpoints of the second. In [Sch94] this was added to the coupling invariant together with the obvious but important equations:

10 stop = stop'

11 mode = mode'

12 vi = vi'

These formulas 1 – 12 formed the first version of the coupling invariant INV(x,x') when we began to prove the lemmas (17) with the system. In these proofs, INV(x,x') on the left side of the implication asserts the *existence* of an "F_{left}" before rule execution. The F appearing on the right hand side of the implication INV(x,x') ("F_{right}") has to be instantiated relative to F_{left}. Naturally F is left unchanged in most cases. Only in the proof of select-21 F_{right} is instantiated with F_{left} + (new(s')/currnode).

Up to here, INV concentrates on the dependencies between the two abstract machines (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 don't, as we will show below.

6.2 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 needs them explicitly) and how proving these new formulas leaves new gaps an so on. We take this proof-historical point of view to emphasize the evolutionary nature of solving the given problem.

6.2.1 Injectivity of F

After only 5 min. (and 6 interactions) of proving select-21 we reached the unprovable goal (abbreviated here):

$$F \hat{} \text{ bottom} = \text{root}, F \hat{} \text{ breg'} = \text{root}, ... \vdash \text{breg'} = \text{bottom}, ... \tag{19}$$

This formula holds (see Fig. 12.), but how to deduce it? A short look at the corresponding branch in the visualized proof tree shows that this proof situation arose by trying to guarantee that in the backtracking case interpreter2 stops (with failure) if *and only if* interpreter1 stops! The first direction is trivial because

$$\text{breg'} = \text{bottom} \rightarrow \text{father} \hat{} \text{ currnode} = \text{root} \tag{20}$$

follows from part 4 and 9 of INV. But the other direction

$$\text{father} \hat{} \text{ currnode} = \text{root} \rightarrow \text{breg'} = \text{bottom} \tag{21}$$

which is with 4 and 9 equivalent to

$$F \hat{} \text{ breg'} = F \hat{} \text{ bottom} \rightarrow \text{breg'} = \text{bottom} \tag{22}$$

cannot be deduced. But it would if *injectivity* of F were available. Although that seems to be obvious, (see Fig. 12.) we have to add the injectivity of F explicitly to INV:

13 $F \hat{} n = F \hat{} n_1 \rightarrow n = n_1$

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

6.2.2 Characterization of the Stack

Unfortunately, this was too rough. The attempt on proving select-21 fails with a goal where injectivity of `F + (new(s')/currnode)` is asserted. In other words, we are not able to guarantee that the select rule preserves the injectivity of F. It can not be proved because it is not true!

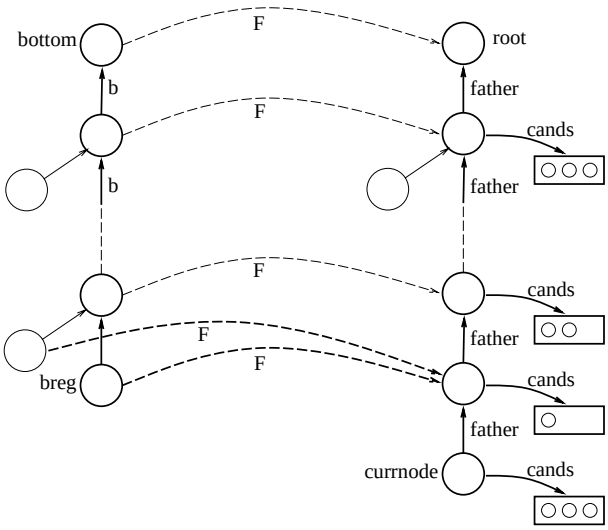

Fig. 13.

Fig. 13. shows a situation where two different nodes of the interpreter2 tree are mapped to the same node of the interpreter1 tree.

The problem arises because of the abandoned nodes that are no longer 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 the injectivity. But in the restricted context of reachable nodes (called *active* in [BR95]) the injectivity holds. These reachable nodes are really what is meant to be the stack. What we need now is a logical characterization of the stack, 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:

$$\ldots. \vdash (\text{cands} + (\text{currnode} \,/\, x)) \hat{} (F \hat{} n) = \text{cands} \hat{} (F \hat{} n), \ldots. \tag{23}$$

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

14 $F \hat{} n \neq \text{currnode}$

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 the currnode is not in the range of the stack under F.

A first approach to logically characterize the stack would be an algebraic specification of a function b-list that collects all nodes up to bottom and puts them into a list, such as:

b-list(bottom, b) = nil
$n \neq \text{bottom} \rightarrow$ b-list(n, b) = cons(st, b-list(b $\hat{}$ n, b))

But this is not a consistent definition because a cyclic `b` or a unreachable `bottom` would lead to a infinite list. Of course, that critical case will never occur in the given Evolving Algebra, because the rules never build up a cycle in `b`. But that is exactly what we have to prove while it is not guaranteed by the data structure (Essentially, this is the same problem as was discussed for function `clls` in section 4.1).

A correct approach is to use Dynamic Logic for expressing reachability. Thereby the inonsistency problem is altered to the question of program termination, which now is a subject to proof. We define a program b-list#:

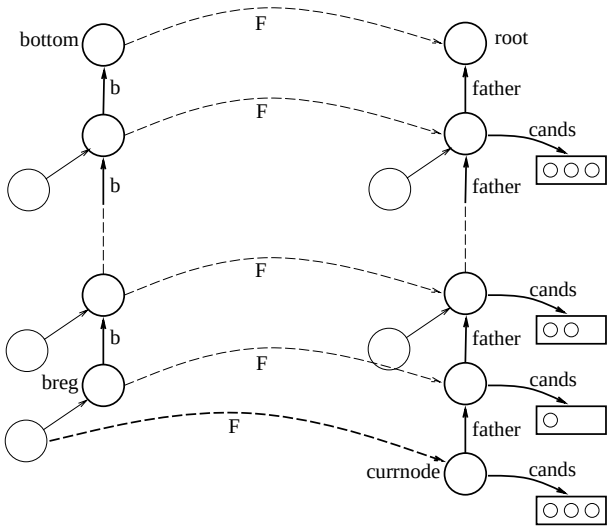

Fig. 14.

b-list#(n, b; **var** stack)
if st = bottom **then** stack := snil **else**
begin b-list#(b ^ n, b; stack); stack := cons(n, stack) **end**

Now let ψ(n) be the conjunction of all subformulas, which depend on the selected node n (5 to 8 and 13) and φ the conjunction of the remaining subformulas (1 to 4, 9 to 12 and 14). Then the coupling invariant INV gets the form:

$$\exists\ \mathrm{F}\colon \varphi \wedge \langle \text{b-list\#(breg, b; stack)} \rangle\ (\forall\ \mathrm{n}.\ \mathrm{n} \in \text{stack} \rightarrow \psi(\mathrm{n})) \tag{24}$$

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`.

6.2.3 Cutpoints

Proving cut-21 with this version of INV shows another difficulty. After (symbolic) execution of the cut# procedure, ψ must be guaranteed for the new stack that is given by b-list# applied to the new breg, which is the first cutpoint of the current decorated goal sequence. The new stack inherits ψ from the old one because it is a part of the old one! This is true but 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 need a new predicate called `cutptsin` to assert:

$$\text{decglseqreg cutptsin stack} \tag{25}$$

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

$$\text{(sdecglseq ^ n) cutptsin substack} \tag{26}$$

where substack is the output of $\langle$b-list#(b ^ n, b; substack)$\rangle$.

With these additions the coupling invariant (24) changes to:

$$
\begin{array}{l}
\exists\ \mathrm{F}.\quad \varphi \\
\quad \wedge\ \langle \text{b-list\#(breg, b; stack)} \rangle \\
\qquad (\quad \text{decglseqreg cutptsin stack} \\
\qquad\ \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \text{stack} \\
\qquad\qquad \rightarrow\quad \psi(\mathrm{n}) \\
\qquad\qquad\quad \wedge\ \langle \text{b-list\#(b \^{} n, b; substack)} \rangle (\text{sdecglseq \^{} n) cutptsin substack))}
\end{array}
\tag{27}
$$

Again, this is not strong enough. Proving cut-21 with this INV fails because `cutptsin` so far does not care about any ordering. Executing the cut#-procedure (which means that breg is repointed to the first cutpoint of decglseqreg) shortens the stack like some pop operations would do (compare Fig. 5. in section 2.2). 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`.

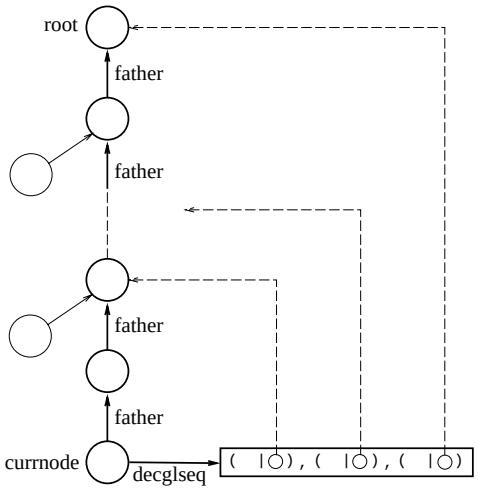

Fig. 15.

For this we have to change the definition of cutptsin (using an auxiliary function `from`, see specification in appendix A), 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.

6.2.4 More Properties

The coupling invariant is still not complete. Several further proof attempts revealed the necessity to make some 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 the candidates belonging to different nodes is empty

In addition, the two sets `ns` and `s` of both abstract machines, which characterize the domains of sorts `node` and `state` had to be described more exactly, for example:

- the stack is in s

- the range of the stack under F is in ns
- all cands are in ns

Finally, all formulas referring to `cands ^ currnode` have to be restricted to the `select` mode, because in call-mode `cands ^ currnode` is not yet defined. Please recognize that the final couling invariant is not a arbitrary accumulation of properties. All of them are actually needed to close proof goals!

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 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. The resulting coupling invariant is given in appendix D. With that formula the proofs of the lemmas (17) succeed together with the lemmas depending on (17).

6.3 Statistics

All in all it took 12 proof attempts to reach a correct coupling invariant. The verification work was done by the two authors in one month. The effort of two person months also included specification (see appendix A) and writing the interpreter programs (both about 100 lines of code, see appendix B and C). In contrast, verification of the final correct version took only two days. 1416 proof steps were necessary to complete the top-level proofs, which involved programs. Of these proof steps, 378 had to be given interactively, the rest were found automatically by the heuristics of the KIV system, giving an automation degree of 73.3 %. In addition to these proofs, 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, no or one interaction).

After the work on this case study, the KIV system was improved in Ulm from the experiences we learned. 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 present in NQTHM ([BM79]) (but not restricted to free datatypes), was added. 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. As can be seen from the statistic data in E, the improvements of KIV saved about 1/3 of the necessary interactions (now 246).

7 Conclusion

We have presented a framework for the formal verification of the compilation of Prolog to the WAM as given in [BR95]. The framework is based on the translation of deterministic Evolving Algebras to imperative programs over algebraic specifications. With this translation correctness and completeness of the transformation of one EA into another is expressible as program equivalence in Dynamic Logic.

We have shown a proof technique, based on coupling invariants, which corresponds to the use of proof maps over EA's. We have found that the correct coupling invariant, which is needed to show correctness and completeness of the first transformation step is far too complex to be stated correctly in a first attempt. The time to develop a correct version incrementally is much larger than the time it takes to verify the correct solution. Therefore besides the pure power of the theorem prover, the 'proof engineering' support offered by the verification system is crucial for the feasibility of the case study. The following items summarize the features of KIV, which were important for the successful verification:

- Explicit proof trees: KIV offers a visualization of proof trees of the sequent calculus, which allows to view every intermediate node by just clicking on it. Wrong decisions in proving a goal can be undone, by just pruning away parts of the proof tree. Also complex tactic applications like simplification, which appear as single steps in the proof tree, can be expanded to proof trees on demand. Analysing proof trees may be irrelevant for small case studies, but is of invaluable help in proving complex theorems, where goals often cover one or two pages, and proof trees can grow to sizes of several hundred steps. Then one is continuously faced with questions like: "What formulas in a large goal are relevant for proving it?", "This goal seems unprovable. Why and how did I get it?". Such questions can be efficiently answered only by inspecting the proof tree.

- Correctness management: KIV does not rely on bottom-up theorem proving as many other interactive theorem provers do. Instead a correctness management keeps track of the used lemmas in proving a theorem, and prohibits cycles in lemma dependencies. Also if a lemma is modified, exactly those proofs where it is used, are invalidated. Our typical procedure in proving the goals from the previous section was to define lemmas on the fly, but to prove the lemmas only after completing the main proof. This allows to follow the main line of arguments in a complex proof (which may take hours to complete), without being distracted by the need to prove auxiliary lemmas. In case the lemma defined was erroneous, we found that correcting it only very rarely caused significant revisions to the main proof. Most times a simple replay with the corrected lemma suffices.

- Reuse of proofs: KIV offers a strategy for the reuse of proofs, which goes beyond a simple replay of the old proof attempt. But although this strategy can handle a lot of modifications to the goal, we still are not satisfied with the degree of automation this gives when redoing proofs with a changed coupling invariant. We hope that a strategy for the reuse of proofs on arbitrary changes to the theorem based on a proof analysis similar to the one developed in ([RS95]) for program changes, which is currently developed, will improve the situation.

- Efficient simplification with large sets (more than 300) of rewrite rules: KIV compiles rewrite rules to compiled LISP code using the technique described in [Kap87] (with some extensions like AC-rewriting). With this technique the system time consumed by rewriting on the term level is only about 15%.

- Automation via suitable heuristics: KIV relies on heuristics to automate proofs [RSS95]. The heuristics used in a proof can be chosen freely, and can be changed any time during the proof. In our case study, we mainly used three sets of heuristics: One set for the difficult proofs of select and call rule, and another set, which more often splits cases for the easier rules. For the inductive proof (14) a third set was used, which additionally includes heuristics for loop's. Extending heuristics to better handle non-functional procedures (which use reference parameters as result *and* input) and loop's resulted in improvements in automation.

Verification showed that [BR95] is indeed an excellent analysis of the compilation problem from Prolog to WAM. Apart from a few syntactic details, which had to be corrected, the only remarkable change that was necessary is the introduction of an explicit compiler assumption (see sect. 2).

Parallel to our work, some work on the verification of a Prolog compiler is also done in Munich with the Isabelle-System ([Pus96]). In contrast to our approach, which was to model the EA-approach in DL, and to verify the correctness of transformations described in [BR95] as faithfully as possible, they started from an operational Prolog semantics defined in [DM88], which is already based on stacks, not on search trees, and used ([BR95]) only as a guidance for transformation steps. Therefore their first two transformation steps have no counterpart in our verification. Some comparison can be done for our second interpreter vs. their third interpreter. The two main differences are:

- Our interpreter program is an imperative program, whose semantics is a relation on states. In Isabelle, this semantics is given explicitly as an inductively defined relation.

- Our representation of the stack is the list `stack = [breg, b ^ breg, b ^ (b ^ breg), ...]` of nodes reachable from `breg` via the `b`-function as computed by the `B-LIST#` program. Information is attached to the nodes via functions `sub`, `decglseq` and `cll`. With the knowledge of the invariant, we derived in section 4, and it's complexity due to the use of pointer structures, this representation has been simplified in Isabelle to a list of tuples of the values of `sub(n)`, `decglseq(n)`, and `cll(n)` attached to the nodes `n` $\in$ `stack`. In this way, the sort `state` of stack nodes can be avoided altogether.

Let us conclude with an outlook on the continuing work on this case study. Parallel to the work on this paper three more transformation steps (covering section 1 and section 2.1 in [BR95]) have been verified, with the last transformation documented in [Ahr95] (in German). Work on the verification of more transformations is still continued. Also verification of all transformation steps does only yield a specification for a compiler, not an implementation. Therefore we also plan to implement a (verified) compiler based on the compiler assumptions derived from the transformation steps.

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 challenging, but tractable task.

8 Acknowledgements

We thank our colleagues Wolfgang Reif, Kurt Stenzel and Matthias Ott for their valuable comments on earlier drafts of this paper, and our student Harald Vogt for redoing the verification with the improved system.

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.

[DM88] Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *Journal of Logic Programming*, 5:61 – 91, 1988.

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

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

[Har79] D. Harel. *First Order Dynamic Logic.* Springer LNCS, 1979.

[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.

[Pus96] Cornelia Pusch. Verification of Compiler Correctness for the WAM. Unpublished report, 1996.

[Rei93] W. Reif. An Approach to Parameterized First-Order Specifications: Semantics, Correctness, Parameter Passing. In Pottosin, Bjorner, and Broy, editors, *Conference on Formal Methods in Programming and Their Applications*, Springer LNCS. Novosibirsk, Russia, 1993.

[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.

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

[SS86] Sterling and Shapiro. *The Art of Prolog.* MIT Press, 1986.

[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.

A The specification

A.1 Parameters

The sort of clause lines. Constant `undefcode` will be used to initialize function `cll`.

code =
specification
    sorts codesort;
    constants undefcode : codesort;
    variables co: codesort;
end specification

Clause lines for the second level. This time they are linked with a `next`-function, called `+` in [BR95].

codearea =
specification
    sorts codearea;
    constants failcode : codearea;
    functions next : codearea → codearea ;
    variables coa: codearea;
end specification

The sort of Prolog programs

program =
specification
    sorts program;
    variables db: program;
end specification

nodes of the search tree

node =
specification
    sorts nodesort;
    variables no: nodesort;
end specification

nodes of sort `state` for the stack of the second interpreter.

state =
specification
    sorts statesort;
    variables st: statesort;
end specification

Substitutions (not specified completely)

subst =
specification
sorts substitution;
constants $@_{su}$: substitution;
functions . o . : substitution × substitution → substitution ;
variables su_2, su_1, su: substitution;

axioms

(su o su_1) o su_2 = su o su_1 o su_2,
su o $@_{su}$ = su,
$@_{su}$ o su = su

end specification

A.2 Natural numbers

These specifications are from the library. Some of the functions and predicates of these specifications (e.g. <) are not used in this case study, but we do not care

nat-basic1 =
data specification
nat = 0
| . +1 (. -1 : nat)
;
variables n: nat;
order predicates . < . : nat × nat;
end data specification

nat-basic2 =
enrich nat-basic1 **with**
functions . + . : nat × nat → nat ;
variables n_0, m: nat;

axioms

n + 0 = n,
m + n +1 = (m + n)+1,
$n < n_0 \vee n = n_0 \vee n_0 < n$

end enrich

nat = nat-lec + nat-sub

nat-sub =
enrich nat-basic2 **with**
functions . - . : nat × nat → nat **prio** 4 **left**;

axioms

m - 0 = m,
m - n +1 = (m - n)-1

end enrich

nat-lec =
enrich nat-basic2 **with**
constants 1 : nat; 2 : nat;
predicates
$. \leq . \quad : \quad \text{nat} \times \text{nat};$
$. > . \quad : \quad \text{nat} \times \text{nat};$
$. \geq . \quad : \quad \text{nat} \times \text{nat};$

axioms

$1 = 0 + 1,$
$0 \neq 1,$
$2 = 0 + 1 + 1,$
$2 \neq 0,$
$2 \neq 1,$
$m \leq n \leftrightarrow \neg\ n < m,$
$m > n \leftrightarrow n < m,$
$m \geq n \leftrightarrow \neg\ m < n$

end enrich

A.3 pairs

Library specifications.

elem =
specification
sorts elem;
variables c, b, a: elem;
end specification

elemi =
rename elem **by morphism**
elem → elem', a → a'
end rename

elemii =
rename elem **by morphism**
elem → elem", a → a"
end rename

elemi-ii = elemi + elemii

pair =
generic data specification
parameter elemi-ii
pair = mkpair (. .p1 : elem', . .p2 : elem");
variables p_1, p_0, p: pair;
end generic data specification

A.4 lists

The first three specifications are from the library. The rest are different actualizations. To resolve overloading, different instances of the datatype have different subscripts for their operations.

list-data =
generic data specification
parameter elem **using** nat
list = @ **with** $@_p$
| . $\oplus$. (. .first : elem, . .rest : list)
;
variables z, y, x: list;
size functions # : list $\to$ nat ;
order predicates . $\ll$. : list $\times$ list;
end generic data specification

list =
enrich list-data **with**
functions . $\odot$. : list $\times$ list $\to$ list **prio** 4;
predicates . $\in$. : elem $\times$ list;

axioms

@ $\odot$ x = x,
a $\oplus$ x $\odot$ y = a $\oplus$ (x $\odot$ y),
a $\in$ x $\leftrightarrow$ ($\exists$ y, z. x = y $\odot$ a $\oplus$ z)

end enrich

sublist =
enrich list **with**
predicates . subli . : list $\times$ list;

axioms

@ subli x,
$\neg$ a $\oplus$ x subli @,
a $\oplus$ x subli b $\oplus$ y $\leftrightarrow$ a = b $\wedge$ x subli y $\vee$ a $\neq$ b $\wedge$ a $\oplus$ x subli y

end enrich

statelist =
actualize list **with parameter** state **by morphism**
elem $\to$ statesort, list $\to$ statelist, @ $\to$ snil, $\oplus$ $\to$ $+_{sl}$, .first $\to$ scar **prio** 0, .rest $\to$ scdr **prio** 0, $\odot$ $\to$ $\odot_{stl}$, # $\to$ $\#_{stl}$, $\ll$ $\to$ $\ll_s$, $\in$ $\to$ $\in_{sl}$, $@_p$ $\to$ snilp, a $\to$ st, x $\to$ stl
end actualize

nodelist =
actualize list **with parameter** node **by morphism**
elem $\to$ nodesort, list $\to$ nodelist, @ $\to$ nnil, .first $\to$ ncar **prio** 0, .rest $\to$ ncdr **prio** 0, # $\to$ $\#_{nl}$, $\odot$ $\to$ $\odot_{nl}$, $\oplus$ $\to$ $+_{nl}$, $\ll$ $\to$ $\ll_n$, $\in$ $\to$ $\in_{nl}$, $@_p$ $\to$ nnilp, a $\to$ no, x $\to$ nol
end actualize

codelist =
actualize list **with parameter** code **by morphism**
elem $\to$ codesort, list $\to$ codelist, @ $\to$ cnil, # $\to$ $\#_{col}$, $\oplus$ $\to$ $+_{col}$, .first $\to$ ccar **prio** 0, .rest $\to$ ccdr **prio** 0, $\odot$ $\to$ $\odot_{col}$, $@_p$ $\to$ cnilp, $\ll$ $\to$ $\ll_c$, $\in$ $\to$ $\in_{col}$, a $\to$ co, x $\to$ col
end actualize

codealist =
actualize list **with parameter** codearea **by morphism**

elem → codearea, list → codealist, @ → canil, $\oplus \rightarrow +_{cal}$, .first → cacar **prio** 0, .rest → cacdr **prio** 0, # → $\#_{cal}$, $\odot \rightarrow \odot_{cal}$, $\ll \rightarrow \ll_{ca}$, $\in \rightarrow \in_{cal}$, $@_p$ → canilp, a → coa, x → cal
end actualize

clauselist =
actualize sublist **with** clause **by morphism**
elem → clausesort, list → clauselist, @ → clnil, $\oplus \rightarrow +_{cli}$, .first → clcar **prio** 0, .rest → clcdr **prio** 0, # → $\#_{cli}$, $\odot \rightarrow \odot_{cli}$, $\ll \rightarrow \ll_{cl}$, subli → subli_of, $\in \rightarrow \in_{cli}$, $@_p$ → clnilp, a → cl, x → cli
end actualize

decgoallist =
actualize list **with** decgoal **by morphism**
elem → decgoal, list → decgoallist, @ → dnil, $\oplus \rightarrow +_{dl}$, .first → dcar **prio** 0, .rest → dcdr **prio** 0, # → $\#_{dgl}$, $\odot \rightarrow \odot_{dgl}$, $\ll \rightarrow \ll_{d}$, $\in \rightarrow \in_{dgl}$, $@_p$ → dnilp, a → dg, x → dgl
end actualize

sdecgoallist =
actualize list **with** sdecgoal **by morphism**
elem → sdecgoal, list → sdecgoallist, @ → sdnil, $\oplus \rightarrow +_{sdl}$, .first → sdcar **prio** 0, .rest → sdcdr **prio** 0, # → $\#_{sdl}$, $\odot \rightarrow \odot_{sdl}$, $\ll \rightarrow \ll_{sd}$, $\in \rightarrow \in_{sdl}$, $@_p$ → sdnilp, a → sdg, x → sdgl
end actualize

A.5 sets

The specification given in the main text has been split into the basic specification from the library, and the enrichment of the `new` function. The library specification includes some additional functions and predicates.

set =
generic specification
parameter elem **using** nat **target**
sorts set;
constants $\emptyset$: set;
functions

. ++ .	:	set × elem	→	set	**prio** 5 **left**;
. '	:	elem	→	set	;
#	:	set	→	nat	;
. − .	:	set × elem	→	set	**prio** 5 **left**;

predicates
. ∈ . : elem × set;
. ⊆ . : set × set;
variables s', s: set;

axioms

set **generated by** $\emptyset$, ++;
s = s' ↔ (∀ a. a ∈ s ↔ a ∈ s'),
¬ a ∈ $\emptyset$,
a ∈ s ++ b ↔ a = b ∨ a ∈ s,
a ' = $\emptyset$ ++ a,
#($\emptyset$) = 0,
¬ a ∈ s → #(s ++ a) = #(s)+1,
a ∈ s − b ↔ a ≠ b ∧ a ∈ s,
s ⊆ s' ↔ (∀ a. a ∈ s → a ∈ s')

end generic specification

enrset =
enrich set **with**
functions new : set → elem ;

axioms

¬ new(s) ∈ s

end enrich

nodeset =
actualize enrset **with parameter** node **by morphism**
elem → nodesort, set → nodeset, ∅ → $@_{ns}$, ++ → $+_{ns}$, - → $-_{ns}$, # → $\#_{ns}$, ' → $'_{ns}$, new → new, ∈ → $\in_n$, ⊆ → $\subseteq_{ns}$, a → no, s → ns
end actualize

stateset =
actualize enrset **with parameter** state **by morphism**
elem → statesort, set → stateset, ∅ → $@_s$, ++ → $+_s$, - → $-_s$, # → $\#_s$, ' → $'_s$,
new → snew, ∈ → $\in_s$, ⊆ → $\subseteq_s$, a → st, s → s
end actualize

enrnodeset =
enrich nodeset **with**
constants root : nodesort;

axioms

new($@_{ns}$) = root

end enrich

enrstateset =
enrich stateset **with**
constants bottom : statesort;

axioms

snew($@_s$) = bottom

end enrich

A.6 modes

mode =
data specification
modesort = select
| call
;
variables mode: modesort;
end data specification

stopmode =
data specification
stopmodesort = success
| failure
| run
;
variables stop: stopmodesort;
end data specification

A.7 terms,clauses,goals,lists of decorated goals

Terms are specified as a parameter, with only the information, that `!`,`true` and `fail` are included

paramterm =
specification
sorts paramterm;
constants ! : paramterm; true' : paramterm; fail' : paramterm;
predicates is_user_defined : paramterm;
variables trm: paramterm;

axioms

! $\neq$ true',
! $\neq$ fail',
true' $\neq$ fail',
is_user_defined(trm) $\leftrightarrow$ trm $\neq$ true' $\wedge$ trm $\neq$ fail' $\wedge$ trm $\neq$!

end specification

goals are lists of terms.

goal =
actualize list **with parameter** paramterm **by morphism**
elem $\rightarrow$ paramterm, list $\rightarrow$ goalsort, @ $\rightarrow$ gnil, $\oplus \rightarrow +_g$, .first $\rightarrow$ gcar **prio** 0, .rest $\rightarrow$ gcdr **prio** 0, # $\rightarrow \#_{goal}$, $\odot \rightarrow \odot_{goal}$, $\ll \rightarrow \ll_g$, $\in \rightarrow \in_{goal}$, $@_p$ $\rightarrow$ gnilp, a $\rightarrow$ trm, x $\rightarrow$ go
end actualize

clauses are pairs of head and body, where head is a term and body is a goal.

clause =
actualize pair **with** goal **by morphism**
elem' $\rightarrow$ paramterm, elem" $\rightarrow$ goalsort, pair $\rightarrow$ clausesort, mkpair $\rightarrow$ mkclause, .p1 $\rightarrow$ hd **prio** 0, .p2 $\rightarrow$ bdy **prio** 0, p $\rightarrow$ cl
end actualize

The result of the `clause'` function, which selects clauses from clause lines in the second interpreter. The special value `null` is called `nil` in [BR95].

clauseornull =
data specification
using clause
clauseornull = mkclau (the_clau : clausesort)
| null
;
variables cln: clauseornull;
end data specification

decorated goals are specifed as pairs of goals and nodes.

decgoal =
actualize pair **with parameter** node, goal **by morphism**
elem" $\rightarrow$ nodesort, elem' $\rightarrow$ goalsort, pair $\rightarrow$ decgoal, mkpair $\rightarrow$ mkdecgoal, .p1 $\rightarrow$ .1, .p2 $\rightarrow$ .2, p $\rightarrow$ dg
end actualize

decorated goals for the second interpreter: pairs of goals and nodes of sort `state`.

sdecgoal =
actualize pair **with parameter** state, goal **by morphism**
elem" → statesort, elem' → goalsort, pair → sdecgoal, mkpair → mksdecgoal,
.p1 → .s1, .p2 → .s2, p → sdg
end actualize

A.8 dynamic functions

Dynamic functions. Mixfix-Operation operation `. + ( . / . )` is simulated by two infix operations. To resolve overloading, different instances of the datatype have different subscripts for their operations.

dynfun =
generic specification
parameter elemi-ii **target**
sorts dynfun, pairdomcod;
functions

constfun	:	elem"	→	dynfun	;
. ^ .	:	dynfun × elem'	→	elem"	**prio** 1;
. $+_{fun}$.	:	dynfun × pairdomcod	→	dynfun	;
. / .	:	elem' × elem"	→	pairdomcod	**prio** 9;

variables $func_2$, $func_1$, func: dynfun; pdc: pairdomcod;

axioms

dynfun **generated by** constfun, $+_{fun}$;
pairdomcod **generated by** /;
constfun(a") ^ a' = a",
func $+_{fun}$ a' / a" ^ a' = a",
a' ≠ b' → func $+_{fun}$ a' / a" ^ b' = func ^ b'

end generic specification

The type of dynamic function `cands`.

cands =
actualize dynfun **with parameter** node, nodelist **by morphism**
elem' → nodesort, elem" → nodelist, pairdomcod → pairnodenodelist, dynfun → candsfun, constfun → ccands, ^ → $^{\wedge}_n$, / → $/_n$, $+_{fun}$ → $+_n$, func → cands
end actualize

The type of dynamic function `father`.

father =
actualize dynfun **with parameter** node **by morphism**
elem' → nodesort, elem" → nodesort, pairdomcod → pairnodenode, dynfun → fatherfun, constfun → cfather, ^ → $^{\wedge}_{fa}$, / → $/_{fa}$, $+_{fun}$ → $+_{fa}$, func → father
end actualize

The type of dynamic function `sub`.

sub =
actualize dynfun **with parameter** node, **parameter** subst **by morphism**
elem' → nodesort, elem" → substitution, pairdomcod → pairnodesubst, dynfun → subfun, constfun → csub, ^ → $^{\wedge}_u$, / → $/_u$, $+_{fun}$ → $+_u$, func → sub
end actualize

The type of dynamic function `cll`.

cll =
actualize dynfun **with parameter** node, **parameter** code **by morphism**
elem' → nodesort, elem" → codesort, pairdomcod → pairnodecode, dynfun → cllfun, constfun → ccll, ^ → $^{\wedge}_{cll}$, / → $/_{cll}$, $+_{fun} \rightarrow +_{cll}$, func → cll
end actualize

The type of dynamic function `decglseq`.

decglseq =
actualize dynfun **with parameter** node, decgoallist **by morphism**
elem' → nodesort, elem" → decgoallist, pairdomcod → pairnodedecgoallist, dynfun → decglseqfun, constfun → cdecglseq, ^ → $^{\wedge}_{d}$, / → $/_{d}$, $+_{fun} \rightarrow +_{d}$, func → decglseq
end actualize

The type of dynamic function `b`.

b =
actualize dynfun **with parameter** state **by morphism**
elem' → statesort, elem" → statesort, pairdomcod → pairstatestate, dynfun → bfun, constfun → cb, ^ → $^{\wedge}_{b}$, / → $/_{b}$, $+_{fun} \rightarrow +_{b}$, func → b
end actualize

The type of dynamic function `ssub` (replaces `sub` from the first level).

ssub =
actualize dynfun **with parameter** subst, **parameter** state **by morphism**
elem' → statesort, elem" → substitution, pairdomcod → pairstatesubst, dynfun → ssubfun, constfun → cssub, ^ → $^{\wedge}_{su}$, / → $/_{su}$, $+_{fun} \rightarrow +_{su}$, func → ssub
end actualize

The type of dynamic function `sdecglseq` (replaces `decglseq` from the first level).

sdecglseq =
actualize dynfun **with parameter** state, sdecgoallist **by morphism**
elem' → statesort, elem" → sdecgoallist, pairdomcod → pairstatesdecgoallist, dynfun → sdecglseqfun, constfun → csdecglseq, ^ → $^{\wedge}_{sd}$, / → $/_{sd}$, $+_{fun} \rightarrow +_{sd}$, func → sdecglseq
end actualize

The type of dynamic function `scll` (replaces `cll` from the first level).

scll =
actualize dynfun **with parameter** state, **parameter** codearea **by morphism**
elem' → statesort, elem" → codearea, pairdomcod → pairstatecoa, dynfun → scllfun, constfun → cscll, ^ → $^{\wedge}_{sc}$, / → $/_{sc}$, $+_{fun} \rightarrow +_{sc}$, func → scll
end actualize

A.9 substitution,renaming and unification

The value `fail` is added to the (parameter) of substitutions.

substorfail =
data specification
using parameter subst
substorfail = oksubst (the_subst : substitution)

| fail
;
variables subst: substorfail;
end data specification

function `unify` (unspecified).

unify =
enrich substorfail, **parameter** paramterm **with**
functions unify : paramterm × paramterm → substorfail ;

end enrich

Application of substitutions on terms (unspecified).

substterm =
enrich parameter subst, **parameter** paramterm **with**
functions . $\hat{}_t$. : substitution × paramterm → paramterm ;

end enrich

Application of substitutions on goals.

substgoal =
enrich substterm, goal **with**
functions . $\hat{}_{sg}$. : substitution × goalsort → goalsort ;

axioms

su $\hat{}_{sg}$ gnil = gnil,
su $\hat{}_{sg}$ trm $+_g$ go = (su $\hat{}_t$ trm) $+_g$ su $\hat{}_{sg}$ go

end enrich

Application of substitution on decorated goals (first level)

subres =
enrich decgoallist, substgoal **with**
functions subres : decgoallist × substitution → decgoallist ;

axioms

subres(dnil, su) = dnil,
subres(mkdecgoal(go, no) $+_{dl}$ dgl, su) = mkdecgoal(su $\hat{}_{sg}$ go, no) $+_{dl}$ subres(dgl, su)

end enrich

Application of substitution on decorated goals (second level)

ssubres =
enrich sdecgoallist, substgoal **with**
functions ssubres : sdecgoallist × substitution → sdecgoallist ;

axioms

ssubres(sdnil, su) = sdnil,
ssubres(mksdecgoal(go, st) $+_{sdl}$ sdgl, su)
= mksdecgoal(su $\hat{}_{sg}$ go, st) $+_{sdl}$ ssubres(sdgl, su)

end enrich

Renaming of clauses (unspecified)

rename =
enrich nat, clause **with**
functions ren : clausesort × nat → clausesort ;

end enrich

A.10 clause occurences, procdef

function `clause` yields the clause at a clause line (first level).

clausefun =
enrich parameter code, **parameter** program, clause **with**
functions clause : codesort × program → clausesort ;

end enrich

function `clause'` yields the clause at a clause line (second level).

clause'fun =
enrich parameter codearea, clauseornull, **parameter** program **with**
functions clause' : codearea × program → clauseornull ;

axioms

clause'(failcode, db) = null

end enrich

The `procdef` function of the first level

procdef =
enrich parameter paramterm, **parameter** program, codelist **with**
functions procdef : paramterm × program → codelist ;

end enrich

The `procdef` function of the second level

procdef1 =
enrich parameter codearea, **parameter** program, **parameter** paramterm **with**
functions procdef' : paramterm × program → codearea ;

end enrich

A.11 toplevel specification for the first interpreter

Contains some auxiliary functions used in the coupling invariant.

prologtree =
enrich union1 **with**
functions

mapclause	:	codelist × program	→	clauselist	;
map	:	cllfun × nodelist	→	codelist	;

predicates

every : fatherfun × nodelist × nodesort;
nodups : nodelist;
. nl⊆s . : nodelist × nodeset;
disjoint : nodelist × nodelist;
disjointls : nodelist × nodeset;

axioms

mapclause(cnil, db) = clnil,
mapclause(co $+_{col}$ col, db) = clause(co, db) $+_{cli}$ mapclause(col, db),
every(father, nnil, no),
every(father, no_1 $+_{nl}$ nol, no) $\leftrightarrow$ father $\hat{}_{fa}$ no_1 = no $\wedge$ every(father, nol, no),
map(cll, nnil) = cnil,
map(cll, no $+_{nl}$ nol) = (cll $\hat{}_{cll}$ no) $+_{col}$ map(cll, nol),
nodups(nnil),
nodups(no $+_{nl}$ nol) $\leftrightarrow$ $\neg$ no $\in_{nl}$ nol $\wedge$ nodups(nol),
nol nl⊆s ns $\leftrightarrow$ ($\forall$ no. no $\in_{nl}$ nol $\rightarrow$ no $\in_n$ ns),
disjoint(nol, nol_0) $\leftrightarrow$ ($\forall$ no. no $\in_{nl}$ nol $\rightarrow$ $\neg$ no $\in_{nl}$ nol_0),
disjointls(nol, ns) $\leftrightarrow$ ($\forall$ no. no $\in_{nl}$ nol $\rightarrow$ $\neg$ no $\in_n$ ns)

end enrich

A.12 toplevel specification for the second interpreter

Contains some auxiliary functions used in the coupling invariant.

prologstack =
enrich union2 **with**
functions
mapclause' : codealist × program → clauselist ;
. from . : statelist × statesort → statelist **prio** 3;
cdr : statelist → statelist ;
predicates
. cutptsin . : sdecgoallist × statelist;
. ctpelem . : sdecgoallist × stateset;
. sl⊆s . : statelist × stateset;

axioms

mapclause'(canil, db) = clnil,
mapclause'(coa $+_{cal}$ cal, db) = the_clau(clause'(coa, db)) $+_{cli}$ mapclause'(cal, db),
sdnil cutptsin stl,
mksdecgoal(go, st) $+_{sdl}$ sdgl cutptsin stl
$\leftrightarrow$ (st = bottom $\vee$ st $\in_{sl}$ stl) $\wedge$ sdgl cutptsin stl from st,
snil from st = snil,
st $+_{sl}$ stl from st = st $+_{sl}$ stl,
$st_1 \neq$ st $\rightarrow$ st_1 $+_{sl}$ stl from st = stl from st,
sdnil ctpelem s,
mksdecgoal(go, st) $+_{sdl}$ sdgl ctpelem s $\leftrightarrow$ st $\in_s$ s $\wedge$ sdgl ctpelem s,
stl sl⊆s s $\leftrightarrow$ ($\forall$ st. st $\in_{sl}$ stl $\rightarrow$ st $\in_s$ s),
cdr(snil) = snil,
cdr(st $+_{sl}$ stl) = stl

end enrich

A.13 specification of first transformation step

The mapping `F` from states to nodes used in the invariant.

f-st-no =
actualize dynfun **with parameter** node, **parameter** state **by morphism**
elem' $\rightarrow$ statesort, elem" $\rightarrow$ nodesort, pairdomcod $\rightarrow$ pairstatenode, dynfun $\rightarrow$ funstatenode, constfun $\rightarrow$ cfn, $\hat{} \rightarrow \hat{}_{fn}$, / $\rightarrow$ $/_{fn}$, $+_{fun} \rightarrow +_{fn}$, func $\rightarrow$ fn
end actualize

Some auxiliary functions used in the coupling invariant, involving either `F` or data types from both interpreters.

tree+stack+f =
enrich f-st-no, prologtree, prologstack **with**
functions

fnd	:	funstatenode $\times$ sdecgoallist	$\rightarrow$	decgoallist	;
fns	:	funstatenode $\times$ stateset	$\rightarrow$	nodeset	;

predicates

candsdisjoint	:	funstatenode $\times$ candsfun $\times$ statelist;
. injonn .	:	funstatenode $\times$ statelist;
nocands	:	funstatenode $\times$ candsfun $\times$ statelist;

axioms

fnd(fn, sdnil) = dnil,
fnd(fn, mksdecgoal(go, st) $+_{sdl}$ sdgl) = mkdecgoal(go, fn $\hat{}_{fn}$ st) $+_{dl}$ fnd(fn, sdgl),
fns(fn, $@_s$) = $@_{ns}$,
fns(fn, s $+_s$ st) = fns(fn, s) $+_{ns}$ (fn $\hat{}_{fn}$ st),
candsdisjoint(fn, cands, stl)
$\leftrightarrow \forall$ st, st_1. st $\in_{sl}$ stl $\wedge$ st_1 $\in_{sl}$ stl $\wedge$ st $\neq$ st_1
$\rightarrow$ disjoint(cands $\hat{}_n$ fn $\hat{}_{fn}$ st_1, cands $\hat{}_n$ fn $\hat{}_{fn}$ st),
fn injonn stl
$\leftrightarrow \forall$ st, st_1. st $\in_{sl}$ stl $\wedge$ st_1 $\in_{sl}$ bottom $+_{sl}$ stl $\wedge$ st $\neq$ st_1 $\rightarrow$ fn $\hat{}_{fn}$ st $\neq$ fn $\hat{}_{fn}$ st_1,
nocands(fn, cands, stl)
$\leftrightarrow \forall$ st, st_1. st $\in_{sl}$ stl $\wedge$ st_1 $\in_{sl}$ bottom $+_{sl}$ stl $\rightarrow \neg$ (fn $\hat{}_{fn}$ st_1) $\in_{nl}$ cands $\hat{}_n$ fn $\hat{}_{fn}$ st

end enrich

A.14 toplevel specification for the equivalence proof

The toplevel specification, in which correctness and completeness of the first transformation are shown. Includes the compiler assumption as an axiom. For the procedure declaration of `clls#` see appendix D.

treetostack =
enrich tree+stack+f **with**
functions $compile_1$: program $\rightarrow$ program ;

axioms

$\langle$clls#(procdef'(lit, $compile_1$(db)), $compile_1$(db); cal)$\rangle$
mapclause(procdef(lit, db), db) = mapclause'(cal, $compile_1$(db)),

end enrich

B The code of interpreter1

The main routine here is eval# (called EVAL1# in the main text)

```
eval#(db, goal; var subst)
var ns = @_ns +_ns root, vi = 0, mode = call, stop = run, currnode = new(@_ns +_ns root) in
var ns = ns +_ns currnode,
     sub = csub(@_su),
     father = cfather(root),
     decglseq = cdecglseq(dnil) +_d currnode /_d (mkdecgoal(goal, root) +_dl dnil),
     cands = ccands(nnil),
     cll = ccll(undefcode)
in
begin
   while stop = run do
       body#(db; ns, vi, mode, stop, currnode, sub, father, decglseq, cands, cll);
   if stop = failure then subst := fail else
     subst := oksubst(sub ^_u currnode)
end

body#(db; var ns, vi, mode, stop, currnode, sub, father, decglseq, cands, cll)
var decgoalseq = decglseq ^_d currnode in
if decgoalseq = dnil then query-success#(; stop) else
  var goal = dcar(decgoalseq).1 in
  if goal = gnil then goal-success#(currnode, decgoalseq; decglseq) else
    var act = gcar(goal), cutpt = dcar(decgoalseq).2, fath = father ^_fa currnode in
    var cont = mkdecgoal(gcdr(goal), cutpt) +_dl dcdr(decgoalseq) in
    if is_user_defined(act) then
      if mode = call then
        call#(currnode, procdef(act, db); ns, father, cll, cands, mode)
      else
        var cnds = cands ^_n currnode in
        select#(db, act, cnds, cll, fath, cont; cands, decglseq, mode, vi, currnode, stop, sub)
    else
      if act = true' then true#(currnode, cont; decglseq) else
        if act = fail' then fail#(fath; stop, currnode, mode) else
          cut#(currnode, cont, cutpt; father, decglseq)

query-success#(var stop) begin stop := success end

goal-success#(currnode, decgoalseq; var decglseq)
decglseq := decglseq +_d currnode /_d dcdr(decgoalseq)

call#(currnode, procdefact; var ns, father, cll, cands, mode)
if procdefact = cnil then
  begin cands := cands +_n currnode /_n nnil; mode := select end
else
  begin
     call#(currnode, ccdr(procdefact); ns, father, cll, cands, mode);
     var no = new(ns) in
      begin
        ns := ns +_ns no;
        father := father +_fa no /_fa currnode;
        cll := cll +_cll no /_cll ccar(procdefact);
        cands := cands +_n currnode /_n (no +_nl (cands ^_n currnode))
```

```
    end
  end

select#(db, act, cnds, cll, fath, cont; var cands, decglseq, mode, vi, currnode, stop, sub)
if cnds = nnil then backtrack#(fath; stop, currnode, mode) else
  var cla = ren(clause(cll ^cll ncar(cnds), db), vi) in
  var uni = unify(act, hd(cla)) in
  if uni = fail then cands := cands +n currnode /n ncdr(cnds) else
    begin
      cands := cands +n currnode /n ncdr(cnds);
      sub := sub +u ncar(cnds) /u ((sub ^u currnode) o the_subst(uni));
      currnode := ncar(cnds);
      decglseq := decglseq +d currnode /d subres(mkdecgoal(bdy(cla), fath) +dl cont, the_subst(uni));
      mode := call;
      vi := vi +1
    end

true#(currnode, cont; var decglseq) begin decglseq := decglseq +d currnode /d cont end

fail#(fath; var stop, currnode, mode) begin backtrack#(fath; stop, currnode, mode) end

cut#(currnode, cont, cutpt; var father, decglseq)
begin
   father := father +fa currnode /fa cutpt;
   decglseq := decglseq +d currnode /d cont
end

backtrack#(fath; var stop, currnode, mode)
if fath = root then stop := failure else
  begin currnode := fath; mode := select end
```

C The code of interpreter2

The main routine here is s-eval# (called EVAL2# in the main text)

```
s-eval#(db', goal; var subst)
var s = @_s +_s bottom,
      vi' = 0,
      mode' = call,
      stop' = run,
      breg = bottom,
      ssub = cssub(@_su),
      subreg = @_su,
      sdecglseq = csdecglseq(sdnil),
      decglseqreg = mksdecgoal(goal, bottom) +_sdl sdnil,
      scll = cscll(failcode),
      cllreg = failcode,
      b = cb(bottom)
in
begin
   while stop' = run do
        s-body#(db'; s, vi', mode', stop', breg, ssub, subreg, sdecglseq, decglseqreg, scll, cllreg, b);
   if stop' = failure then subst := fail else
     subst := oksubst(subreg)
end

s-body#(db'; var s, vi', mode', stop', breg, ssub, subreg, sdecglseq, decglseqreg, scll, cllreg, b)
if decglseqreg = sdnil then s-query-success#(; stop') else
  var goal = sdcar(decglseqreg).s1 in
  if goal = gnil then s-goal-success#(; decglseqreg) else
    var act = gcar(goal), scutpt = sdcar(decglseqreg).s2 in
    var scont = mksdecgoal(gcdr(goal), scutpt) +_sdl sdcdr(decglseqreg) in
    if is_user_defined(act) then
      if mode' = call then s-call#(act, db'; cllreg, mode') else
        s-select#(db', act, scont; stop', cllreg, scll, subreg, ssub, breg, b, decglseqreg, sdecglseq, vi',
s, mode')
    else
      if act = true' then s-true#(scont; decglseqreg) else
        if act = fail' then
          s-fail#(b, sdecglseq, ssub, scll; cllreg, stop', decglseqreg, subreg, breg, mode')
        else
          s-cut#(scont, scutpt; breg, decglseqreg)

s-query-success#(var stop') begin stop' := success end

s-goal-success#(var decglseqreg) begin decglseqreg := sdcdr(decglseqreg) end

s-call#(act, db'; var cllreg, mode')
begin cllreg := procdef'(act, db'); mode' := select end

s-select#(db', act, scont; var stop', cllreg, scll, subreg, ssub, breg, b, decglseqreg, sdecglseq, vi',
s, mode')
if clause'(cllreg, db') = null then
  s-backtrack#(b, sdecglseq, ssub, scll; cllreg, stop', decglseqreg, subreg, breg, mode')
else
  var cla = ren(the_clau(clause'(cllreg, db')), vi') in
```

```
  var uni = unify(act, hd(cla)) in
  if uni = fail then cllreg := next(cllreg) else
    var temp = snew(s) in
    begin
       s := s +_s temp;
       b := b +_b temp /_b breg;
       sdecglseq := sdecglseq +_sd temp /_sd decglseqreg;
       ssub := ssub +_su temp /_su subreg;
       scll := scll +_sc temp /_sc next(cllreg);
       decglseqreg := ssubres(mksdecgoal(bdy(cla), breg) +_sdl scont, the_subst(uni));
       breg := temp;
       subreg := subreg o the_subst(uni);
       mode' := call;
       vi' := vi' +1
    end

s-true#(scont; var decglseqreg) begin decglseqreg := scont end

s-fail#(b, sdecglseq, ssub, scll; var cllreg, stop', decglseqreg, subreg, breg, mode')
begin s-backtrack#(b, sdecglseq, ssub, scll; cllreg, stop', decglseqreg, subreg, breg, mode')
end

s-cut#(scont, scutpt; var breg, decglseqreg)
begin breg := scutpt; decglseqreg := scont end

s-backtrack#(b, sdecglseq, ssub, scll; var cllreg, stop', decglseqreg, subreg, breg, mode')
if breg = bottom then stop' := failure else
  begin
     decglseqreg := sdecglseq ^_sd breg;
     subreg := ssub ^_su breg;
     cllreg := scll ^_sc breg;
     breg := b ^_b breg;
     mode' := select
  end
```

D The coupling invariant

$\exists$ F. stop = stop'
$\wedge$ mode = mode'
$\wedge$ subreg = sub $\hat{}_u$ currnode
$\wedge$ F $\hat{}_{fn}$ bottom = root
$\wedge$ F $\hat{}_{fn}$ breg = father $\hat{}_{fa}$ currnode
$\wedge$ fnd(F, decglseqreg) = decglseq $\hat{}_d$ currnode
$\wedge$ vi = vi'
$\wedge$ bottom $\in_s$ s
$\wedge$ root $\neq$ currnode
$\wedge$ root $\in_n$ ns
$\wedge$ currnode $\in_n$ ns
$\wedge$ (mode = select
$\rightarrow$ $\langle$clls#(cllreg, db'; cal)$\rangle$
mapclause'(cal, db') = mapclause(mapcll(cll, cands $\hat{}_n$ currnode), db)
$\wedge$ every(father, cands $\hat{}_n$ currnode, currnode)
$\wedge$ $\neg$ currnode $\in_{nl}$ cands $\hat{}_n$ currnode
$\wedge$ $\neg$ root $\in_{nl}$ cands $\hat{}_n$ currnode
$\wedge$ (cands $\hat{}_n$ currnode) nl$\subseteq$s ns
$\wedge$ nodups(cands $\hat{}_n$ currnode))
$\wedge$ $\langle$b-list#(breg, b; stack)$\rangle$
(decglseqreg cutptsin stack
$\wedge$ candsdisjoint(F, cands, stack)
$\wedge$ F injonn stack
$\wedge$ nocands(F, cands, stack)
$\wedge$ stack sl$\subseteq$s s
$\wedge$ ($\forall$ st. st $\in_{sl}$ stack
$\rightarrow$ ssub $\hat{}_{su}$ st = sub $\hat{}_u$ F $\hat{}_{fn}$ st
$\wedge$ F $\hat{}_{fn}$ b $\hat{}_b$ st = father $\hat{}_{fa}$ F $\hat{}_{fn}$ st
$\wedge$ fnd(F, sdecglseq $\hat{}_{sd}$ st) = decglseq $\hat{}_d$ F $\hat{}_{fn}$ st
$\wedge$ $\langle$clls#(scll $\hat{}_{sc}$ st, db'; cal)$\rangle$
mapclause'(cal, db') = mapclause(mapcll(cll, cands $\hat{}_n$ F $\hat{}_{fn}$ st), db)
$\wedge$ every(father, cands $\hat{}_n$ F $\hat{}_{fn}$ st, F $\hat{}_{fn}$ st)
$\wedge$ F $\hat{}_{fn}$ st $\neq$ currnode
$\wedge$ (F $\hat{}_{fn}$ st) $\in_n$ ns
$\wedge$ $\neg$ currnode $\in_{nl}$ cands $\hat{}_n$ F $\hat{}_{fn}$ st
$\wedge$ (cands $\hat{}_n$ F $\hat{}_{fn}$ st) nl$\subseteq$s ns
$\wedge$ (mode = select
$\rightarrow$ $\neg$ (F $\hat{}_{fn}$ st) $\in_{nl}$ cands $\hat{}_n$ currnode
$\wedge$ disjoint(cands $\hat{}_n$ F $\hat{}_{fn}$ st, cands $\hat{}_n$ currnode))
$\wedge$ nodups(cands $\hat{}_n$ F $\hat{}_{fn}$ st)
$\wedge$ $\langle$b-list#(b $\hat{}_b$ st, b; substack)$\rangle$(sdecglseq $\hat{}_{sd}$ st) cutptsin substack))

Additional declarations for procedures b-list# and clls#:

b-list#(st, b; **var** stack)
if st = bottom **then** stack := snil **else**
begin b-list#(b $\hat{}_b$ st, b; stack); stack := st $+_{sl}$ stack **end**

clls#(coa, db'; **var** cal)
if clause'(coa, db') = null **then** cal := canil **else**
begin clls#(next(coa), db'; cal); cal := coa $+_{cal}$ cal **end**

E Some statistics

- Lemmas: 23
- Proof steps: 1475
- Interactions: 246
- Automation: 83.3 %

Statistic for each lemma:

	proof steps	interactions	automation
select-12	359	61	83.0 %
step-lemma-21	147	5	96.5 %
fail-12	124	13	89.5 %
call-12	111	15	86.4 %
t-call-lemma	92	21	77.1 %
cut-12	75	11	85.3 %
eval-imp-21b	72	19	73.6 %
eval-imp-1b2	71	18	74.6 %
true-12	50	3	94.0 %
goal-success-12	49	4	91.8 %
ind-lemma-21	48	16	66.6 %
ind-lemma-12	44	17	61.3 %
query-success-12	31	3	90.3 %
step-lemma-12	29	12	58.6 %
new-fnd-is-fnd	29	6	79.3 %
b-eq-plus-new	28	3	89.2 %
st-notin-b-blist	27	5	81.4 %
blist-elem-from	25	2	92.0 %
b-st-not-new	21	3	85.7 %
t-call-term	15	3	80.0 %
bottom-notin-blist	13	1	92.3 %
st-in-blist	11	1	90.9 %
eval-equiv-b	4	4	0.0 %