Verification of Abstract State Machines

English translation
of the
Doctoral Thesis
to reach the degree of a doctor
of natural sciences (Dr. rer. nat.)
in the faculty of computer science
at the University of Ulm

written by

Gerhard Schellhorn
born in Balingen, Germany

Current Dean: Prof. Dr. Uwe Schöning

Reviewers: Prof. Dr. Wolfgang Reif (University of Ulm)
Prof. Dr. Hemuth Partsch (University of Ulm)
Prof. Dr. Egon Börger (University of Pisa)

Day of Defense: 9. Juni 1999

Acknowledgements

My heartfelt thanks goes to all who have supported me in doing this work.

First there is my doctor father Prof. Reif, whose advices and critical comments have always provided me with the motivation to continue with the work.

Then there are my colleagues Kurt Stenzel, Michael Balser and Andreas Thums, who have helped me with comments as well as with their willingness, to relieve me from other work. I want to emphasize the support of Kurt Stenzel, who has helped me with discussionsas well as with his work on improving the KIV system.

A sincere thanks also goes to the students, which have worked with me on the Polog-WAM case study. There is Wolfgang Ahrendt, whose diploma thesis was an important contribution. Harald Vogt, Christoph Duelli und Tobias Vollmer also have worked many hours on the proofs of the case study. The latter two I also want to thank for their work on proof reading.

Finally I want to thank my parents, my brother and sister, and my two nephews. They all have contributed in their one way to the successful outcome of this work.

Ulm, June 1999

Summary

The context of this work is the application of formal methods in software engineering. It is based on the specification language of abstract state machines (ASMs) defined in [Gur95].

In this work we develop tool support for ASMs, for their specification as well as for the verification of refinements. We want to make possible the development of correct software from a first abstract requirements specification to an implementation that is got by stepwise refinement. Our work consists of four parts.

- Embedding of ASM specifications in a logic: We define a 1:1 mapping of ASM specifications into Dynamic Logic (DL). This makes formal verification of ASM properties possible.
- Modularization of correctness proofs for refinements: Two refinement notions known from literature are formalized in DL. Generic modularization theorems for proving the correctness of refinements are developed, that generalize the theorems known from literature.
- Implementation of the results in the KIV system: The KIV system is a specification and verification tool, that supports algebraic specifications and DL. A number of extensions and improvements were necessary to support ASMs and ASM refinements.
- Demonstration of the practical applicability of the developed concepts in a large case study: The chosen case study from compiler construction treats the translation of Prolog programs into code of the Warren Abstract Machine (WAM). An informal presentation, that transforms a Prolog interpreter in 12 systematic refinements to the WAM was given in [BR95]. The formal specification and verification of 8 of the 12 refinements was a major part of this work. A comparison with two other case studies on the same topic showed, that the necessary verification effort was much smaller due to developed theory for ASM refinement.

Contents

Part I

Abstract State Machines and Correctness Proofs for Refinements

Chapter 1

Introduction

The context of this work is the application of formal methods in Software Engineering. The goal is the development of correct software for safety critical applications.

Application of formal methods presupposes a suitable *specification language* which abstractly and unambigously describes the requirements for the software to be developed. This makes them accessible to a mathematical analysis. *Validation by theorem proving* e.g. by verifying safety properties becomes possible already in the early phases of software development, where no implementation is available. Systematic transformation of abstract requirements to implemented code then requires a suitable notion of *refinement*.

Proofs for the validation of specifications and for the correctness of refinements are possible in various levels of detail, from informal proof sketches to fully formal proofs in a machine-supported calculus.

The goal of this work is to make the specification language of *Abstract State Machines* (in the following abbreviated as ASMs, [Gur95]) available in the specification and verification tool *KIV*.

The choice of the specification language is based on the fact, that there are two main families: The first are algebraic specification languages [Wir90], [Gau92], [CoF97] and their generalization to process algebras [Mil89], [Bae90]. These view a software system as a generalized data structure, with suitable functions and relations for modification. Mathematically a software system is modeled as an *Algebra*, a specification describes a class of algebras as possible implementations. A special case of algebraic specifications are model based specifications, in which a software system is built up from standard data types from set theory (like tuples, functions, power sets).

The second family of specification languages are state based languages, which model a system by a set of *states*, by possible state transitions and thereby resulting traces. Examples e.g. Z [Spi88], VDM [Jon90] and RAISE [JC94]. Abstract State Machines also belong to this family. To describe the components of a state state based specification languages are usually based on algebraic ones. In a sense state based specification languages can even be viewed as a special case of algebraic ones, since state transitions can be modeled as functions or relations on states. Therefore many verification tools support algebraic specification only. The disadvantage of this approach is, that the basic concepts of state based systems have to be modeled in an algebraic setting first.

Traditionally the KIV system supported the algebraic approach to software development. KIV allows to define structured algebraic specifications and offers appropriate proof support [RSSB98]. An elaborated refinement concept is available, which allows the structured, modular refinement of specifications by software modules [Rei95].

This work is a contribution to the realization of support for state based specifications in KIV. The choice of ASMs as the specification language was mainly due to the fact, that ASMs offer a conceptually simple, but very flexible approach to the specification of state based systems, that allows a wide variety of case studies. ASMs were already used successfully in a number of case studies, that dealt with such different topics as the semantics of programming languages (e.g. Prolog [BR94], C [GH93] and Java [BS98b]), communication protocols (e.g. Bakery algorithm

[BGR95]), compiler correctness (e.g. Occam [BD96], Prolog [BR95] and Java [BS98a], [Sch99]), distributed systems (e.g. PVM [BG95]) and hardware architectures (e.g. DLX [BM96]). An overview over a large number of applications can be found in [BH98] and also in the internet under the URLs *http://www.eecs.umich.edu/gasm/* and *http://www.uni-paderborn.de/cs/asm/*. In most case studies the correctness proofs were done as mathematical proofs, they were not supported by a verification system.

To support the formalism of ASMs described in Chap. 2, we first had to define an embedding in the specification language of KIV. Here, compared to purely algebraic specification systems, KIV has the advantage, that abstract programs over algebraic data types (which have state transitions as semantics) are already available. Therefore a first result of this work is the definition of a 1:1 translation of ASM rules to abstract programs. Chapter 3 describes the specification language and the logic used in KIV, and the extensions, which were done in the context of this work. Chapter 4 defines the translation.

Besides formal specification of ASM properties the embedding in KIV also offers the possibility to do formal, machine supported proofs in *Dynamic Logic*, the program logic KIV is based on. To complete the systematic support for ASMs, a refinement notion is defined in Chap. 5. It is shown, that correctness of refinements is expressible in DL.

The kernel of this work is the development of proof support for the modular verification of the correctness of refinements in Chap. 6. A general modularization theorem is developed first in its simplest form for the refinement of deterministic ASMs. Then several generalizations for indeterministic ASMs and for iterated refinement are given. We also give references to other correctness notions for refinements. The main result is a generalization of the known theory of refinements: Instead of using abstraction functions we use arbitrary relations, and instead of commuting diagrams with one rule of each ASM, we consider m:n diagrams with an arbitrary numbers m and n.

As an application of the theory, Chap. 7 shows that correctness of peephole optimizations can be derived as a corollary of the modularization theorem.

The theory defined in Chap. 6 has not been derived by theoretical considerations, how to generalize existing refinement notions. We believe, that there already exist too many concepts for the verification of software, that have nice theoretical properties, but no useful practical applications. Instead the flexibility of the modularization theorem and the quality of the proof support should be evaluated by its usefulness in practical applications. Therefore the theory was developed based on a realistic, large case study.

The chosen case study is the translation of Prolog to assembler code of the *Warren Abstract Machine* (WAM). There was already a mathematical analysis available [BR95], on which we could base our work. The case study showed a variety of problems in working with ASM refinements, especially in the application domain of compiler correctness. With 9 man months of work the case study belongs to the big and challenging works in this area. In the second part of this work we give a detailed presentation of the case study, in which we verified 8 of the 12 refinements given by [BR95].

The main result of the case study was the demonstration of the productiveness of the theory. This becomes clear when one considers two other case studies with other systems on the same topic, which needed substantially more effort to achieve smaller verification results. Currently the theory is also used in [Sch99] in the verification of a Java compiler.

The case study also shows what is gained by a machine checked proof in comparison to a mathematical analysis. We think, that the analysis in [BR95] is already a *very* careful and *detailed* one, and does not contain any conceptual errors. Nevertheless we were able to uncover numerous of small problems, that would have lead to an incorrect compiler. Therefore this work shows that it is worthwhile to invest the high cost of a formal, systematic verification if the application requires absolutely error free software (in this case an error free compiler).

Chapter 2

Abstract State Machines

Abstract State Machines (short ASMs) are a specification language to describe software and hardware systems. The basic idea of ASMs is the stepwise transformation of a state by executing rules. Therefore they belong to the family of specification languages, whose semantics is a state based system. State based systems are defined in the first section. Sect. 2.2 then gives the basic definition of sequential ASMs. A variant of this definition, which is used in the Prolog-WAM case study is explained in Sect. 2.2. Finally, Sect. 2.4 defines distributed ASMs, which are used to model distributed systems. A comprehensive presentation of ASMs, which gives additional concepts besides the basic ones defined here, can be found in [Gur95].

2.1 State Based Systems

The basic idea of a state based system is the transformation of states by rules. More formally a state based system $ZS = (S, I, \rho)$ consists of a set S of possible states, a set $I \subseteq S$ of initial states and a transition relation $\rho : S \times S$. $(st,st') \in \rho$ means, that st' is a possible successor state of st. A set F of final states can be fixed as the set of those states which have no successor state. State based systems are often chosen as a natural formalization of software systems, since the typical computation of a computer with a von-Neumann-architecture involves the state of a memory, that is modified by a processor (which defines the state transition relation). Other examples are finite automata (the set of states the is the set of all strings over an alphabet), Rewrite systems (where a state is a term), communication protocols and interpreters of programming languages. Even mathematical concepts like the derivation notion of logical calculi can be described as state based systems.

An special case of state based systems that is often used are sequential (or deterministic) systems, in which every state st has at most one successor state st' with $(st,st') \in \rho$. For this case a state transition function τ can be defined on all non-final states $(S \setminus F)$ by $\tau(st) = st'$ iff $(st,st') \in \rho$.

For a state based system the set of possible traces can be defined as the set of all finite $(st_0, \ldots, st_n)$ and infinite sequences $(st_0, st_1, \ldots)$ of states with $(st_i,st_{i+1}) \in \rho$ for every i. A trace is required to start in an initial state $st_0 \in I$ and, when finite to end in a final state $st_n \in F$.

2.2 Sequential ASMs

ASMs ([Gur95]) are a formalism to define state based systems. The set of all possible states is given as the class of all possible algebras $Alg(SIG)$ over a (one-sorted) signature SIG. To allow the definition of boolean expressions and partiality, it is assumed that the signature always contains the usual boolean operations (tt, ff, $\wedge$, $\vee$, etc.) as well as a constant $undef$.

The set of initial algebras I is usually given by a set-theoretic description of algebras or an algebraic specification. The transition relation is given by a rule R. For sequential ASMs of this

section rules are defined inductively as follows:

1. $f(\underline{t}) := t'$ is a rule, for every n-ary function symbol f ($n \geq 0$), and ground terms $\underline{t}$ and t'. The rule modifies the value of f at the arguments $\underline{t}$ to be t' .
2. If $R_1, \ldots, R_n$ are rules, then so is their parallel execution $(R_1, \ldots, R_n)$
3. If $R_1, \ldots, R_n$ are rules, and $\varepsilon_1, \ldots \varepsilon_n$, are boolean expressions, then so is the conditional rule
   (**if** ε_1 **then** R_1 **else if** ε_2 **then** R_2 **else** ... **if** ε_n **then** R_n)

The semantics of a rule R is a transition function, that given an algebra $\mathcal{A}$ delivers a new algebra $\mathcal{B}$. $\mathcal{B}$ is defined with the help of a finite set of *updates* $Upd(R, \mathcal{A}) = \{(f_1,\underline{a}_1,b_1), \ldots (f_n,\underline{a}_n,b_n)\}$, which is are computed from the rule R and the algebra $\mathcal{A}$.

Each update $(f, \underline{a}, b)$ consists of an n-ary function symbol f, and values $\underline{a}, b \in A^{n+1}$ over the carrier (the universe) A of the algebra $\mathcal{A}$. Corresponding to the structure of rules the set of updates is defined by

1. $Upd(f(\underline{t}) := t', \mathcal{A}) = \{(f, \underline{t}_{\mathcal{A}}, t'_{\mathcal{A}})\}$
2. $Upd((R_1, \ldots, R_n), \mathcal{A}) = Upd(R_1) \cup \ldots \cup Upd(R_n)$
3. $Upd($**if** ε_1 **then** R_1 **else** ... **else if** ε_n **then** $R_n) = Upd(R_k)$,
   where k is minimal with $\mathcal{A} \models \varepsilon_k$. If for all $k = 1,\ldots, n$ $\mathcal{A} \not\models \varepsilon_k$ holds, then $Upd($**if** ... $) = \emptyset$.

The set $Upd(R, \mathcal{A})$ is *inconsistent*, if it contains several elements $(f, \underline{a}, b)$ with the same function f and vector $\underline{a}$. In this case the transition function is identity, i.e. $\tau(\mathcal{A}) = \mathcal{A}$. If $Upd(R, \mathcal{A}) = \emptyset$, then $\mathcal{A}$ is a final state[1]. If $Upd(R, \mathcal{A})$ is consistent and nonempty, $\mathcal{B}$ has the same carrier as $\mathcal{A}$ and the semantics of its functions is defined by

$$f_{\mathcal{B}}(\underline{a}) = \begin{cases} b & \text{if } (f,\underline{a},b) \in Upd(R,\mathcal{A}) \\ f_{\mathcal{A}}(\underline{a}) & \text{otherwise.} \end{cases}$$

For every ASM operations can be partitioned into two disjoint sets: A set of *dynamic* functions, which occur on the left hand side of an assignment in a rule, and the complementary set of *static* functions, which are never changed during the run of the ASM.

Static functions are used, to model operations on data structures (like + on natural numbers, or append on lists). Of course it is required, that the boolean operations are static.

0-ary dynamic operations (for obvious reasons, we do not call them 'constants') are used as "program variables". Dynamic functions with arguments are often used to model memory. Application of a dynamic function at a results in the content $f(a)$ of memory f at address (or location) a. Modification of the function f at address a means to overwrite the memory location. A dynamic function with finite domain G can also be viewed as an abstract form of an array with indexes in G.

Sorts are modeled in ASMs as unary predicates. To have an addition operation which adds a new element to the carrier of a sort, often the following extension is used: It is assumed, that there is a predefined sort *reserve* (i.e. a unary predicate) that has an infinite carrier ("reserve elements") in every initial state. The new rule construct

import x **in** R **endimport**

then allows to remove an element from *reserve*, to bind it to the variable x and to execute rule R with this binding. Addition of an element to a sort S then can be achieved with

import x **in** S(x) := tt; R **endimport**

[1] [Gur95] does not define final states for sequential ASMs. We add the definition here, since we need final states for the definition of ASM refinements.

This is abbreviated as

extend S **with** x **in** R **endextend**

We do not give a precise definition of this extension, since it has some pitfalls and causes a lot of technical overhead (rules can now use the local variable R, nested imports must return new elements *sequentially*). A precise definition can be found in [Gur95].

2.3 Sequential ASMs in the WAM

The ASMs of the Prolog-WAM case study in [BR95] use a variant of the definition of sequential ASMs. In this variant rules must have the simpler form

if ε **then** $(f_1(\underline{t}_1) := t_1', f_2(\underline{t}_2) := t_2', \ldots, f_n(\underline{t}_n) := t_n')$

Instead of one rule every ASM now has a set of such simpler rules. A state transition consists in the indeterministic choice of a rule, which has a test (often called guard) ε that is true, and the execution of its updates. If all rule tests mutually exclude each other, then such a rule set is obviously equivalent to a nested conditional rule of the previous section (with an arbitrary order of the rules). For the Prolog-WAM case study the mutual exclusion of rule tests was intended (for a case, where the intention was not met, see Sect. 12.2), so we do not need to consider the problem of indeterminism here.

2.4 Distributed ASMs

The basic idea of a distributed ASM also is the modification of a state by rules[2]. But instead of a single rule a distributed ASM has a finite set A of (active) *agents*, where each of the agents has one rule of a finite set of rules $\mathcal{R}$ attached (the attached rule is the program, that the agent currently runs). One state transition then consists in the selection of one agent $a \in A$, and the execution of the rule attached to it. Rules in distributed, indeterministic ASMs can change the set of the active agents as well as the rule attached to each agent.

To formally define these concepts a distributed ASM contains a set N of *rule names*, i.e. static constants ν, which denote rules. For a rule name ν, R_ν is the corresponding rule. The signature also contains a (dynamic) function *Rule*, which maps agents to rule names. The set of active agents is given implicitly as the set of elements, for which $Rule(a) \in N$ holds.

The set of possible states of a distributed ASM is restricted to such algebras, in which rule names denote *different* constants, and in which the set of agents is finite.

Finally, compared to the definition of rules for sequential ASMs, there is one extension: all rules may use the symbol *Self* for the actually chosen agent. If a rule R is executed by an agent a, then in the computation of $Upd(R, \mathcal{A})$ the symbol *Self* is interpreted as a. In this way rules can be parameterized with the agent executing them. If an agent e.g. executes the assignment

Rule(Self) := undef

then it terminates its computation. A distributed ASM reaches a final state when the set of agents becomes empty.

[2] we assume the semantics defined as that of 'Sequential Runs'. [Gur95] gives other possible definitions.

Chapter 3

Dynamic Logic and Algebraic Specifications

3.1 Dynamic Logic

Dynamic Logic (DL) is an extension of first-order logic by program formulas of the form $\langle\alpha\rangle\ \varphi$ and $[\alpha]\ \varphi$. Here, α is an imperative program and φ is again a formula of DL. Programs contain the usual constructs like parallel assignment $\underline{x} := \underline{t}$, sequential composition $\alpha;\beta$, conditional **if** ε **then** α **else** β, while loop **while** ε **do** α and procedure call $p(\underline{t};\underline{x})$ with value Parameters $\underline{t}$ and reference parameters $\underline{x}$. For theoretic reasons we also have the program **skip**, that does nothing, the never terminating program **abort**, i-fold iteration **loop** α **times** i, random assignment $x :=?$ and a procedure call **procbound** i **in** $p(\underline{t};\underline{x})$ with a bound i on the recursion depth (if the bound is exceeded the call does not terminate).

The semantics of programs $[\![\alpha]\!]$ is defined as a binary relation on states, i.e. valuations in the usual sense of first-order logic. For a deterministic program the relation is a partial function, i.e. for every valuation $\mathbf{z}$ there is at most one $\mathbf{z}'$, such that $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ holds. The only indeterministic program construct is random assignment: $\mathbf{z}[\![x :=?]\!]\mathbf{z}'$ holds for all $\mathbf{z}' = \mathbf{z}[x \leftarrow a]$, which result from a modification of the value of x by an arbitrary a.

The program formula $\langle\alpha\rangle\ \varphi$ holds in a state $\mathbf{z}$, if there is a state $\mathbf{z}'$ with $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ and φ holds in $\mathbf{z}'$. Dual to this definition $[\alpha]\ \varphi$ holds in a state if in every state $\mathbf{z}'$ with $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ the formula φ holds.

The program formula $\langle\alpha\rangle\ \varphi$ therefore means, that there is a terminating run of α, such that afterwards φ holds. $[\alpha]\ \varphi$ holds, if φ holds after every terminating run of α. $\varphi \rightarrow [\alpha]\ \psi$ resp. $\varphi \rightarrow \langle\alpha\rangle\ \psi$ express partial resp. total correctness with respect to precondition φ and post-condition ψ.

Syntax and semantics of DL are precisely defined in appendix B. Note, that a many-sorted logic is used, that defines expressions only and does not distinguish between formulas and terms. Formulas are identified with expressions of sort bool. This has the advantage, that by adding lambda expressions the logic can easily be extended to a higher-order logic. A technical advantage is that a general if–then–else Operator ($\varphi \supset t_1; t_2$) is available (φ a formula, $t_1; t_2$ two arbitrary expressions of the same sort). The expression is equal to t_1, if φ is true, and equal to t_2 otherwise.

3.2 Algebraic Specifications

We will use algebraic speicifications with the structuring operations union (+), enrichment, renaming, parameterization (generic specifications), and actualization. For freely generated data types we will use datatype declarations (see e.g. lists as defined in appendix E), which automatically generate apropriate axioms. The syntax should be self-explanatory, the semantics of the structur-

ing operations is defined as usual. It is e.g. almost identical to the definition of the semantics of the standard specification language CASL [CoF97].

In basic specifications we will allow as axioms not only first-order formulas, but also arbitrary DL formulas, generation principles and procedure declarations. The semantics of basic specifications is the class of all models of the axioms (loose semantics). A precise definition is given at the end of appendix B.

3.3 KIV

KIV is a system for the development of correct software. The specification language supported by KIV are structured, algebraic first-order specifications. The software development methodology used until now was based on structured, modular refinement of such specifications by program modules. Their correctness can be expressed by proof obligations in DL. This methodology is comprehensively presented in [Rei95]. The verification of program modules is discussed in [RSS95].

Deduction support in KIV is based on a sequent calculus for Dynamic Logic. An overview of the support for deduction over algebraic specification is given in [RSSB98].

3.4 Improvement of Proof Strategies

In the context of this work the KIV system was improved in a number of ways, particularly in the deduction component. These improvements were important for the efficient verification of ASM refinements, especially in the Prolog-WAM case study (see also the statistics in section 19). This section gives a short listing of the items improved:

- extension of the specification language from structured first-order to structured DL specifications with *global* procedure declarations (instead of local ones). Global procedure declarations make the global definition of ASMs possible.
- Removal of the distinction between terms and formulas, thereby identifying formulas with boolean terms. This modification allows to use boolean dynamic functions (boolean predicates) like all other dynamic functions. This modification also allows (independent of this work) to easily extend DL with higher-order functions by adding λ-terms.
- The proof strategy for programs now can handle parallel assignments. These were supported by the logic, but not by the prover.
- Addition of an induction principle over the recursion depth for procedures. This proof principle simplifies the previously defined proof principle (Induction over environments, see [Ste85]) for recursively defined procedures. The new proof principle was a key concept to verify properties of the *CHAIN#* procedure in the Prolog-WAM case study (see Sect. 15.2). It also simpifies the definition of the semantics and the completeness proof for DL.
- Extension of the tactics and heuristics for while loops, and the loop construct, which both play a central role in the proofs of the proof obligations for the correctness of ASM refinements.
- Extensions of several other heuristics, e.g. the heuristics for unfolding procedures and for quantifier instantiation.
- Implementation of an efficient simplification strategy (see [RSSB98]). The current implementation can deal with the 2000 simplification rules, which occurred in the Prolog-WAM case study.
- Several other efficiency improvements, that became necessary simply by the size of the goals that were to prove. In some case sequents in the Prolog-WAM case study reached the size of 5 screen pages, and proof trees had up to 1000 nodes.

Chapter 4

Formalization of ASMs in DL

This chapter starts with the definition of a translation of ASMs to algebraic specifications and Dynamic Logic (DL). The translation will be essentially one to one, since the basic constructs of both ASMs and DL are assignments. Since there is no need to formalize the semantics of ASMs, i.e. to encode ASM rules as relations over states, DL is a good starting point for the verification of ASM properties. The translation consists of three steps: In the first step (Sect. 4.1) we will show, that algebras, which are used as ASM states can be transformed into valuations over a suitable algebraic specification. The second step (Sect. 4.2) then translates ASM rules to imperative programs, using the valuations of step one as intermediate states of the program.

Sections 4.3 and 4.4 then consider the third step, the translation of sequential resp. distributed ASMs into an imperative program.

The main proof principle for ASMs is induction over the number of executed rules. Section 4.5 shows, how this proof principle is formalized in DL.

In Sect. 4.6 we finally discuss alternatives to our approach of translating ASMs to DL.

4.1 Translation of Specifications

To translate the abstract data types of an ASMs to algebraic specification, we first have to partition the signature into a *static* and a *dynamic* part. The dynamic part contains those sorts and operations, which are modified by assignments of the ASM. The static part typically contains data types like list, number with suitable operations. For this part no translation is necessary; it simply has to be specified algebraically.

The main idea for the translation of the dynamic part is, to encode the semantics of dynamic functions as values of (usual first-order) variables. Assignments of the ASM thereby become assignments in DL.

0-ary functions are simply translated to first-order variables. The case of a function with several arguments can be reduced to the case with one argument by adding a suitable tuple sort. For functions with one argument we have to encode the (second-order) data type of a function into a first-order data type, to make values of the datatype available as the values of variables. This can be accomplished with the datatype shown in Fig. 4.1, which specifies functions from a domain *dom* to a codomain *codom*:

The data type defines a constant function $cf(z)$ for every element z of the codomain. Application of this function to any element x of the domain always gives z, as stated by the first axiom. The (binary) operation "function application of f to x" is written (using mixfix-notation) as $f[x]$ (note that now f is a variable of sort *dynfun*, not a function symbol!). With a suitable constant z of the codomain constant functions are typically used as initial values for dynamic functions.

An assignment $f(x) := t$ of the ASM formalism is translated to the algebraic setting as an assignment $f := f[x \leftarrow t]$ to the variable f. The new function value, which is the old modified at x

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 cf(z) [x] = z,
f [x ← z] [x] = z,
x ≠ y → f [x ← z] [y] = f[y]
end generic specification

Figure 4.1 Specification of Dynamic Functions

by t we again use the mixfix-notation $f[x \leftarrow t]$ (instead of "modify(f,x,t)"). The last two axioms describe its behavior.

It should be noted, that (in contrast to the usual methodology used in KIV when specifying non-free datatypes) it was not necessary to define an extensionality axiom

$$f = g \leftrightarrow \forall x.\ f[x] = g[x]$$

in the specification of dynamic functions. Such an axiom would have allowed to deduce equations between functions like $f = f[x \leftarrow f[x]]$. Since such equations are not part of the ASM formalism, they are not needed for the translation either. For the same reason we could avoid to define an induction principle for dynamic functions (e.g. structural induction over *cf* and *modify*).

It is easy to see, that the set of all functions from *dom* to *codom* is a model of the specification given above. For this model we have the 1:1 correspondence between dynamic functions and valuations of the corresponding variables in the translation.

The basic form of the translation gives an algebraic specification, in which neither the possibilities to use underspecification nor the existence of sorts (except to define tuple and function sorts) has been exploited. This can be improved by using sorts instead of sort predicates wherever possible in the algebraic translation. Underspecification can be used to avoid the use of an explicit error element *undef*.

An important role in the translation of sorts is played by the predicate *reserve* in the ASMs, which defines an infinite set of "reserve elements". Of course it is possible to treat the *reserve* predicate like all other dynamic functions, and to translate it into a boolean dynamic function. For the *import* construct ([Gur95], Sect. 3.2) then a function *some(reserve)* has to be defined, which given the current value of *reserve* delivers an element x with $reserve[x] = tt$. But typically elements of the *reserve* carrier are used only to dynamically add them to the carrier of some other sort (e.g. to increase the set of nodes of a search tree or to allocate a new address in memory). In this case, which uses the abbreviation

extend s **with** x **in** R **endextend**,

to move one element from the *reserve* carrier to the one for sort s, there is a much simpler translation, which avoids to use "reserve elements" completely. To define it, we will encode the current elements of sort s as the valuation of a variable se of sort set (with elements of sort s). To specify such sets usually the specification of finite sets from Fig. 4.2 can be used, since in most cases the used carrier sets will be finite (if the initial carrier set of an ASM is infinite, a suitable constant has to be added). The carrier set of s now contains the infinitely many potential

elements, that can be inserted into the dynamic set *se*. Function *new*(*se*) gives a new element relative to *se*. The sort update above therefore can be expressed in the translation as

var x = new(se) **in begin** se := se ∪ {x}; R **end**

Set =
generic specification
parameter S;
target
sorts set;
constants ∅ : set;
functions
{ . } : s → set;
. ∪ . : set × set → set;
new : set → s;
predicates
. ∈ . : s × set;
variables se, se_1, se_2 : set; x, y :s;
axioms
set **generated by** ∅, { . }, ∪;
¬ x ∈ ∅, x ∈ {y} ↔ x = y,
x ∈ se_1 ∪ se_2 ↔ x ∈ se_1 ∨ x ∈ se_2,
se_1 = se_2 ↔ (∀ x. x ∈ se_1 ↔ x ∈ se_2),
¬ new(se) ∈ se
end generic specification

Figure 4.2 Algebraic Specification of Sets

4.2 Translation of ASM Rules

In this section we will define the translation of ASM rules into (flat) DL programs. It is sufficient to translate condition rule, whose bodies are sequences of update instructions,

if ε_1 **then** U_1 **else**
if ε_2 **then** U_2 **else**
⋮
if ε_n **then** U_n

since iterated application of the transformation

(R, **if** ε **then** R′ **else** R″) ⇒ **if** ε **then** (R,R′) **else** (R,R″)

will bring every rule into this form.

The conditional is unchanged by the translation[1], the translation of a single assignment $f(t) := t'$ to $f := f[t \leftarrow t']$ was already discussed in the previous section. For parallel assignments with several updates of the same function, we must take the possibility of inconsistent updates into account. This is done using additional checks. As an example, $f(x) := t, f(x') := t'$ must be

[1] note, that in DL **if** ε_n **then** U_n is an abbreviation for **if** ε_n **then** U_n **else skip**

translated to **if** $x = x' \wedge \mathrm{t} \neq \mathrm{t}'$ **then skip else** $f := f[x \leftarrow t][x' \leftarrow t']$ (in most cases the inconsistency checks can be simplified using the preconditions of the case under consideration, often they can be completely dropped). With the additional checks inconsistency leads to no state change, as required by the definition of ASM semantics. To improve readability we will write $f[x] := t$ instead of $f := f[x \leftarrow t]$ in DL programs.

4.3 Translation of Sequential ASMs

To simplify the presentation, we will assume in the rest of this work, that the test, if any ASM rule is applicable can be decided using a predicate *final* (*final* is simply the conjunction of all negated rule tests). Then the result of the translation is the following procedure:

ASM(var $\underline{\mathrm{x}}$)
begin
while $\neg$ final($\underline{\mathrm{x}}$) **do** RULE(;$\underline{\mathrm{x}}$)
end

The allowed initial states of the ASM are given by suitable initial valuations of the variables $\underline{x}$. The variables $\underline{x}$ are used as input and output. They store the valuations of all dynamic functions. Iterated application of rules is done with a while loop. procedure *RULE* contains the translated code of the ASM rule (the semicolon before the variables $\underline{x}$ in the call indicates reference parameters). A separate procedure was defined simply to have a suitable abbreviation in the following.

The equivalence of the *while* program to the definition of the ASM semantics is given by considering the sequences of states, through which the program runs at the beginning of the while loop. The possible sequences are (modulo the translation of algebras to valuations) exactly the same as in the ASM. A restriction of the expressiveness of DL is only, that we are not able to talk directly about these *sequences* of states and their properties. This would require either the introduction of operators similar to temporal logic, or the definition of a data type of streams to encode the sequences. In main topic of this work, ASM refinements, the explicit representation of traces will be mostly sufficient. In particular, traces of states will not occur in the proof obligations for refinement correctness. Only for ASMs with unbounded indeterminism (Sect. 6.4) we will need the temporal logic operator AF, and in the definition of trace correctness in Sect. 6.3 we will make use of a formalization of streams as (dynamic) functions from natural numbers to states.

4.4 Translation of Distributed ASMs

The main problem in the translation of distributed ASMs is the indeterministic choice of an agent a from a finite set A of candidates. Although the finite set A can be described using the datatype of finite sets from Sect. 4.1, it is not possible to use an additional function *some*, since for a set s such a function would always deliver *the same* element *some*(s). Nevertheless a solution in DL is easy: One uses a procedure *SOME*, that has the current set of active agents as input and returns the agent *Self*, which should execute a rule. *Self* is now a program variable. For the procedure *SOME* only the axioms

$$\mathrm{a} \in \mathrm{A} \rightarrow \langle \mathrm{SOME(A;Self)} \rangle \ \mathrm{Self} = \mathrm{a} \tag{4.1}$$

and

$$[\mathrm{SOME(A;Self)}] \ \mathrm{Self} \in \mathrm{A}$$

are needed. They say, that the input/output relation of *SOME* is in all models of the specification equal to the element relation (the first axioms says superset, the second subset). This means,that every time each choice of an agent from the set is possible, corresponding to the definition of the ASM semantics. An implementation of the procedure *SOME* would be a scheduler for agents. Such an implementation will usually have a strategy for choosing the next agent and therefore not be fully indeterministic. It will often also depend on other state components. Therefore, to support arbitrary schedulers, *SOME* can be called with the complete state $\underline{x}$ of the ASM and the axiom (4.1) can be replaced by the weaker totality axiom

$$A \neq \emptyset \rightarrow \langle \text{SOME}(\underline{x};\text{Self}) \rangle \text{ true}$$

Then, the input/output relation of a scheduler is only required to be a total subrelation of the element relation. This makes it possible to relate different schedulers in ASM refinements (see Chap. 5), e.g. by stating that every choice of a concrete scheduler should be possible by the abstract one too). It should be noted, that restrictions such as fairness constraints will probably make it necessary to talk about the sequence of selected *Self* values. To do this will require extensions of Dynamic Logic or the explicit use of streams (see also the translation of linear temporal logic (LTL) discussed in [Vog97]).

Using the *SOME* procedure the distributed ASM is translated to

ASM(var $\underline{x}$)
begin
while A $\neq \emptyset$ **do**
  begin
  SOME($\underline{x}$;Self);
  if Rule(Self) $= \nu_1$ **then** RULE$_1$(;$\underline{x}$) **else**
  if Rule(Self) $= \nu_2$ **then** RULE$_2$(;$\underline{x}$) **else**
  $\vdots$
  if Rule(Self) $= \nu_n$ **then** RULE$_n$ (;$\underline{x}$)
  end
end

where the rules $RULE_1$, $RULE_2$, ..., $RULE_n$ are translated as for sequential ASMs. Note, that the currently selected agent *Self*, the set of active agents A and the dynamic function *Rule*, which gives the rule name for an agent are all part of the vector of program variables. The rule names are specified as an enumeration type with values $\nu_1, \ldots \nu_n$.

Like in the sequential case the possible sequences of states at the beginning of the while loop coincide with the possible traces of the ASM (modulo encoding algebras as valuations). To have a uniform notation for sequential and distributed ASMs, we will also write $RULE(;\underline{x})$ for the body of the while loop, and we will use the general test $\mathit{final}(\underline{x})$ instead of the special $A \neq \emptyset$ used here.

4.5 Rule Induction in DL

The main proof principle to reason about ASMs that we will use in the following is induction over the number of executed rules ("rule induction"). In this section we give the formal corresponding proof principle in DL, induction on the number of while loop iterations. Induction on this number is possible using the Omega-Axiom of Dynamic Logic:

$$\langle \textbf{while } \varepsilon \textbf{ do } \alpha \rangle \, \varphi \leftrightarrow \exists \text{ i. } \langle \textbf{loop if } \varepsilon \textbf{ then } \alpha \textbf{ times i} \rangle \, (\varphi \wedge \neg \, \varepsilon) \tag{4.2}$$

In this axiom i is a natural number (which can be used for induction) that counts the number of loop iterations. The loop program **loop** α **times** i executes its body α i times. Axiom (4.2) therefore states, that a formula φ holds after the execution of a while loop, if and only if there is a number i chosen sufficiently large, such that after iterating **if** ε **then** α this often φ holds and the test ε of the while loop is false. Note that for some fixed initial state the value of i need not be chosen as the *exact* number of times the while loop will be iterated when starting from this state. Any number greater than this number will also be sufficient, since executing **if** ε **then** α when ε is already false has no effect. This gives some extra degree of freedom in proofs where only some properties of the initial state are known (replacing **if** ε **then** α in the body of the loop construct by **if** ε **then** α **else abort** gives the more restrictive variant, where i must be the exact number of iterations).

The loop construct is defined in DL recursively by the two axioms:

$$\begin{array}{l} \langle \textbf{loop } \alpha \textbf{ times } 0 \rangle \ \varphi \leftrightarrow \varphi \\ \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i} +1 \rangle \ \varphi \leftrightarrow \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i} \rangle \ \langle \alpha \rangle \ \varphi \end{array} \tag{4.3}$$

4.6 Alternatives to our Formalization

The translation of ASMs to DL is not the only alternative to realize deduction support for ASMs. Several others are possible:

1. Embedding ASMs in a higher-order variant of Dynamic Logic.
2. Definition of an "ASM logic": Such a logic must support the modification of algebras by programs. A suitable candidate would be MLCM (modal logic if creation and modification [GdL94],[GR95]). [Sch95] is an attempt, to implement a variant of MLCM in the KIV system.
3. Instead of formalizing ASMs, their semantics, i.e. state based systems can be formalized algebraically. This is possible with first-order logic and was done for the Prolog-WAM case study in Isabelle [Pus96] (the formalization used higher-order logic, but this was not compulsory). ASM rules are replaced with an explicit description of the state transition relation, and an inductive definition of the relation between input and output states.
4. Embedding ASMs in temporal logic

The first solution is a variant of our solution, which replaces the datatype 'dynamic function' by second-order functions. The solution requires to extend DL with higher-order expressions (such an extension is currently planned). The solution would have the advantage, that the special 'apply' operation could be replaced with the usual function application. An argument for the current solution is, that it does not mix dynamic functions with general higher-order functions. The first are usually used as global registers and can be destructively overwritten while the other usually may not be modified destructively. Separation of the two cases could therefore ease efficient implementation.

The second solution is also similar to our solution. From our viewpoint it has the disadvantage, that the definition of a new logic requires much more effort: In addition to the implementation of new tactics and the definition of a new semantics also a correctness and completeness proofs for the new logic has to be done. Note also, that the correctness proofs for ASM refinements sometimes make it necessary to quantify over dynamic functions (for an example see Sect. 11.2), which is impossible in MLCM.

The third solution is much more different from ours, since it requires to develop a general theory of inductive relations (or an even more general fixpoint theory as it was done in PVS [BDvH$^+$96]), to make induction over the number of executed rules possible. Such a theory was

defined e.g. in Isabelle ([Pau94]). In our approach such a theory is already present in the axioms for while loops (compare to the previous Sect. 4.5).

For practical applications the solution has the disadvantage that every modification of the state must refer to the *whole* state (this is known as the "frame problem"). An assignment

$$x_i := f(\underline{y})$$

to a single component x_i of the state must be replaced by a relation $\Rightarrow$ (written infix)

$$(x_1, \ldots, x_i, \ldots x_n) \Rightarrow (x_1, \ldots, f(\underline{y}), \ldots x_n)$$

in which the whole state $(x_1, \ldots, x_n)$ is mentioned, causing notational overhead. Also adding a new component to the state will require to change all existing proofs, even if they do not consider the new component.

For the generic definition and the proof of the modularization theorem for ASM refinements, that will be done in Chap. 6, the frame problem is irrelevant, since in the theorem states will be considered as an unspecified, monolithic parameter sort. We will therefore have a short look on the first-order formalization of the theorem in Sect. 6.2.5.

An advantage of using inductive relations against ASMs is that they (like DL programs) allow arbitrary recursion. Arbitrary recursion for ASMs requires to extend the basic formalism (see [GS97]).

The fourth solution, embedding ASMs in a temporal logic (like CTL*) is a good alternative, when properties of single ASMs are considered. But relations between ASMs (like refinement) require to consider several state transition relations at one, which make an encoding more difficult (or require the use of a multimodal temporal logic).

Finally it should be noted, that instead of transforming the rules of an ASMs to a normal form (Sect. 4.1) a general operator for parallel execution of programs could be added. The transformation to normal form then can be described by rules *in* the logic. [Sch95] shows, how this possibility can be realized for MLCM. We currently prefer the transformation, since it is more efficient and we currently see no way to avoid it: inconsistency of a rule can be detected easily only, when the rule is in normal form.

Chapter 5

Refinement of ASMs and Formalization in DL

A refinement of one ASM $= (S, I, \rho)$ to another ASM$' = (S', I', \rho')$ is given by a relation $IN : I \times I'$ on initial states and a relation $OUT : F \times F'$ on the final states F and F'. Often special cases are considered, where functions instead of general relations IN and OUT are given.

Definition 1 *correctness and completeness of refinements*
A refinement of ASM to ASM' is *correct*, if for every finite trace $(st'_0, \ldots, st'_n)$ of ASM$'$ (with $st'_n \in F'$) and every st_0 of ASM with $IN(st_0, st'_0)$ there exists a trace finite trace $(st_0, \ldots, st_m)$ of ASM with $st_m \in F$ and $OUT(st_m, st'_n)$. We will write ASM $\triangleright$ ASM$'$ for a correct refinement. A refinement from ASM to ASM$'$ is *complete*, short ASM $\triangleleft$ ASM$'$, iff the refinement from ASM$'$ to ASM is correct

Correctness and completeness of a refinement is often expressed as the commutativity of the diagram in Fig. 5.1:

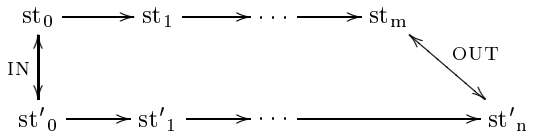


Figure 5.1 : Diagrammatic Visualization of an ASM Refinement

Correctness and completeness can be defined relative to one algebra, or relative to all models of the common specifications of both ASMs. The proof obligations, that we will derive in the following chapter will imply the correctness resp. completeness in every single model of the common specification (this is stronger than "if the proof obligations hold in every model, then we have correctness resp. completeness"), therefore the distinction is unimportant in the following.

The notions of 'correctness' and 'completeness' are drawn from ASM terminology ([BR95]). In the literature several other terms are used: In the Verifix project ([GDG+96]) they are called 'preservation of partial correctness' and 'preservation of total correctness'. A correct and complete refinement is sometimes called a 'Bisimulation'. In case studies with the NQTHM system ([BHMY89]) the notion 'interpreter equivalence' is used.

Our correctness notion compares the input/output behavior of the ASMs. It is adequate for ASMs, whose purpose is the "computation of a result". If an ASM describes a reactive system, there is another correctness notion, which compares traces of both ASMs. We will postpone the definition of such a notion ("trace correctness") until Sect. 6.3 where we will show that the proof obligations for both correctness notions differ only marginally.

5.1 Compiler Verification

A typical example where ASM refinements are used is compilation of programming languages. Two ASMs are considered, where the first is an interpreter for the source language and the other is an interpreter for the target language. Initial states store the source and target code of the program that should be executed. The *IN* relation between the initial states is given by a function *compile* which does the compilation:

$$\mathrm{IN}(\mathrm{st},\mathrm{st}') \leftrightarrow \mathrm{program}'(\mathrm{st}') = \mathrm{compile}(\mathrm{program}(\mathrm{st})) \wedge \mathrm{I}(\mathrm{st}) \wedge \mathrm{I}'(\mathrm{st}').$$

Usually an initial states is fixed uniquely by a given program that should be interpreted. Sometimes the weaker condition, that for every initial ASM′ state st' there is an ASM state st with $IN(st,st')$, is required.

For the output relation OUT it is usually required, that it should be possible to recover the (abstract) result of the source code interpreter by applying an abstraction function to the result of the target code interpreter.

$$\mathrm{OUT}(\mathrm{st},\mathrm{st}') \leftrightarrow \mathrm{result}(\mathrm{st}) = \mathrm{abstract}(\mathrm{result}'(\mathrm{st}')).$$

5.2 Formalization of Correctness in DL

The Correctness of a refinement from ASM to ASM′ can be expressed in DL as

$$\begin{array}{l}\mathrm{ASM} \triangleright \mathrm{ASM}' \equiv \\ \mathrm{IN}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \wedge \langle \mathrm{ASM}'(;\underline{\mathrm{x}}')\rangle \underline{\mathrm{x}}' = \underline{\mathrm{x}}'_0 \rightarrow \langle \mathrm{ASM}(;\underline{\mathrm{x}})\rangle\ \mathrm{OUT}(\underline{\mathrm{x}},\underline{\mathrm{x}}'_0)\end{array} \tag{5.1}$$

In the formula $\underline{x}$ and $\underline{x}'$ are two disjoint vectors of variables that result from the translation of dynamic functions from both ASMs. The formula states that $IN(\underline{x},\underline{x}')$ and the existence of a terminating run of ASM′ with result $\underline{x}'_0$ imply the existence of a terminating run of ASM, such that relation OUT holds for $\underline{x}'_0$ and its result (note that the $\underline{x}$ in $IN(\underline{x},\underline{x}')$ denotes an arbitrary initial value of the variables, while the $\underline{x}$ in $OUT(\underline{x},\underline{x}'_0)$ denotes the valuation of the variable *after* the execution of ASM).

For the formalization of completeness simply the roles of ASM and ASM′ are switched:

$$\begin{array}{l}\mathrm{ASM} \triangleleft \mathrm{ASM}' \equiv \\ \mathrm{IN}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \wedge \langle \mathrm{ASM}(;\underline{\mathrm{x}})\rangle \underline{\mathrm{x}} = \underline{\mathrm{x}}_0 \rightarrow \langle \mathrm{ASM}'(;\underline{\mathrm{x}}')\rangle\ \mathrm{OUT}(\underline{\mathrm{x}}_0,\underline{\mathrm{x}}')\end{array} \tag{5.2}$$

The equivalence of ASM and ASM′ then is the conjunction of (5.1) and (5.2). If the state vectors of both ASMs have the same types, and if $OUT(\underline{x},\underline{x}')$ is defined as $\underline{x} = \underline{x}'$, this conjunction can be simplified to the program equivalence

$$\begin{array}{l}\mathrm{ASM} \bowtie \mathrm{ASM}' \equiv \\ \mathrm{IN}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \rightarrow (\langle \mathrm{ASM}(;\underline{\mathrm{x}})\rangle \underline{\mathrm{x}} = \underline{\mathrm{x}}_0 \leftrightarrow \langle \mathrm{ASM}'(;\underline{\mathrm{x}}')\rangle \underline{\mathrm{x}}' = \underline{\mathrm{x}}_0)\end{array}$$

Chapter 6

A Generic Proof Method for ASM Refinements

This chapter is the kernel of the theoretical work. It is shown, that the correctness and completeness proofs for a refinement from ASM to ASM′ can be modularized. The proof obligations that guarantee the correctness of the modularization were formulated in Dynamic Logic, and were verified with the KIV system.

The first two sections consider sequential, deterministic ASMs. For introduction, Sect. 6.1 discusses the special case of "data refinement" known from literature. In this case one rule application of ASM corresponds to one rule application of ASM′ and an abstraction function is given, that maps states of ASM′ to states of ASM.

Section 6.2 then considers the general case, in which the correspondence between states is given by an arbitrary relation, that we call a "coupling invariant". The restriction, that *one* rule application of ASM must correspond to *one* of ASM′ is dropped. Instead it is only required that the diagram shown in Fig. 5.1 can be decomposed into smaller diagrams, such that the coupling invariant holds at all partitioning points. The main result of this section is the theorem, that under this condition the commutativity proof of the whole diagram can be split to commutativity proofs for the subdiagrams. It is shown, that it is sufficient to prove one proof obligation for each subdiagram in order to show correctness *and* completeness.

Section 6.3 considers an alternative to the definition of refinement correctness we gave in Sect. 5. The new correctness notion is called "trace correctness", since it does not rely on input/output behavior, but compares traces of the ASMs. Trace correctness is stronger than correctness. For deterministic ASMs correctness and completeness imply trace correctness. We will give an example, that shows, that this is not the case for indeterministic ASMs. Therefore we will, before we consider indeterministic ASMs, define trace correctness formally. Like for the deterministic case we will generalize the approach from literature which uses abstraction functions to the use of arbitrary coupling invariants. We will show, that the proof obligations for correctness and trace correctness differ only marginally.

Section 6.4 treats refinements of indeterministic ASMs. We will show, which modifications are necessary, to apply the modularization theorem for indeterministic ASMs. As the main difference we will have two separate proof obligations for correctness and completeness. Also the complication must be considered, that the size of subdiagrams resulting from the modularization may now depend on indeterministically chosen rules.

Section 6.5 gives optimizations of the theorem, that are possible for an iterated refinement from ASM first to ASM′ and then to ASM″.

Section 6.6 finally discusses some related work. Correctness in the sense, that both ASMs make the same outputs during runs ("behavioral correctness") is is identified as a special case of trace correctness.

6.1 Data Refinement

6.1.1 Definition

The simplest case of a refinement of a sequential ASM is "data refinement" ([Hoa72]). The idea is to transform an "abstract" set of states S in ASM to a more "concrete" state set S' in ASM$'$ (this idea is also the basis of many purely algebraic refinement notions). If a state from S e.g. stores a set of elements, then the state in S' that represents it could store a list of the same elements. In data refinement the connection between states is usually given by an abstraction function

$$\text{abstr} : \text{S}' \to \text{S}$$

that maps concrete states to abstract ones. The function may be partial, since not every concrete state must represent an abstract one (e.g. only duplicate-free lists could be used as representations of sets). The function also does not need to be injective, since several concrete states may represent the same abstract one (in the example [1,2] and [2,1] would represent the same set). The state transition function τ' of ASM$'$ has to be chosen in this kind of refinement, such that it achieves the same effect on concrete states as τ of ASM on abstract ones. This can be formalized as

$$\begin{array}{l} \text{abstr}(\underline{\text{x}}') = \underline{\text{x}} \wedge \neg\ \text{final}(\underline{\text{x}}) \wedge \neg\ \text{final}'(\underline{\text{x}}') \\ \to \langle \text{RULE}(;\underline{\text{x}}) \rangle\ \langle \text{RULE}'(;\underline{\text{x}}') \rangle\ \text{abstr}(\underline{\text{x}}') = \underline{\text{x}} \end{array} \tag{6.1}$$

in DL (where $\underline{x}$ and $\underline{x}'$ are two disjoint vectors of program variables, that result from the translation of dynamic functions from the two ASMs). Informally the equivalence of rule applications can be described as the commutativity of the diagram in Fig. 6.1.

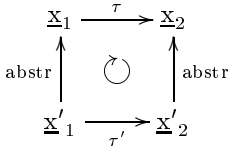


Figure 6.1 : Commuting 1:1 Diagram

Since one rule application of ASM is equivalent to one of ASM$'$, both systems work synchronously. The fact, that (6.1) is the main criterion sufficient for the equivalence of ASM and ASM$'$ is shown by induction on the number of executed steps. Informally commuting diagrams are put together as shown in Fig. 6.2:

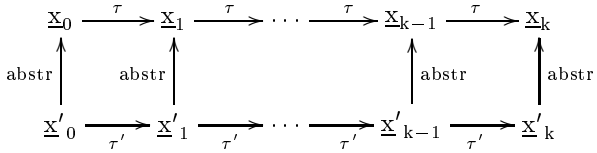


Figure 6.2 : Commuting 1:1 Diagrams

For the induction base it is required, that initial states are connected by the abstraction function:

$$\text{IN}(\underline{\text{x}},\underline{\text{x}}') \to \text{abstr}(\underline{\text{x}}') = \underline{\text{x}} \tag{6.2}$$

Normally this is guaranteed by simply defining $IN(\underline{x}, \underline{x}')$ as $abstr(\underline{x}') = \underline{x}$. Finally it is needed, that two final states deliver the same output

$$\text{abstr}(\underline{\text{x}}') = \underline{\text{x}} \wedge \text{final}'(\underline{\text{x}}') \wedge \text{final}(\underline{\text{x}}) \rightarrow \text{OUT}(\underline{\text{x}},\underline{\text{x}}') \tag{6.3}$$

and that both ASMs reach their final states simultaneously:

$$\text{abstr}(\underline{\text{x}}') = \underline{\text{x}} \rightarrow (\text{final}(\underline{\text{x}}) \leftrightarrow \text{final}'(\underline{\text{x}}')) \tag{6.4}$$

Putting everything together we get the theorem

Theorem 1 *correctness and completeness for data refinement*
The validity of the four proof obligations (6.1), (6.2), (6.3) and (6.4) implies the correctness and completeness of the refinement from ASM to ASM′

$$(6.1) \wedge (6.2) \wedge (6.3) \wedge (6.4) \Rightarrow \text{ASM} \bowtie \text{ASM}'$$

6.2 The Modularization Theorem

6.2.1 Informal Description

In this section we give a generic theorem for the modularization of equivalence proofs for refinements of sequential ASMs. We will first give an informal correctness proof. Then we will sketch its formalization in KIV. Finally we will also show a proof for the first-order formalization of ASMs. This will assure, that the theorem is independent of the formalization of ASMs.

The basic idea of the theorem is shown most easily by looking at the commuting diagram, that describes the equivalence of two ASMs. To modularize the proof, we decompose the diagram into subdiagrams, as it is shown in Fig. 6.3. Edges connecting states represent an (arbitrary!) relation *INV*, that we call the *coupling invariant*. The basic assumption, underlying a modularization of

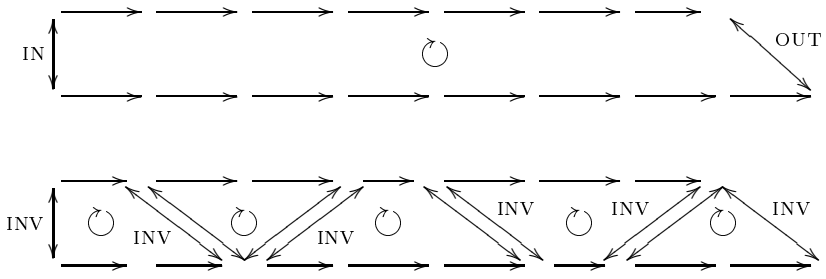


Figure 6.3 : decomposition of the full diagram (above) in subdiagrams (below) using a coupling invariant

this kind is, that the correspondence between two computations of the ASMs can be reduced to the correspondence of suitable "subcomputations" (i.e. finite sequences of rule applications), that both ASMs do in the same order. Corresponding "similar" states are characterized by the coupling invariant. This correspondence automatically decomposes the full diagram into subdiagrams (simply connect corresponding states). For practical cases it is helpful, to also name corresponding sequences of rule applications. This helps to understand how the commuting diagrams look, and

we will of course do this in the Prolog-WAM-case study. But for the formalization it is redundant to give corresponding rule sequences.

Since we allow full freedom in the definition of the coupling invariant, a subcomputation can consist of an arbitrary number of rule applications. The number can even depend on the values of certain program variables. As an important special case some subcomputation of one ASM may be dropped in the other altogether. This case results in triangular diagrams.

The basic assumption, that both ASMs run through corresponding subcomputations, need not always be fulfilled (ASM′ could be the result of an arbitrary program transformation on ASM, e.g. ASM′ could do the computation steps of ASM in reverse order). But for many cases the assumption is valid, especially in compiler verification, where corresponding subcomputations are a natural result of executing corresponding parts of the compiled program.

The idea for the modularization theorem therefore is: Given a decomposition of the full diagram into subdiagrams, then commutation of all subdiagrams implies equivalence of both ASMs.

6.2.2 Definition of the Theorem

To turn the idea into a theorem, we will now

1. formally specify how to decompose diagrams into subdiagrams in DL
2. give proof obligations for the commutativity of subdiagrams
3. formally state and prove the modularization theorem

We assume, that we are given ASM and ASM′ translated to DL as ASM($\underline{x}$) and ASM′($\underline{x}'$) with two disjoint vectors $\underline{x}$ and $\underline{x}'$ of variables. A correspondence between states will then be given as a coupling invariant, i.e. a DL formula $INV(\underline{x}, \underline{x}')$ with free variables in $\underline{x} \cup \underline{x}'$. Defining the edges of subdiagrams to be those pairs of states $(\underline{x}, \underline{x}')$ for which INV holds already gives a suitable decomposition of the diagram into subdiagrams. If there are no triangular diagrams, it is sufficient to show, that for each pair of nonfinal states, a commuting (sub-)diagram as shown in Fig. 6.4 can be attached.

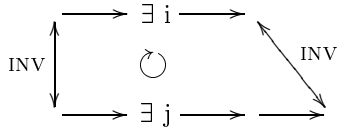


Figure 6.4 : generic commuting diagram

The size of the diagram need not be given explicitly, it is sufficient to show, that there are positive numbers of rule applications for both ASMs, such that INV holds again in the resulting states. Formalized as a DL formula this results in the following proof obligation (the precondition $ndt(\underline{x}, \underline{x}') = mn$ can be ignored, it will be explained in the following):

$$\begin{aligned} &\mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{mn} \\ \rightarrow\ &\exists\ \mathrm{i} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}(\underline{\mathrm{x}})\ \textbf{then}\ \mathrm{RULE}(;\underline{\mathrm{x}})\ \textbf{times}\ \mathrm{i}\rangle \\ &\quad \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}'(\underline{\mathrm{x}}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{\mathrm{x}}')\ \textbf{times}\ \mathrm{j}\rangle \\ &\quad\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \end{aligned} \tag{6.5}$$

An additional problem occurs when triangular diagrams are present. Then it must be prohibited that the whole diagram consists solely of triangular ones as shown in Fig. 6.5 and 6.6. In the first case ASM′ could have an infinite run, while ASM would not make a single step, which would violate completeness. Similarly, the second case would violate correctness.

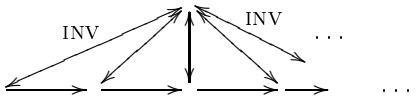


Figure 6.5 : infinite sequence of 0:n diagrams

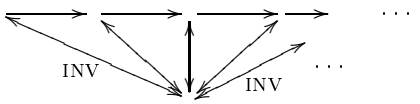


Figure 6.6 : infinite sequence of m:0 diagrams

Since triangular diagrams often occur in applications as results of optimizations, we must restrict the number of possible successive triangular diagrams. To do this, we first have to decide for every pair of states $(\underline{x}, \underline{x}')$, for which *INV* holds, which type of diagram follows:

- An m:n diagram, where both ASM and ASM′ make a positive number of steps,
- An m:0 diagram, where only ASM makes a positive number of steps, or
- a 0:n diagram, where only ASM′ makes a positive number of steps

For this purpose we introduce a function *ndt* ("'next diagram type"'), which returns for every pair of states $(\underline{x}, \underline{x}')$, for which *INV* holds, an element from {*mn,m0,0n*}. To implement the restriction on the number of successive m:0 diagrams we use a function *execm0*. For $(\underline{x}, \underline{x}')$ with $INV(\underline{x}, \underline{x}')$ and $ndt(\underline{x}, \underline{x}') = m0$ the result of $ndt(\underline{x}, \underline{x}')$ should be a natural number that bounds the number of successive m:0 diagrams.

Proof obligation (6.5) now considers the case of m:n diagrams and therefore gets the additional precondition $ndt(\underline{x}, \underline{x}') = mn$. For m:0 diagrams we have the following proof obligation:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \wedge \mathrm{execm0}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{k} \\ \rightarrow \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}(\underline{\mathrm{x}}) \textbf{ then } \mathrm{RULE}(;\underline{\mathrm{x}}) \textbf{ times } \mathrm{i} \rangle \\ \qquad (\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \\ \qquad\ \wedge\ (\neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \rightarrow \mathrm{execm0}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') < \mathrm{k})) \end{array} \tag{6.6}$$

It says, that a m:0 diagram must preserve the coupling invariant, and if another m:0 diagram follows, then the value of of *execm0* must have decreased (if $execm0(\underline{x}, \underline{x}') = k$, then at most $k+1$ successive m:0 diagrams are possible). For 0:n diagrams we get the following dual proof obligation:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{0n} \wedge \mathrm{exec0n}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{k} \\ \rightarrow \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \textbf{ then } \mathrm{RULE}'(;\underline{\mathrm{x}}') \textbf{ times } \mathrm{j} \rangle \\ \qquad (\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \\ \qquad\ \wedge\ (\neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{0n} \rightarrow \mathrm{exec0n}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') < \mathrm{k})) \end{array} \tag{6.7}$$

Note that the proof obligation for m:0 diagrams does not assume, that ASM′ is not in a final state. It is possible (and indeed does occur in the Prolog-WAM case study, see Sect. 13.2) that ASM′ has already terminated, while ASM is still doing "superfluous" steps (such a situation is not possible in data refinement). But in this case it must be required that *only* m:0 diagrams are possible:

$$\mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{final}'(\underline{\mathrm{x}}') \rightarrow \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \tag{6.8}$$

Dually it has to be shown for n:0 diagrams that

$$\mathrm{INV}(\underline{x}, \underline{x}') \wedge \mathrm{final}(\underline{x}) \wedge \neg\, \mathrm{final}'(\underline{x}') \rightarrow \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \tag{6.9}$$

To connect the coupling invariant to the input/output relation of the refinement, we finally have to require

$$\mathrm{IN}(\underline{x}, \underline{x}') \rightarrow \mathrm{INV}(\underline{x}, \underline{x}') \tag{6.10}$$

and

$$\mathrm{INV}(\underline{x}, \underline{x}') \wedge \mathrm{final}(\underline{x}) \wedge \mathrm{final}'(\underline{x}') \rightarrow \mathrm{OUT}(\underline{x}, \underline{x}') \tag{6.11}$$

similar to proof obligations (6.2) and (6.3) for data refinement. With these proof obligations we can now state the modularization theorem.

Theorem 2 *Modularization Theorem for Sequential ASMs.*
Given a refinement from ASM to ASM′ of deterministic ASMs, a predicate *INV* and functions *ndt*, *exec0n*, *execm0*, such that the proof obligations (6.5), (6.6), (6.7), (6.8), (6.9), (6.10) and (6.11) hold, then the refinement is correct and complete:

$$\begin{aligned} &\text{ASM deterministic} \wedge \text{ASM}' \text{ deterministic} \\ &\wedge\ (6.5) \wedge (6.6) \wedge (6.7) \wedge (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\ &\Rightarrow \text{ASM} \bowtie \text{ASM}' \end{aligned}$$

Before we discuss the proof of the theorem, here are some remarks on how it is applied:

- The theorem does not require to verify separate proof obligations for correctness and completeness.
- The main difficulty in applying the theorem is to find a suitable coupling invariant. The type of the following subdiagram usually follows simply from which case of the rules of ASM and ASM′ is executed in the pair (x, x') of states. *execm0* (and similarly *exec0n*) usually is constantly 0, i.e. an m:0 diagram is never followed by another. Otherwise the result of *execm0* often is the size of a datastructure (e.g. a stack) from the state of ASM, that is currently reduced (e.g. to the empty stack).
- Data refinement is the simple special case, in which $INV(\underline{x}, \underline{x}') \equiv abstr(\underline{x}') = \underline{x}$ and *ndt* is constantly *mn* (no triangular diagrams). The proof obligation (6.1) from data refinement is then the case of (6.5), where both i and j are instantiated by 1. (6.4) follows trivially from (6.8) and (6.9).
- The subdiagrams resulting from the decomposition have the same form as the original diagram. It is therefore possible to apply the modularization theorem recursively on the subdiagrams. This was done in the Prolog-WAM case study for the refinement 5/6 considered in Sect. 15.2.

6.2.3 The Proof of the Theorem

The proof of the modularization theorem consists of two parts. In the first part it is shown, that the proof obligations imply correctness, in the second that they imply completeness. Both proofs are dual (only the roles of ASM and ASM′ are exchanged) therefore we only consider the proof for correctness.

The proof is done by reducing the correctness assertion to a property, that can be shown by induction over the number of applied rules in ASM′. To state this properties easily, we denote by $\underline{x}_i$ the state of ASM that results from i rule applications, when starting in $\underline{x}$. In DL, $\underline{x}_i$ can be formally defined as

$$\underline{y} = \underline{x} \rightarrow \langle \textbf{loop if} \neg \text{ final}(\underline{y}) \textbf{ then } \text{RULE}(;\underline{y}) \textbf{ times } i \rangle \underline{y} = \underline{x}_i$$

(note, that for a final state $\underline{x}$, we have $\underline{x}_i = \underline{x}$). Now we define a property *PROP* by

$$\text{PROP}(\underline{x}, \underline{x}') \leftrightarrow \exists \text{ i, j. } \text{INV}(\underline{x}_i, \underline{x}'_j)$$

Informally, *PROP* says that $(\underline{x}, \underline{x}')$ is a pair of states, such that there is a number i of rule applications of ASM and a number j of rule applications of ASM′, such that for the states $\underline{x}_i$ and $\underline{x}_j$ reached then the coupling invariant holds. For this property the following lemma holds:

Lemma 1 *PROP* is an invariant of ASM′: If $\underline{x}$, $\underline{x}'$ are two states of ASM and ASM′ with $INV(\underline{x}, \underline{x}')$, then $PROP(\underline{x}, \underline{x}'_k)$ will hold for all states $\underline{x}'_k$, that are reached during the run of ASM′ (starting from $\underline{x}'$).

Proof of Lemma 1 The proof is by induction over the number k of applied rules. The base case ($k = 0$) is trivial. In the induction step we can assume two states $\underline{x}$, $\underline{x}'$ with $INV(\underline{x}, \underline{x}')$ and two values i and j such that $INV(\underline{x}_i, \underline{x}'_{k+j})$, and we have to find i' and j', so that $INV(\underline{x}_{i'}, \underline{x}'_{(k+1)+j'})$ holds. The case $j \neq 0$ is simple with $i' := i$, $j' := j - 1$ as well as the case where $\underline{x}'_k$ is already a final state. Otherwise we need Lemma 2 described below, to deduce from $INV(\underline{x}_i, \underline{x}'_k)$ that we can construct an $i'' \geq 0$ with $INV(\underline{x}_{i+i''}, \underline{x}'_k)$ and either $ndt(\underline{x}_{i+i''}, \underline{x}'_k) \neq m0$ or $final(\underline{x}_{i+i''}, \underline{x}'_k)$. In the first case assumptions (6.5) and (6.7) guarantee the existence of $i''' \geq 0$ and $j''' > 0$ such that $INV(\underline{x}_{i+i''+i'''}, \underline{x}'_{k+j'''})$ holds. Therefore we can choose $i' := i'' + i'''$, $j' := (j''' - 1)$. In the other case because of (6.9) a 0:n diagram follows and the proof follows with (6.7) as above. □

The proof uses the following lemma, that says, that two corresponding states can be followed by only finitely many m:0 diagrams. The state thereby reached by ASM is $\underline{x}_i$.

Lemma 2 For every two states $\underline{x}$, $\underline{x}'$ with $INV(\underline{x}, \underline{x}')$ there is an $i \geq 0$, such that $INV(\underline{x}_i, \underline{x}')$ and either $ndt(\underline{x}_i, \underline{x}') \neq m0$ or $final(\underline{x}_i)$ hold.

Proof of Lemma 2 In the case, that $ndt(\underline{x}, \underline{x}')$ is equal to $m0$ and we do not already have $final(\underline{x})$ (otherwise the theorem holds with $i := 0$), the proof is by (noetherian) induction on the size of $execm0(\underline{x}, \underline{x}')$. (6.6) implies that there is an $i' > 0$, such that $INV(\underline{x}_{i'}, \underline{x}')$ and either $execm0(\underline{x}_{i'}, \underline{x}')$ has become smaller or $ndt(\underline{x}_{i'}, \underline{x}') \neq m0$. In the first case the statement follows from the induction hypothesis, in the second $i := i'$ is sufficient. □

Proof of Theorem 2 Using lemmas 1 and 2 the proof of the correctness of the refinement is as follows: Let $(\underline{x}', \underline{x}'_1, \ldots \underline{x}'_k)$ be an arbitrary terminating run of ASM′ (so we have $final'(\underline{x}'_k)$) and $\underline{x}$ a state with $IN(\underline{x}, \underline{x}')$. Then (6.10) implies $INV(\underline{x}, \underline{x}')$. Now Lemma 1 implies, that there are i, j, such that $INV(\underline{x}_i, \underline{x}'_{k+j})$ holds. Because of definition we have $\underline{x}'_{k+j} = \underline{x}'_k$, therefore $INV(\underline{x}_i, \underline{x}'_k)$ holds. With Lemma 2 we get an i', such that $INV(\underline{x}_{i+i'}, \underline{x}'_k)$ and either $ndt(\underline{x}_{i+i'}, \underline{x}'_k) \neq m0$ or $final(\underline{x}_{i+i'})$. The first case is impossible because of (6.8), therefore $\underline{x}_{i+i'}$ is a final state too, and

we have a terminating run of ASM. (6.11) finally implies that at the end we have $OUT(\underline{x}_{i+i'}, \underline{x}'_k)$. □

From the proof of the modularization theorem we immediately get

Corollary 1 *If it is possible, to verify a refinement by decomposing it into m:n diagrams, then there is also a possibility to verify it using 1:1, 0:1 and 1:0 diagrams.*

As the new coupling invariant simply choose *PROP*. Of course to really choose the stronger decomposition into smaller diagrams is not a good idea for practical applications, since then part of the generic proof has to be done when verifying the proof obligations. Proofs will get even bigger, when one tries to avoid rule applications (or equivalently DL programs) in *PROP*. This is possible when all diagrams have a fixed size (that is independent of the size of data structures in the ASMs). Then a function *nextij* can be defined that computes for two states the numbers i and j of rule applications, that are necessary to reach two states again, for which *INV* holds. Instead of using quantification over all possible i and j, we can then formulate *PROP* as a conjunction over the formulas

$$\text{nextij}(\underline{\text{x}},\underline{\text{x}}') = (\text{i},\text{j}) \rightarrow \text{INV}(\underline{\text{x}}_\text{i}, \underline{\text{x}}'_\text{j})$$

where (i, j) runs through all concrete values, that are less than the maximal diagram size. Finally, the rule applications of ASM must be removed from the formulas $\underline{x}_i$ (and similarly for the $\underline{x}'_j$) by symbolic execution (this is possible, since i is now a concrete number in each case). The result is a coupling invariant which is sufficient to show refinement correctness. But since *INV* is the conjunct for $(i, j) = (0, 0)$, the computed new coupling invariant is unnecessary complicated, unless the original decomposition used no other than 1:1, 0:1 and 1:0 diagrams. In general it is therefore a good idea in practical applications to make diagrams *as large as possible*, to have a small coupling invariant. Two cases in the Prolog-WAM case study that exemplify this fact are the refinements 2/3 and 3/4 (see the remarks at the end of Sect. 13.2, and the comparison of effort for the two refinements in KIV vs. in Isabelle in Sect. 20).

6.2.4 Formalization of the proof in DL

It is possible to formalize the proof of the modularization theorem given above in DL. Property *PROP* is then defined as

$$\begin{aligned}&\text{PROP}(\underline{\text{x}}, \underline{\text{x}}') \equiv\\ &\exists\ \text{i, j.}\ \langle\textbf{loop if}\ \neg\ \text{final}(\underline{\text{x}})\ \textbf{then}\ \text{RULE}(;\underline{\text{x}})\ \textbf{times}\ \text{i}\rangle\\ &\qquad\langle\textbf{loop if}\ \neg\ \text{final}'(\underline{\text{x}}')\ \textbf{then}\ \text{RULE}'(;\underline{\text{x}}')\ \textbf{times}\ \text{j}\rangle\ \text{INV}(\underline{\text{x}}, \underline{\text{x}}')\end{aligned} \tag{6.12}$$

The formal proof of the modularization theorem required 452 proof steps and 64 interactions in KIV. Half of these were necessary to show correctness, the other half to show completeness of the refinement. The numbers include proofs of elementary facts such as $(\underline{x}_i)_{i'} = \underline{x}_{i+i'}$. By instantiation (actualization) the modularization theorem can be applied on every concrete ASM refinement. The full formal specification and the proved theorems and lemmas are given in appendix C.2. Theorems *corr-step* and *finite-0n* from the appendix correspond to the Lemma 1 and to the case $ndt(\underline{x}, \underline{x}') = 0n$ of Lemma 2.

6.2.5 Formalization of the Proof in First-Order Logic

The proof of the modularization theorem can also be formalized in first-order logic. This first requires to formalize state transition relations as a datatype (in higher-order logic this step can be dropped). The simplest formalization uses the datatype of dynamic functions from Sect. 4. A relation is a dynamic function r that assigns a boolean result to a pair $st_1 \times st_2$ of states. $r[st_1$

$\times$ st_2] holds if and only st_2 is a possible successor state of st_1. The state transition relation ρ of an ASM then is a constant of this datatype. Since we consider sequential ASMs, ρ will fulfill the functionality axiom

$$\rho[\mathrm{st}_1 \times \mathrm{st}_2] \wedge \rho[\mathrm{st}_1 \times \mathrm{st}_3] \rightarrow \mathrm{st}_2 = \mathrm{st}_3$$

The predicate *final*, that characterizes final states, is defined as

$$\mathrm{final}(\mathrm{st}) \equiv \neg\ \exists\ \mathrm{st}_0.\ \rho[\mathrm{st} \times \mathrm{st}_0]$$

To formalize the proof in first-order logic we must then formalize the semantics of the ASMs. To define i-fold rule application, a relation ρ^{i} is defined by

$$\rho^0[\mathrm{st}_1 \times \mathrm{st}_2] \leftrightarrow \mathrm{st}_1 = \mathrm{st}_2$$
$$\rho^{\mathrm{i}+1}[\mathrm{st}_1 \times \mathrm{st}_2] \leftrightarrow \exists\ \mathrm{st}_0.\ \rho^{\mathrm{i}}[\mathrm{st}_1 \times \mathrm{st}_0] \wedge \rho[\mathrm{st}_0 \times \mathrm{st}_2]$$

The relation ρ^i corresponds to the semantics of

loop if $\neg$ final(st) **then** RULE(st) **else abort times** i

in DL. Finally we can define the input/output relation ρ^* of the ASM as

$$\rho^*[\mathrm{st}_1 \times \mathrm{st}_2] \leftrightarrow \exists\ \mathrm{i}.\ \rho^{\mathrm{i}}[\mathrm{st}_1 \times \mathrm{st}_2] \wedge \mathrm{final}(\mathrm{st}_2)$$

Again this corresponds to the semantics of the while loop in DL. The proof obligations and the first-order proofs then can be got from the DL version by simply replacing

$$\exists\ \mathrm{i}.\ \langle \textbf{loop if } \neg\ \mathrm{final(st)}\ \textbf{then}\ \mathrm{RULE(st)}\ \textbf{times}\ \mathrm{i}\rangle\ \varphi(\mathrm{st})$$

with

$$\exists\ \mathrm{i},\mathrm{st}_0\ .\ \rho^{\mathrm{i}}(\mathrm{st},\mathrm{st}_0) \wedge \varphi(\mathrm{st}_0)$$

(using a new variable st_0). The effort for doing the proofs in first-order logic in KIV was with 98 interactions somewhat higher than in DL. The main reason for this is, that DL automates the computation of the necessary iterations of a while loop with heuristics, while in the first-order variant this number has to be given interactively by quantifier instantiation. The number of proof steps for the first-order variant is 247, which is somewhat less than in DL, since applications of tactics for DL programs are now replaced by applications of rewrite rules, and one application of the simplification tactic will often apply several rewrite rules in one step.

6.3 Trace Correctness

The definition of refinement correctness given in Chap. 5 was based on a comparison of the input/output behavior of the two ASMs. An alternative is to compare the traces of the ASMs. In the simplest case there is an abstraction function *abstr* (like in data refinement, see Sect. 6.1), such that for every run $(\underline{x}'_0, \underline{x}'_1, \ldots)$ of ASM′ $(abstr(\underline{x}'_0), abstr(\underline{x}'_1), \ldots)$ is a run of ASM. The main differences to our definition: Already the definition of refinement correctness mentions an abstraction function, and not only finite but also infinite runs are considered. In a correct refinement it

is no longer allowed to implement a terminating run by a nonterminating one. For deterministic ASMs this restriction is not very important, since in a *complete* refinement the implementation of a terminating run by a nonterminating one is impossible. But for indeterministic (e.g. distributed) ASMs which will be considered in the next section there is a major difference. The difference can be exemplified by looking at the refinement of the deterministic ASM defined by the rule

RULE(**var** init,b) ≡
    if init **then** b := false, init := false

to the indeterministic ASM′ defined by the rule (the DL statement $b := ?$ "guesses" a boolean value. It is equivalent to the **choose** statement of ASMs as defined in Sect. 4.2 in [Gur95])

RULE′(**var** init,b) ≡
    if init **then** b := ?, init := false **else if** b **then** b := b

For an initial state with $b = init = true$ ASM has exactly one trace, that applies $RULE$ once, setting b and $init$ to $false$, and terminates (since $RULE$ is no longer applicable). The same run is possible in ASM′ too, if the first rule application chooses $b = false$. But ASM′ has an additional nonterminating run, when the choice $b = true$ is taken. In this run $RULE'$ is applied infinitely often without changing the state ($b = true$ and $init = false$) any more.

The refinement is correct and complete in the sense of our definition (when both the IN and OUT relation are chosen to be identity), since for every *finite* run of one of the ASMs there is a suitable finite run of the other. But the refinement is not trace-correct, since for the infinite run of ASM′ there is no corresponding run in ASM.

Whether the refinement is viewed as correct in an intuitive sense depends on whether the whole run or only the result of an ASM can be observed. If only results are relevant, then the refinement *is* correct, since ASM′ does not deliver any other results than ASM. But if both ASMs are viewed as reactive systems, and an observer can view and compare at least some of the intermediate states, then the refinement should not be considered to be correct.

Therefore we define at this point the notion of "trace correctness", such that it is general enough to be usable for indeterministic ASMs. Instead of using abstraction functions, we again use the more general notion of "corresponding states" defined by a coupling invariant. We require, that for a trace-correct refinement, that for every run of ASM′ there exists a corresponding run of ASM and intermediate ("observable") pairs of states, for which the coupling invariant holds. For a finite run, we require the run of ASM and the number of corresponding states to be finite. Also the last pair of states should then be the two final states. For an infinite run, we require an infinite run of ASM and an infinite number of corresponding states. Formally this gives

Definition 2 A refinement of ASM to ASM′ is *trace-correct*, in short ASM ▸ ASM′, if there is a coupling invariant $INV(\underline{x}, \underline{x}')$, such that

- for every finite run $(\underline{x}'_0, \underline{x}'_1, \ldots, \underline{x}'_m)$ of ASM′ (with $\underline{x}'_m \in F'$) and for every $\underline{x}_0$ with $IN(\underline{x}_0, \underline{x}'_0)$ there is a finite run $(\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_n)$ of ASM (with $\underline{x}_n \in F$) and two strictly monotonic sequences of natural numbers $(i_0, i_1, \ldots, i_p)$ and $(j_0, j_1, \ldots, j_p)$ of the same length, such that $i_p = m$, $j_p = n$ and for all $k \leq p$ $INV(\underline{x}_{i_k}, \underline{x}'_{j_k})$ holds.
- for every infinite run $(\underline{x}'_0, \underline{x}'_1, \ldots)$ of ASM′ and every state x_0 such that $IN(\underline{x}_0, \underline{x}'_0)$ there is an infinite run $(\underline{x}_0, \underline{x}_1, \ldots)$ of ASM and two infinite, strictly monotonic sequences of natural numbers $(i_0, i_1, \ldots)$ and $(j_0, j_1, \ldots)$, such that for all n $INV(\underline{x}_{i_n}, \underline{x}'_{j_n})$ holds.
- (6.11) holds, i.e. for every pair of final states the coupling invariant implies OUT.

The pairs of states comparable with the coupling invariant are $(\underline{x}_{i_0}, \underline{x}_{i_1}, \ldots)$ and $(\underline{x}_{j_0}, \underline{x}_{j_1}, \ldots)$. The definition immediately implies

Theorem 3 *Relations between Correctness and Trace Correctness.*
For every two abstract state machines ASM and ASM′:

stream =
enrich Dynfun[nat,state] **with**
functions cons : state × stream → stream;
cdr : stream → stream;
variables st : state; s : stream;
axioms cons(st,s)[0] = st,
cons(st,s)[m +1] = s[m],
cdr(s)[m] = s[m +1]
end enrich

Figure 6.7 : Specification of Streams

- ASM ▸ ASM′ ⇒ ASM ▷ ASM′.
- ASM′ deterministic ∧ ASM ⋈ ASM′ ⇒ ASM ▸ ASM′

To formalize the definition of trace correctness in DL, we first need a formal definition of the traces of an ASM. For this purpose we use the enrichment of dynamic functions given in Fig. 6.7.

For an ASM rule *RULE* with state argument *st* a stream *s* is a trace of the ASM (with initial state *s*[0]), if the predicate *Trace*(*s*) defined by

Trace(s) ≡
∀ m, st. st = s[m] → ⟨**if** ¬ final(st) **then** RULE(;st)⟩ st = s[m +1]

holds. The definition depends on the chosen ASM rule *RULE* and is such that a finite trace (st_0, st_1, ..., st_m) corresponds to a stream *s* with $s[k] = st_k$ for $k \leq m$ and $s[k] = st_m$ for $k > m$ (because of the test for ¬ *final*(*st*)). With this definition, the requirement of trace correctness relative to some *INV* can then be formalized as

$$\begin{array}{l} \forall\ \mathrm{s}'.\quad \mathrm{Trace}'(\mathrm{s}') \\ \quad \rightarrow \exists\ \mathrm{s}.\quad \mathrm{Trace}(\mathrm{s}) \\ \qquad \wedge\ \forall\ \mathrm{m},\ \mathrm{k}.\ \exists\ \mathrm{i},\ \mathrm{j}.\quad \mathrm{i} \geq \mathrm{m} \wedge \mathrm{j} \geq \mathrm{k} \wedge \mathrm{INV}(\mathrm{s[i]},\mathrm{s}'[\mathrm{j}]) \\ \qquad\qquad\qquad\qquad \wedge\ (\mathrm{final}(\mathrm{s[i]}) \leftrightarrow \mathrm{final}'(\mathrm{s}'[\mathrm{j}])) \end{array} \tag{6.13}$$

In the formula *Trace′* is the predicate for *RULE′* of ASM′ and *Trace* is the predicate for*RULE* of ASM. Note that "*INV* holds infinitely often" is formalized as "for every two positions m, k in both traces, there are two larger ones, for which *INV* holds" as it is usual in temporal logic ("infinitely often φ" $\equiv \Box\Diamond\varphi$). The case distinction over finite and infinite runs is unnecessary because of our formalization of traces (that extends finite to infinite runs that repeat the final state). The requirement $final(s[i]) \leftrightarrow final'(s'[j])$ is for the special case of finite runs.

We will now show, that the difference between correctness and trace correctness is minimal, since the proof obligations for correctness already imply trace correctness for the coupling invariant. Informally the reason for this is simply, that our decomposition of the whole commuting diagram in commuting subdiagrams does not require finiteness of the traces. Also the decomposition does neither allow n:∞ diagrams nor infinitely many successive 0:n diagrams. If we analyze the proof for the modularization theorem, we find that the condition, that we must have only finitely many successive 0:n diagrams (i.e. that the value of *exec0n* in proof obligation (6.7) decreases) is not necessary for correctness, but for completeness as well as for trace correctness. Formally we have the following theorem:

Theorem 4 *Trace Correctness for sequential ASMs*
If all proof obligations of theorem 2 hold, then the refinement of ASM to ASM′ is also trace-correct

for the coupling invariant INV:

$$(6.5) \wedge (6.6) \wedge (6.7) \wedge (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11)$$
$$\Rightarrow \text{ASM} \blacktriangleright \text{ASM}'$$

To prove the theorem we define

$$\text{INV}'(\text{st},\text{st}') \equiv \text{INV}(\text{st},\text{st}') \wedge (\text{final}(\text{st}) \leftrightarrow \text{final}(\text{st}'))$$

and show as a first lemma, that for every pair of states with INV two more can be reached in the further run of the ASMs with INV':

Lemma 3 If the proof obligations of Theorem 2 hold, and if for a state st of ASM and a trace s' of ASM' $INV(st, s'[0])$ holds, then there are a trace of ASM starting with $s[0] = st$ and numbers $i, j \geq 0$, such that $INV'(s[i], s'[j])$ holds.

Proof of Lemma 3 For the proof 4 cases have to be considered. The two cases in which st and $s'[0]$ are either both final states or both nonfinal states are trivial with $i = j := 0$. If st is a final state, but not $s'[0]$, then according to Lemma 2 there is an i, such that $INV(s[i], s'[0])$ holds, and we have either $final(s[i])$ or $ndt(s[i], s'[0]) \neq \text{m0}$. Since the second case is impossible because of proof obligation (6.8), the proof is completed with $j := 0$ in the first case. Finally we have the fourth case in which $s'[0]$ is a final state, but not st. This case follows similarly with the dual lemma of Lemma 2. □

Using the lemma we are now able to prove, that whenever we have two states with INV', we can add a diagram with a positive number of steps for both ASMs, such that INV' holds again at the end.

Lemma 4 If the proof obligations for Theorem 2 hold, and if for a state st of ASM and a trace s' of ASM' $INV'(st,s'[0])$ holds, then there are a trace s of ASM with $s[0] = st$ and numbers $i, j > 0$, such that again $INV'(s[i], s'[j])$ holds.

Proof of Lemma 4 If both $final(st)$ and $final'(s'[0])$ hold, then we have $s'[1] = s'[0]$ and $s[1] = s[0] = st$ for an arbitrary trace s starting with st. Therefore $i = j := 1$ will be sufficient to prove the goal. Otherwise both states are nonfinal, and we have 3 cases:

- $ndt(st,s'[0]) = mn$. Then (according to proof obligation (6.5)) after $i > 0$ steps of ASM and $j > 0$ steps of ASM' two states are reached such that $INV(s[i], s'[j])$ holds, and the goal follows with Lemma 3 above.

- $ndt(st, s'[0]) = m0$. Lemma 2 give $i > 0$, such that $INV(s[i], s'[0])$ and $ndt(s[i], s'[0]) \neq m0$ hold. If now $ndt(s[i], s'[0]) = mn$, the goal follows as in the previous case. Otherwise $ndt(s[i], s'[0]) = 0n$, and the next 0:n diagram (according to proof obligation (6.7)) gives $j > 0$, such that $INV(s[i], s'[j])$ holds. Again the goal is now implied by Lemma 3.

- $ndt(st, s'[0]) = 0n$. This case is dual to the previous one.

□

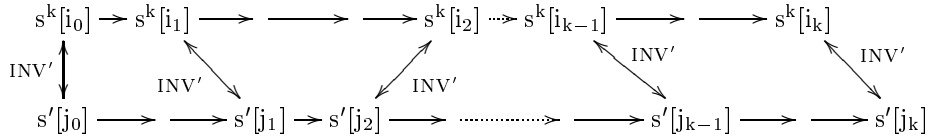


Figure 6.8 : commuting diagrams in the proof of trace correctness

Proof of Theorem 4 The proof is done by inductively adding m:n diagrams with $m, n > 0$, that keep INV' valid, using Lemma 4 in the induction step. Formally we construct in the k^{th} step a trace s^k and two strict monotone sequences $(i_0, \ldots i_k)$ and $(i_0, \ldots i_k)$ such that for all $p \leq k$

$$\text{INV}'(\text{s}^{\text{k}}[\text{i}_{\text{p}}],\ \text{s}'[\text{j}_{\text{p}}])$$

holds. The trace s^k contains k commuting diagrams as shown in diagram 6.8.

The induction base follows from Lemma 3, since in two initial states of the ASMs the coupling invariant holds. The induction step follows from Lemma 4 using the axiom of choice of higher-order logic

$$(\forall\ \text{x}.\ \exists\ \text{y}.\ \text{p(x,y)}) \rightarrow \exists\ \text{f}.\ \forall\ \text{x}.\ \text{p(x,f(x))}$$

The axiom is used, to turn the *possibility* of adding a commuting diagram (in Appendix C.3 formalized as the predicate p) into a *function*, which constructs the next tracess^{k+1}, and the next numbers i_{k+1} and j_{k+1} from the previous ones. Finally we define the trace s that is needed in the theorem by $s[k] := s^k[k]$. s agrees with every s^k until position $i_k (\geq k)$. Choosing positions i and j in the theorem to be $i_{max(m,n)}$ and $j_{max(m,n)}$ is sufficient to prove it, since both are greater or equal to m and n. □

The inductive construction of tuples (consisting of s^k, i_k and j_k) makes the formal proof of trace correctness in KIV somewhat more elaborate than the proof of correctness. Altogether the proof for the most general case (indeterministic ASMs with diagrams of indeterministic size, which we will consider in the next section) required 412 proof steps and 138 interactions (not including Lemma 2, on which the proof was based). A full listing of the theorems and lemmas proved can be found in appendix C.3.

For the special case, in which all diagrams are 1:n or 0:n (i.e. the case, in which proof obligations (6.5) and (6.6) are both provable with $i := 1$) all states of ASM are observable (i.e. all states of ASM are connected with $INV$to some state of ASM′). We can then define a corollary for this case in which the sequence $(i_0, i_1, \ldots)$ is specialized to be (0,1, ...). A similar corollary is possible for the dual case of m:1 and m:0 diagrams. Since data refinement (1:1 diagrams) is in the intersection of both special cases, the corollaries imply that for data refinement $INV(\underline{x}_n, \underline{x'}_n)$ holds for every n.

6.4 Extensions for Indeterministic ASMs

In this section we will consider arbitrary indeterministic ASMs instead of sequential ones. Distributed ASMs, that were described in Sect. 4.4 are an important example for indeterminism. Also the extension with a *CHOOSE* construct (as described in [Gur95], Sect. 4.1) that corresponds to the random assignment in DL results in ASMs that are indeterministic. In the next subsection, we will describe how the modularization theorem of the previous section can be adapted to indeterministic ASMs. The second subsection then gives an example of *diagrams of indeterministic size*. The adaptions discussed to handle this case are more complex than the ones discussed in the

first subsection, since there it is assumed that the size of a diagram can be computed from the knowledge about the initial states alone.

6.4.1 Adaption of the Modularization Theorem to Indeterministic ASMs

A first look at the basic ideas underlying the modularization theorem gives the impression, that decomposing diagrams into smaller diagrams should be possible for indeterministic ASMs in the same way as for deterministic ones.

But if one analyses the proof of the previous section, it becomes clear that the determinism of ASM was essential to express the commutativity of a subdiagram as *one* proof obligation.

This can be shown by looking at proof obligation (6.5) for m:n diagrams: for an indeterministic ASM the requirement only says that for two states $\underline{x}$ and $\underline{x}'$ with *INV* there exist numbers i, j, such that for *one possible successor state* $\underline{x}_i$ and ${\underline{x}'}_j$ of each ASM *INV* holds again. But for correctness, we must find *for every possible* successor state ${\underline{x}'}_j$ a suitable state $\underline{x}_i$ with *INV*. For indeterministic ASMs proof obligation (6.5) must therefore be generalized to

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x},\ \underline{x}') = \mathrm{mn} \\ \rightarrow \exists\ \mathrm{j} > 0.\ [\textbf{loop if} \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}] \\ \qquad \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}\rangle \\ \qquad\qquad \mathrm{INV}(\underline{x},\ \underline{x}') \end{array} \tag{6.14}$$

The right hand side of the implication now states that there is a j, such that for *every* terminating possibility to apply j rules of ASM$'$ an i exists, such that after i (suitable!) rule applications of ASM the invariant holds again. That this *is* the suitable generalization, follows from the fact, that ASMs have no nonterminating rules. Therefore all possibilities to apply j rules terminate (statements of the form "all runs of a program terminate" require an extension of DL, see the discussion in [Gol82], p. 101).

The proof of completeness now requires dually the following proof obligation for m:n diagrams:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\ \underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x},\ \underline{x}') = \mathrm{mn} \\ \rightarrow \exists\ \mathrm{i} > 0.\ [\textbf{loop if} \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}] \\ \qquad \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}\rangle \\ \qquad\qquad \mathrm{INV}(\underline{x},\ \underline{x}') \end{array} \tag{6.15}$$

For the case in which the next i rules applicable in ASM state $\underline{x}$ as well as the next j rules applicable in ASM$'$ state $\underline{x}'$ are deterministic, the three conditions (6.5), (6.14) and (6.15) are all equivalent. If deterministic rules are refined by other deterministic rules, then we have to prove only one obligation (6.5).

The generalization for m:n diagrams can analogously be done for m:0 and 0:n diagrams. But instead of two proof obligations we only get one. For completeness we have to require

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x},\ \underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \mathrm{ndt}(\underline{x},\ \underline{x}') = \mathrm{m0} \\ \wedge\ \mathrm{execm0}(\underline{x},\ \underline{x}') = \mathrm{k} \\ \rightarrow \exists\ \mathrm{i} > 0.\ [\textbf{loop if} \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}] \\ \quad (\quad \mathrm{INV}(\underline{x},\ \underline{x}') \\ \quad\ \wedge\ (\neg\ \mathrm{final}(\underline{x}) \wedge \mathrm{ndt}(\underline{x},\ \underline{x}') = \mathrm{m0} \rightarrow \mathrm{execm0}(\underline{x},\ \underline{x}') < \mathrm{k})) \end{array} \tag{6.16}$$

for m:0 diagrams. For correctness the weaker condition (6.6) is still sufficient. Similarly the correctness proof requires for 0:n diagrams

$$
\begin{aligned}
&\quad \mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\, \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \\
&\wedge\ \mathrm{exec0n}(\underline{x}, \underline{x}') = \mathrm{k} \\
&\rightarrow \exists\ \mathrm{j} > 0.\ [\textbf{loop if} \neg\, \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
&\qquad (\quad \mathrm{INV}(\underline{x}, \underline{x}') \\
&\qquad\ \wedge\ (\neg\, \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \rightarrow \mathrm{exec0n}(\underline{x}, \underline{x}') < \mathrm{k}))
\end{aligned}
\tag{6.17}
$$

which implies the weaker condition (6.7), which is necessary for completeness. With the new proof obligations we can now prove the modularization theorem for indeterministic ASMs:

Theorem 5 *Modularization Theorem for Indeterministic ASMs*
Given a refinement of an indeterministic ASM to ASM′, a predicate *INV* and functions *ndt*, *exec0n*, *execm0*, such that proof obligations (6.14), (6.15), (6.16), (6.17), (6.8), (6.9), (6.10), (6.11) all hold, then the refinement is correct and complete.

$$
\begin{aligned}
&\quad (6.14) \wedge (6.15) \wedge (6.16) \wedge (6.17) \\
&\wedge\ (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\
&\Rightarrow \mathrm{ASM} \bowtie \mathrm{ASM}'
\end{aligned}
$$

For correctness and trace-correctness it is sufficient to prove (6.14), (6.17), (6.8), (6.9), (6.10), (6.11) and instead of (6.16) the weaker condition (6.6):

$$
\begin{aligned}
&\quad (6.14) \wedge (6.17) \wedge (6.6) \\
&\wedge\ (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\
&\Rightarrow \mathrm{ASM} \blacktriangleright \mathrm{ASM}'
\end{aligned}
$$

The proof of correctness and completeness of the refinement is the same as in Sect. 6.2.3. The only difference is, that instead of one invariant *PROP* we now need two dually defined properties *KPROP* and *VPROP*, one for the correctness, the other for the completeness proof:

$$
\begin{aligned}
&\mathrm{KPROP}(\underline{x}, \underline{x}') \equiv \\
&\exists\ \mathrm{j}.\ [\textbf{loop if} \neg\, \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
&\qquad \exists\ \mathrm{i}.\ \langle \textbf{loop if} \neg\, \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}\rangle\ \mathrm{INV}(\underline{x}, \underline{x}')
\end{aligned}
\tag{6.18}
$$

$$
\begin{aligned}
&\mathrm{VPROP}(\underline{x}, \underline{x}') \equiv \\
&\exists\ \mathrm{i}.\ [\textbf{loop if} \neg\, \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}] \\
&\qquad \exists\ \mathrm{j}.\ \langle \textbf{loop if} \neg\, \mathrm{final}'(\underline{x}')\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}\rangle\ \mathrm{INV}(\underline{x}, \underline{x}')
\end{aligned}
\tag{6.19}
$$

It should be noted, that whenever the proof mentions $\underline{x}'_k$, this state no longer denotes the unique state that can be reached from $\underline{x}'$ in k steps, but *some* arbitrary state which can be reached in k steps.

6.4.2 Diagrams of Indeterministic Size

An analysis of the proof obligation (6.14) of the previous section shows, that it does not capture the most general form of a commuting m:n diagram with $m, n > 0$ that is sufficient for the correctness of a refinement. The reason is that the proof obligation fixes the number j of rule applications of ASM′, such that from all states $\underline{x}'_j$ a state $\underline{x}_i$ must be reachable with *INV*, *prior* to the execution of ASM′.

Now it may happen, that the number j of steps necessary, does not only depend on the initial state, but also on indeterministic "guessing" steps of ASM. To illustrate this phenomenon, let us consider the following two ASMs defined by *RULE* and *RULE′*.

RULE(**var** x):
if x = 0 **then** x := 1

RULE′(**var** x):
if x = 0 **then choose** y ∈ nat **in** x := y +1 **else**
if x > 1 **then** x := x −1

Both are started in a state $x = 0$, and both terminate in a state with $x = 1$. In the first step ASM′ randomly chooses ("guesses") a natural number y, and sets x to this number plus one. The positive value of x then is decremented by the following rule applications until 1 is reached. Obviously this is equivalent to ASM, which immediately sets x to 1. Nevertheless there is *no uniform* number j of ASM′ rule applications, that reach the final state (i.e. a state equivalent to the final state of ASM). The number of rule applications is dependent on the number of x chosen in the first rule application.

If one looks at more complicated refinements, then it may be the case that not only *one* indeterministic rule applications at the beginning of a diagram determines its size, but that there are several, which influence the size. Nevertheless it is sufficient for correctness that for *each* trace of ASM′ *eventually* a state state is reached, such that *INV* holds again.

To formalize this in DL, we define an Operator $AF(\alpha, \varphi)$[1], which says, that each iterated execution of α will eventually lead to a state in which φ holds.

Using streams, as they were defined in Sect. 6.3 we can define $AF(\alpha, \varphi)$ as an abbreviation for

$$\mathrm{AF}(\alpha, \varphi) \equiv \forall\ \mathrm{s.}\ (\mathrm{Trace(s)} \wedge \underline{\mathrm{x}} = \mathrm{s[0]} \rightarrow \exists\ \mathrm{m.}\ \varphi[\underline{\mathrm{x}} \leftarrow \mathrm{s[m]}]) \tag{6.20}$$

In the formula, $\underline{x}$ are the variables modified by α, s is a stream of values of this type, and *Trace*(s) is defined by

$$\begin{aligned}&\mathrm{Trace(s)} \equiv\\ &\qquad \forall\ \mathrm{m}, \underline{\mathrm{x}}.\ \underline{\mathrm{x}} = \mathrm{s[m]} \rightarrow \langle\alpha\rangle\underline{\mathrm{x}} = \mathrm{s[m+1]}\end{aligned}$$

Instead of using streams, it is also possible to define the operator $AF(\alpha, \varphi)$ semantically:

Definition 3 $\mathcal{A}, \mathbf{z} \models AF(\alpha,\varphi)$ iff for all sequences of $(\mathbf{z}_0, \mathbf{z}_1, \ldots)$ of states for which $\mathbf{z}_0 = \mathbf{z}$ and $\mathbf{z}_i[\![\alpha]\!]\mathbf{z}_{i+1}$ hold there is an n such that $\mathcal{A}, \mathbf{z_n} \models \varphi$ holds.

To axiomatize the new operator we define two properties $AF_1(M)$ and $AF_2(M, \mathbf{z_0})$ for sets of states M. Both properties presuppose a given algebra $\mathcal{A}$, a fixed program α and a formula φ. The second property also assumes a fixed (initial) state $\mathbf{z_0}$.

$$\begin{aligned}&AF_1(M) :\Leftrightarrow \text{each state } \mathbf{z} \text{ is in } M, \text{ if } \mathcal{A}, \mathbf{z} \models \varphi \text{ holds, or if all successor states } \mathbf{z}'\\ &\text{(for which } \mathbf{z}[\![\alpha]\!]_{\mathcal{A},\mathbf{z}'} \text{ holds) are in } M\end{aligned} \tag{6.21}$$

and

$$\begin{aligned}&AF_2(M, \mathbf{z_0}) :\Leftrightarrow \text{each state } \mathbf{z} \text{ is in } M, \text{ if it is reachable from } \mathbf{z_0} \text{ (i.e. it is on a trace}\\ &\text{starting at } \mathbf{z_0}) \text{ and if } \mathcal{A}, \mathbf{z} \models \varphi \text{ holds or if all sucessor states are in } M\end{aligned} \tag{6.22}$$

For the two properties we have the following theorem:

Theorem 6 *Characterisation of* $AF(\alpha, \varphi)$
The set of states, for which $AF(\alpha, \varphi)$ holds, is equal to the intersection of all sets M, that have the property $AF_1(M)$. In a state $\mathbf{z_0}$ $AF(\alpha, \varphi)$ holds, if and only if it is in the intersection of all sets M with $AF_2(M, \mathbf{z_0})$.

[1] The term AF is from temporal logic, see e.g. [Eme90]

Proof of Theorem 6 In the proof let M_0 be the set of all states, for which $AF(\alpha, \varphi)$ holds, $M_1 := \bigcap \{M \mid AF_1(M)\}$, and $M_2(\mathbf{z_0}) := \bigcap \{M \mid AF_2(M,\mathbf{z_0})\}$. Then we obviously have $AF_1(M_0)$, which implies $M_1 \subseteq M_0$. Also for each choice of $\mathbf{z_0}$ we have that every set M with $AF_1(M)$ also has the property $AF_2(M,\mathbf{z_0})$, since (6.21) implies (6.22) for every $\mathbf{z_0}$. So each $M_2(\mathbf{z_0})$ is a subset of M_1. To complete the proof, it is therefore sufficient to show, that each $\mathbf{z'_0} \in M_0$ is also in $M_2(\mathbf{z_0})$. If this were not the case, i.e. $\mathbf{z'_0} \notin M_2(\mathbf{z_0})$ then we would have a set M with $AF_2(M,\mathbf{z_0})$ that does not contain $\mathbf{z'_0}$. Now, (6.22) implies that φ does not hold in $\mathbf{z'_0}$ and that there is a successor state $\mathbf{z'_1}$ which is not in M either. Continuing in this way, a sequence $\mathbf{z'_0}$, $\mathbf{z'_1}$, ... of states can be constructed inductively, that are all not in M (but reachable from $\mathbf{z'_0}$!), for which φ does not hold. But this is a contradiction to $\mathbf{z'_0} \in M_0$. □

The semantic definition of $AF(\alpha,\varphi)$ now immediately implies the correctness of the axiom

$$\mathrm{AF}(\alpha,\varphi) \leftrightarrow \varphi \vee [\alpha]\mathrm{AF}(\alpha,\varphi) \tag{6.23}$$

The characterization of AF as the intersection of all sets M with $AF_1(M)$ implies that axiom

$$(\forall \underline{\mathrm{x}}.\ ((\varphi \vee [\alpha]\psi) \rightarrow \psi)) \rightarrow (\mathrm{AF}(\alpha,\varphi) \rightarrow \psi) \tag{6.24}$$

is valid. The characterization with $AF_2(M,\mathbf{z})$ implies the validity of the stronger axiom

$$(\forall\ \mathrm{i}.\ [\textbf{loop}\ \alpha\ \textbf{times}\ \mathrm{i}]((\varphi \vee [\alpha]\psi) \rightarrow \psi)) \rightarrow (\mathrm{AF}(\alpha,\varphi) \rightarrow \psi) \tag{6.25}$$

This axiom allows, to restrict the states for which $(\varphi \vee [\alpha]\psi) \rightarrow \psi$ has to be shown to those, which are reachable from the initial state. Formulas (6.23) and (6.25) are sufficient to axiomatize $AF(\alpha, \varphi)$ to prove the following theorems, so we can avoid to refer to streams by using (6.20).

Using the AF operator we can now set up proof obligations for diagrams of indeterministic size by schematically replacing formulas of the form "$\exists\ i.\ [\textbf{loop}\ \alpha\ \textbf{times}\ i]\ \varphi$" with $AF(\alpha,\varphi)$. This results in the following formulas:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{mn} \\ \rightarrow \mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}'(\underline{\mathrm{x}}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{\mathrm{x}}'), \\ \qquad \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}(\underline{\mathrm{x}})\ \textbf{then}\ \mathrm{RULE}(;\underline{\mathrm{x}})\ \textbf{times}\ \mathrm{i}\rangle \\ \qquad\qquad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}')) \end{array} \tag{6.26}$$

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{mn} \\ \rightarrow \mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}(\underline{\mathrm{x}})\ \textbf{then}\ \mathrm{RULE}(;\underline{\mathrm{x}}), \\ \qquad \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}'(\underline{\mathrm{x}}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{\mathrm{x}}')\ \textbf{times}\ \mathrm{j}\rangle \\ \qquad\qquad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}')) \end{array} \tag{6.27}$$

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \wedge \mathrm{execm0}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{k} \\ \rightarrow \mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}(\underline{\mathrm{x}})\ \textbf{then}\ \mathrm{RULE}(;\underline{\mathrm{x}}), \\ \qquad (\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \\ \qquad\ \wedge (\neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \rightarrow \mathrm{execm0}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') < \mathrm{k}))) \end{array} \tag{6.28}$$

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \wedge \mathrm{exec0n}(\underline{x}, \underline{x}') = \mathrm{k} \\ \rightarrow \mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}'), \\ \qquad (\quad \mathrm{INV}(\underline{x}, \underline{x}') \\ \qquad\ \wedge\ (\neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \rightarrow \mathrm{exec0n}(\underline{x}, \underline{x}') < \mathrm{k}))) \end{array} \tag{6.29}$$

Theorem 7 *Modularisation Theorem for Unbounded Indeterminism*
Given a refinement of ASM to ASM′, a predicate *INV* and functions *ndt*, *exec0n*, *execm0*, such that all proof obligations (6.26), (6.27), (6.28), (6.29), (6.8), (6.9), (6.10), (6.11) can be shown, then the refinement is correct and complete:

$$\begin{array}{l} \quad (6.26) \wedge (6.27) \wedge (6.28) \wedge (6.29) \\ \wedge\ (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\ \Rightarrow \mathrm{ASM} \bowtie \mathrm{ASM}' \end{array}$$

To prove trace correctness it is sufficient to prove (6.26), (6.28), (6.8), (6.9), (6.10), (6.11) and instead of (6.29) the weaker property (6.7). For correctness the condition, that *exec0n* decreases, can be dropped from condition (6.7).

$$\begin{array}{l} \quad (6.26) \wedge (6.28) \wedge (6.7) \\ \wedge\ (6.8) \wedge (6.9) \wedge (6.10) \wedge,(6.11) \\ \Rightarrow \mathrm{ASM} \blacktriangleright \mathrm{ASM}' \end{array}$$

The formal proofs in KIV do not change, only the definition of *KPROP* and *VPROP* has to be modified:

$$\begin{array}{l} \mathrm{KPROP}(\underline{x}, \underline{x}') \equiv \\ \mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}'), \\ \qquad \exists\ \mathrm{i}.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i} \rangle\ \mathrm{INV}(\underline{x}, \underline{x}')) \end{array}$$

$$\begin{array}{l} \mathrm{VPROP}(\underline{x}, \underline{x}') \equiv \\ \mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x}), \\ \qquad \exists\ \mathrm{j}.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j} \rangle\ \mathrm{INV}(\underline{x}, \underline{x}')) \end{array}$$

Instead of the axioms (4.3) for loops the axioms (6.23) and (6.25) for the AF operator are used. Since the AF operator currently is available in KIV only as an abbreviation, the proof of the modularisation theorem requires some more effort as in the deterministic case (466 proof steps and 94 interactions). The formal specifications and the proved theorems and lemmas can be found in appendix C.3.

We want to finish this section with some further remarks on the definition of the AF operator; AF can not be defined uniformly as an abbreviation in DL (the extension of DL with streams is not uniform, since the datatype of streams depends on the types of the variables modified by α), since $AF(\alpha,\varphi)$ is equivalent to the statement: The program $AF\#$, defined by ($\underline{x}$ are the variables occuring in α)

AF#(;**var** $\underline{x}$)
begin
if φ **then**
  begin
  α;
  AF#(;$\underline{x}$)
  end
end

always terminates. Now the fact, that an indeterministic program always terminates, cannot be expressed in DL in general (see [Gol82]). But there is a special case, in which this is possible nevertheless:

Theorem 8 *Bounded Indeterminism*
If α is an always termitating program with only *bounded* indeterminism, i.e. if for every state $\mathbf{z}$ there are only finitly many successor states $\mathbf{z}'$ with $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$, then:

$$\mathrm{AF}(\alpha,\varphi) \leftrightarrow \exists\ \mathrm{j}.\ [\textbf{loop if} \neg\ \varphi\ \textbf{then}\ \alpha\ \textbf{times}\ \mathrm{j}]\ \varphi$$

Proof of Theorem 8 For the proof from left to right (the other direction is trivial) one has to consider all traces from a fixed initial state $\mathbf{z}$ such that for all states on the trace $\neg\ \varphi$ holds. These traces form a tree structure, that according to the precondition has no infinite paths. Since α has only bounded indeterminism, the tree is finitely branching. Now König's Lemma from set theory (see e.g. [Knu73], p. 381–383) implies that the tree is finite. The length of each path (trace) is bounded by the depth d of the tree. Therefore $j := d + 1$ is sufficient to prove the formula on the right hand side of the equivalence.

Always terminating programs, that have only bounded indeterminism, result from the translation of distributed ASMs to DL. In contrast to the ASM from the beginning of the section, which could choose one of *infinitely* many natural numbers, a distributed ASM has only bounded indeterminism, since it always chooses from *finitely* many agents. Therefore we do not need the AF operator in the case of distributed ASMs.

For the proof obligations this means that we can keep the old proof obligations from the previous section. Only the tests $\neg$ *final*($\underline{x}$) resp. $\neg$ *final'*($\underline{x'}$) of the Box-Formulas with *loop* constructs have to be replaced by the more complex tests

$$\neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \neg\ \varphi$$

where φ is the post condition of the loop. This exploits that we allow arbitrary formulas in the tests of conditions.
As an example we consider ASM from the beginning of the section and ASM$'$ with the rule:

RULE$'$(**var** x):
if x = 0 **then choose** b **in**
if b **then** x := 3
else x:= 2
else if x > 1 **then** x := -1

ASM$'$ now chooses the value of x indeterministically to be 2 or 3 — now there are finitely many choices. Therefore it is sufficient to show

$$\begin{array}{l}\exists\ \mathrm{i}.\ [\textbf{loop if} \quad \neg\ \mathrm{x} = 1\\ \qquad\qquad \wedge \neg\ \exists\ \mathrm{j}.\ \langle\textbf{loop if}\ \neg\ \mathrm{x}' = 1\ \textbf{then}\ \mathrm{RULE}'(;\mathrm{x}')\rangle\ \mathrm{x} = \mathrm{x}'\\ \quad \textbf{then}\ \mathrm{RULE}(;\mathrm{x})]\\ \qquad \exists\ \mathrm{j}.\ \langle\textbf{loop if}\ \neg\ \mathrm{x}' = 1\ \textbf{then}\ \mathrm{RULE}'(;\mathrm{x}')\rangle\ \mathrm{x} = \mathrm{x}'\end{array}$$

for correctness. This is possible with $i = 3$ und $j = 1$.

6.5 Extensions for Iterated Refinement

In this section we are concerned with the problem, that the systematic translation of a programming language to assembler code often requires several refinements, that introduce orthogonal concepts. Now, in the verification of two successive refinements ASM $\triangleright$ ASM$'$ $\triangleright$ ASM$''$ we often get coupling invariants INV and INV' which have many common parts (we will see examples

in the Prolog-WAM case study in Sect. 17.2 and 18). The common parts consist of properties of ASM′, which are relevant for both equivalence proofs. If $MINV'(\underline{x}')$ is a common part of INV and INV' our current modularization theorem requires, that $MINV'(\underline{x}')$ is shown in both refinements to be invariant in ASM′. In this section we give a generic method, that allows us to avoid this duplication of proofs. We assume that the equivalence of ASM and ASM′ has already been proven with a coupling invariant INV. Then it is easy to see, that the formula

$$\exists \underline{x}.\ \mathrm{INV}(\underline{x}, \underline{x}') \tag{6.30}$$

holds in all states of ASM′, which are at the "corners" of commuting diagrams of the refinement. Now usually it is simple to characterize these states by a predicate $MINVNOW'(\underline{x}')$, which consists of a disjunction of ASM′ rule tests. Then the formula MINV′ defined as

$$\mathrm{MINVNOW}'(\underline{x}') \rightarrow \exists \underline{x}.\ \mathrm{INV}(\underline{x}, \underline{x}') \tag{6.31}$$

is an invariant of ASM′. Since every weaker formula is also an invariant, one will usually choose a formula that is implied by (6.31) and that does not mention the variables $\underline{x}$ of ASM anymore.

To make sure, that $MINVNOW'$ does indeed characterize the corners of diagrams, we must strengthen the conditions of the correctness proof of the refinement from ASM to ASM′ (the completeness proof can be left unchanged). In the following we show, how this has to be done in the indeterministic case without diagrams of indeterministic size. The special case of deterministic ASMs (Diamonds instead of Boxes) and the generalization to diagrams of indeterministic size (AF operator instead of Boxes) are treated as in the previous sections.

The two necessary change are to strengthen the rule tests of ASM′ with the additional condition $\neg\ MINVNOW'(\underline{x}')$, and to additionally require $MINVNOW'(\underline{x}')$ in the post condition. This assures, that ASM rules are applied as long as $\neg\ MINVNOW'(\underline{x}')$ holds. For m:n and 0:n diagrams this changes conditions (6.14) and (6.17) to

$$\begin{array}{l}
\quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \mathrm{MINVNOW}'(\underline{x}') \wedge \neg\ \mathrm{final}'(\underline{x}') \\
\wedge\ \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{mn} \\
\rightarrow [\textbf{if} \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ ] \\
\quad \exists\ \mathrm{j}.\ [\textbf{loop if} \neg\ \mathrm{final}'(\underline{x}') \wedge \neg\ \mathrm{MINVNOW}'(\underline{x}') \\
\qquad \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
\qquad (\quad \mathrm{MINVNOW}'(\underline{x}') \\
\qquad \wedge\ \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(\underline{x})\ \textbf{times}\ \mathrm{i}\rangle \\
\qquad\qquad \mathrm{INV}(\underline{x}, \underline{x}'))
\end{array} \tag{6.32}$$

$$\begin{array}{l}
\quad \mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{MINVNOW}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \\
\wedge\ \mathrm{exec0n}(\underline{x}, \underline{x}') = \mathrm{k} \\
\rightarrow [\textbf{if} \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(\underline{x}')] \\
\quad \exists\ \mathrm{j} > 0.\ [\textbf{loop if} \neg\ \mathrm{final}'(\underline{x}') \wedge \neg\ \mathrm{MINVNOW}'(\underline{x}') \\
\qquad \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
\qquad (\quad \mathrm{MINVNOW}'(\underline{x}') \wedge \mathrm{INV}(\underline{x}, \underline{x}') \\
\qquad \wedge\ (\quad \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \\
\qquad\qquad \rightarrow \mathrm{exec0n}(\underline{x}, \underline{x}') < \mathrm{k}))
\end{array} \tag{6.33}$$

The proof obligation for m:0 diagrams (6.6) is unchanged. With the new proof obligations the following theorem can be shown.

Theorem 9 *Iterated Refinement of ASMs.*
The proof obligations (6.32), (6.33),(6.6) (6.8), (6.9), (6.10) and (6.11) imply in addition to the correctness and trace correctness of the refinement from ASM to ASM′ that every formula $MINV'(\underline{x}')$, which satisfies

$$(\text{MINVNOW}'(\underline{x}') \rightarrow \exists \underline{x}.\ \text{INV}(\underline{x}, \underline{x}')) \rightarrow \text{MINV}'(\underline{x}')$$

is an invariant of ASM′. Formally it can be proved that

$$\begin{array}{l}(\exists \text{ st. IN(st, st')}) \\ \rightarrow \forall \text{ j. [}\textbf{loop if } \neg \text{ final'(st') } \textbf{then } \text{RULE}'(; \text{ st') } \textbf{times } \text{j]}\ \text{MINV}'(\underline{x}')\end{array}$$

holds

So, $\text{MINV}'(\underline{x}')$ is true for all states during any run of ASM′, provided that the initial state is related to some initial state of ASM with the *IN* relation (usually a trivial assumption). The proof for refinement correctness follows the same lines as before, only the definition of *KPROP* has to be changed to

$$\begin{array}{l}\text{KPROP}(\underline{x}, \underline{x}') \equiv \\ \exists \text{ j. [}\textbf{loop if } \neg \text{ final}'(\underline{x}') \wedge \neg \text{ MINVNOW}'(\underline{x}') \\ \quad \textbf{then } \text{RULE}'(\underline{x}') \textbf{ times } \text{j]} \\ \quad (\ \ \text{MINVNOW}'(\underline{x}') \\ \quad \wedge \exists \text{ i. } \langle \textbf{loop if } \neg \text{ final}(\underline{x}) \textbf{ then } \text{RULE}(\underline{x}) \textbf{ times } \text{i}\rangle \\ \qquad\quad \text{INV}(\underline{x}, \underline{x}'))\end{array} \tag{6.34}$$

The invariance of *KPROP* in ASM′ immediately implies the invariance of

$$\text{MINVNOW}'(\underline{x}') \rightarrow \exists \underline{x}.\ \text{INV}(\underline{x}, \underline{x}')$$

in ASM′. So the weaker formula MINV′ is an invariant too.

MINV′ can now be used in the proof obligations for the refinement from ASM′ to ASM″ as an additional precondition. Using invariants as additional preconditions can be iterated by defining another predicate MINVNOW″ for the refinement from ASM′ to ASM″. Then the refinement proof will give another invariant MINV″ for ASM″, which can be used in the next refinement.

Appendix C.4 defines the proof obligations for refinement correctness for the case, that we already have an invariant $MINV(\underline{x})$ for ASM and want to construct an invariant $MINV'(\underline{x})$ for ASM′. The proof in KIV required 502 proof steps and 89 interactions. The proof obligations shown above are the special case, in which no invariant for ASM is given (i.e. the case in which $MINV(\underline{x})$ is simply set to *true*).

6.6 Related Work

Most known work on equivalence proofs for ASMs is from the field of compiler verification. In most cases, the interpreters are not defined using the ASM formalism, but some are equivalent.

In work on compiler verification, the case of 1:1 diagrams is by far the most common case. Often several variants are discussed, where *IN*, *OUT* and *INV* are functions in one direction or the other (e.g. in [BHMY89]). As a generalization, often the case of 1:n diagrams with $n > 0$ is considered. This case often occurs, when one instruction of the source language has to be implemented by several instructions of the target language. This generalization of data refinement is only marginal, since the proof of refinement correctness can still be done directly by induction on the number of executed ASM rules.

An example of a formal verification of a compiler, in which 1:n diagrams occur, is the verification of the compilation if an imperative programming language (GYPSY), that was translated in

several refinements first to a high-level assembler language (Piton) and then in machine code of the FM8502 processor. The verification which was done with NQTHM is described in ([BM79], [BM88]). Since NQTHM does not allow existential quantification, the number n of steps of ASM$'$ that is necessary to simulate m steps of ASM is computed by a skolem function as $n = clock(m, st_0)$, where st_0 is the initial state of ASM.

A similar skolem function (*num_non_visible*) is also used in [Cyr93]. The correctness notion used there is trace correctness for sequential ASMs with respect to an abstraction function *abstr*. All states of the abstract ASM are required to be visible. This corresponds to a restriction of 1:n with $n > 0$ for the possible diagram forms. The paper sketches two proof techniques. The first ("speeding up the implementation machine") corresponds to the direct verification of the 1:n diagrams with the coupling invariant

$$\mathrm{INV}(\underline{x},\underline{x}') \equiv \mathrm{visible}(x') \rightarrow \mathrm{abstr}(\underline{x}') = \underline{x}$$

The used function *visible_I*, that encodes *num_non_visible* many steps of ASM$'$ into one steps, corresponds exactly to our

loop if $\neg$ final$'(\underline{x}')$ **then** RULE$'(;\underline{x}')$ **times** num_non_visible$(\underline{x}')$

The second proof technique ("slowing down the Specification Machine") splits the 1:n diagrams into one 1:1 and n-1 1:0 diagrams ("stuttering steps"), that are verified separately. The "termination" condition used there corresponds to our requirement, that the *exec0n* function must decrease. The approach sketched in the paper seems to require the explicit introduction of time in the specification. The outlook of [Cyr93] gives as desirable extensions indeterminism, stuttering of both machines (i.e. 0:n and n:0 diagrams in one refinement), and iterated refinement ("hierarchical decomposition"), that we all have treated in this work.

Arbitrary m:n diagrams with $m, n > 0$ are roughly sketched in [McG72]. The paper assumes determinism and a coupling invariant $INV(\underline{x}, \underline{x}')$ that has the special form $f_1(\underline{x}) = f_2(\underline{x}')$.

A formal treatment of m:n diagrams with $m, n > 0$ has been worked out in parallel to this work in [Dol98]. The paper generalizes the approach of [Cyr93] by using two *num_non_visible* functions (one for each ASM). Indeterminism is considered, but only bounded indeterminism (for unbounded indeterminism it is impossible to define a function *num_non_visible*). Also still abstraction functions are used.

Another new work on ASM refinement in compiler verification is [ZG97]. The correctness notion given there is only defined semantically (there is no logic for formal verification). As the only approach known to us it uses a *relation* ρ instead of an abstraction function between the states of both ASMs. The relation corresponds to the semantics of our coupling invariant *INV*.

The correctness notion is based on equivalence (modulo an abstraction function) of the output that is made during two ASM runs. Output is defined implicitly as changes of the values of certain output variables. To formalize this correctness notion in our setting, it is necessary to modify the ASMs so that they collect the outputs in a list *outputlist* (we introduce a "history variable" in the sense of [AL91]). Then the correctness notion of [ZG97] is equivalent to trace correctness with

$$\mathrm{IN}(\underline{x},\underline{x}') \equiv \mathrm{outputlist} = \mathrm{outputlist}' = []$$
$$\mathrm{OUT}(\underline{x},\underline{x}') \equiv \mathrm{map(abstr,outputlist}') = \mathrm{outputlist}$$

(this corresponds to the conditions of Theorem 4 for the relation ρ). [ZG97] also gives a modularization theorem (Theorem 5, "Horizontal Decomposition"). The idea is also to decompose the whole diagram into subdiagrams. The decomposition requires, that each subdiagram contains at most one rule that produces output. If one depicts a rule application with output by a continuous arrow, an arbitrary number of rule applications with no output as a dotted arrow, then Fig. 6.9 gives a visualization of the proof obligations.

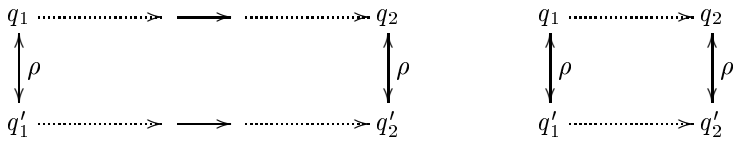


Figure 6.9 : Modularization according to [ZG97]

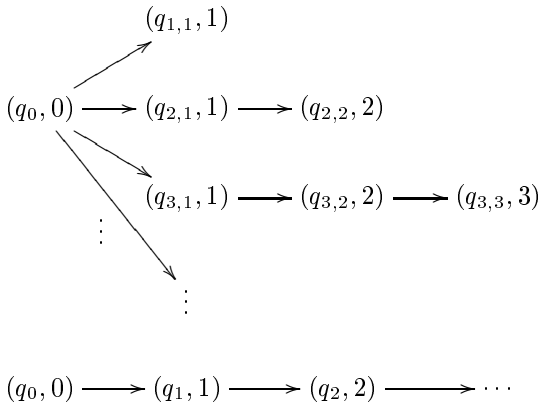


Figure 6.10 Incorrect refinement with unbounded indeterminism

But the theorem is not correct for several reasons: First, it is possible to verify incorrect refinements with infinite sequences of m:0 diagrams like in diagram 6.6 (see Sect. 6.2.2). Second, some implicit assumptions are missing. Finally, the formalization (accidently) excludes 1:n diagrams with $n > 1$.

The assumptions that are present in the examples, but not explicitly stated are, that external functions do not cause unbounded indeterminism and that the outputs are collected in an *output-list* as above. Without these assumptions the counter examples shown in Fig. 6.10 and in Fig. 6.11 can be constructed: the figures present the ASMs as automata with two program variables. The first stores the internal state, the second stores the current output. Figure 6.11 shows the unpleasant possibilities of unbounded indeterminism, that made the introduction of the AF operator in Sect. 6.4 necessary. Figure 6.10 exploits, that the possibility of a state transition from q_1' to q_2' with one output does not imply that there is one output on all paths from q_1' to q_2'.

m:n diagrams with $n > 1$ (especially 1:n diagrams which often show up in applications) are ruled out by the formalization, since it is required that the diagrams shown in Fig. 6.9 commute for *every* q_2' (especially for each direct successor of q_1') and not only for *some arbitrary* successor on each path starting at q_1', as our theorem requires.

If one adds the implicit assumptions to the theorem and excludes infinite sequences of m:0 diagrams, then it can be shown that Theorem 5 from [ZG97] is a special case of Theorem 5, p. 35. The problem of infinite sequences of 0:n diagrams does not occur, since the theorem does allow only 0:n diagrams that can be extended to a 1:n diagram: therefore we can always choose $ndt(\underline{x}, \underline{x}') \neq 0n$.

Figure 6.11 Incorrect refinement with no *outputlist*

Chapter 7

Peephole Optimization

In this section we will apply the modularization theorem for correctness proofs of ASM refinements to "peephole optimization" of program code (usually assembler code). The idea of such an optimization is to walk with a window of fixed size (the "peephole") over a piece of program code, thereby replacing inefficient sequences of instructions with more efficient ones.

Sect. 7.1 first gives a general approach for the case, when the optimized instructions do not contain any jump instructions (but the whole code may contain jumps). It is shown, that the conditions necessary for correctness can be defined simply by instantiating the modularization theorem for ASM refinements.

The idea of a general approach for the verification of peephole optimizations was taken from [DvHPR97], which consists of 2 parts. The first part formalizes peephole optimization and proves, that certain proof obligations are sufficient for correctness. The second part then verifies a number of example optimizations, which were taken from [TvS82].

Sect. 7.2 shows, that our approach generalizes the one given in [DvHPR97]. Although both approaches are generic in the sense, that they do not fix a set of instructions, [DvHPR97] requires the program code to be a list of instructions which are executed sequentially. This is not realistic, since real assembler code always contains jump instructions. The restriction to linear code without jumps can not be removed easily since the proof for correctness of peephole optimization essentially depends on induction over the length of the instruction list.

In contrast to the restriction to linear code for the approach in [DvHPR97], we show that our approach can also handle the examples with jump instruction from [TvS82] by just a minimal change to the coupling invariant. The reason is, that the examples all fall into the special case, where only the *last* instruction of an optimized instruction sequence is a jump instruction. Finally we discuss with a simple example, that optimizations of instruction sequences with jumps in the middle can also be verified, by simply splitting the diagrams, which are required to commute, at the jump instructions.

7.1 Formalization of Peephole Optimization

We first need to formalize a general interpreter as an ASM. We assume, that the program code is stored in a memory db (we do not consider self-modifying code, therefore db is a constant), and that with $code(pc,db)$ we can select the instruction at an address stored in a program counter pc. An ASM rule $RULE$ executes a given instruction $i = code(pc,db)$, and thereby modifies a program state st and the program counter pc. To allow erroneous execution of instructions (e.g. division by zero, or an attempt to get the top element of an empty stack) we assume that a predicate $ok(pc,st,db)$ is defined. The predicate should hold, iff execution of the next instruction $code(pc,db)$ does not lead to an error. We assume that $RULE$ is not applicable, when $ok(pc,st,db)$ does not hold. Finally, we assume a special instruction *halt*, which indicates the end of the program.

Since we want to consider code with jump instructions, we do not require that each instruction

increments *pc*. Nevertheless such instructions, called *linear* instructions, are important in the following. We define the following auxiliary functions and predicates for them:

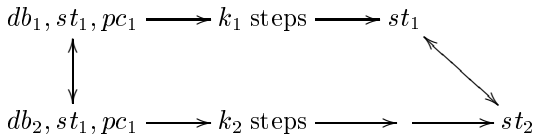
$$\text{instrs(pc,db,n)} = [\text{code(pc,db)}, \ldots, \text{code(pc} + \text{n} - 1\text{ ,db)}]$$

$$\begin{array}{l} \quad \text{lin(i)} \\ \leftrightarrow \forall\ \text{pc,pc}_0\text{,db,st.} \quad \text{code(pc,db)} = \text{i} \wedge \text{pc} = \text{pc}_0 \wedge \text{ok(pc,st,db)} \\ \qquad\qquad\quad \rightarrow \langle \text{RULE\#(db;pc,st)} \rangle\ \text{pc} = \text{pc}_0 + 1 \end{array}$$

$$\text{linear(pc,db,n)} \leftrightarrow \forall\ \text{k.}\ 0 \leq \text{k} < \text{n.}\ \text{lin(pc} + \text{k,db)}$$

instrs(pc,db,n) computes the list of n instructions that follow *pc*. *lin(i)* states, that the instruction i is linear, i.e. that it will increment *pc* regardless of the state in which it is executed. *linear(pc,db,n)* says, that all instructions in *instrs(pc,db,n)* are linear, and therefore will be executed in the interpreter in this order. Such linear instruction sequences will be replaced by more efficient ones in peephole optimization.

For the definition of peephole optimization we define a predicate *peephole*(st_1, pc_1,db_1,k_1, il_2), that should hold iff the instructions *instrs*(db_1,pc_1,k_1) that are executed in state db_1,st_1,pc_1 of the ASM can be equivalently replaced by il_2. If k_2 denotes the length of il_2, then the requirement corresponds intuitively to the commutativity of the k_1:k_2 diagram



Formalized in Dynamic Logic this is the requirement, that

$$\begin{array}{l} \quad \text{I(db}_1\text{,st}_0\text{,pc}_0\text{)} \\ \wedge\ \exists\ \text{i.}\ \langle \textbf{loop}\ \text{RULE(db}_1\text{;pc}_0\text{,st}_0\text{)}\ \textbf{times}\ \text{i} \rangle\ (\text{pc}_0 = \text{pc}_1 \wedge\ \text{st}_0 = \text{st}_1) \\ \wedge\ \text{peephole(st}_1\text{,pc}_1\text{,db}_1\text{,k}_1\text{,il}_2\text{)} \\ \wedge\ \text{db}_2 = \text{repl(pc}_1\text{,db}_1\text{,k}_1\text{, il}_2\text{)} \\ \wedge\ \text{pc}_1 = \text{pc}_2 \wedge\ \text{st}_1 = \text{st}_2 \\ \rightarrow \quad \text{linear(instrs(pc}_1\text{,db}_1\text{,k}_1\text{))} \\ \wedge\ \text{linear(il}_2\text{)} \\ \wedge\ \langle \textbf{loop}\ \text{RULE(db}_1\text{;pc}_1\text{,st}_1\text{)}\ \textbf{times}\ \text{k}_1 \rangle \\ \qquad \langle \textbf{loop}\ \text{RULE(db}_2\text{;pc}_2\text{,st}_2\text{)}\ \textbf{times}\ \text{k}_2 \rangle\ \text{st}_1 = \text{st}_2 \end{array} \tag{7.1}$$

holds. The precondition

$$\begin{array}{l} \quad \text{I(db}_1\text{,st}_0\text{,pc}_0\text{)} \\ \wedge\ \exists\ \text{i.}\ \langle \textbf{loop}\ \text{RULE(db}_1\text{;pc}_0\text{,st}_0\text{)}\ \textbf{times}\ \text{i} \rangle\ (\text{pc}_0 = \text{pc}_1 \wedge\ \text{st}_0 = \text{st}_1) \end{array}$$

in the formula states, that the state (pc_1,st_1) is reachable from an initial state (st_0,pc_0) specified with a predicate I. The precondition is often unnecessary, since usually the diagram commutes for all states (pc_1,st_1).

The linearity conditions for *instrs*(pc_1,db_1,k_1) resp. il_2 make sure, that the instructions are really executed before resp. after the optimization. Function *repl*(pc_1,db_1,k_1,il_2) actually replaces the instructions *instrs*(pc_1,db_1,k_1) by il_2. We must have

$$\begin{array}{l} \quad \mathrm{db}_2 = \mathrm{repl}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2) \\ \rightarrow \forall\ \mathrm{k}.\ \mathrm{k} < \mathrm{k}_2 \rightarrow \mathrm{code}(\mathrm{pc}_1{+}\mathrm{k},\mathrm{db}_2) = \mathrm{get}(\mathrm{k},\mathrm{il}_2) \end{array} \tag{7.2}$$

This definition says, that the new program stores the new instructions at the addresses *pc*, *pc +1*, ... *(pc*$_1$ *+k*$_2$ *-1)*. But this is not sufficient. We must also make sure, that the resulting code has no gaps by moving code by $k_2 - k_1$. Also the addresses of jump instructions must be updated. Since we do not want to go into details of jump instructions, we simply require for the result of *repl*, that each moved instruction at $pc' = shift(pc,pc_1, k_2 - k_1)$ has the same effect as the original instruction at *pc*:

$$\begin{array}{l} \quad \mathrm{db}_2 = \mathrm{repl}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2) \\ \wedge\ (\mathrm{pc} < \mathrm{pc}_1 \vee \mathrm{pc} \geq \mathrm{pc}_1 + \mathrm{k}_1) \\ \wedge\ \mathrm{pc}' = \mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\mathrm{k}_2 - \mathrm{k}_1) \wedge \mathrm{st} = \mathrm{st}' \\ \rightarrow \langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st})\rangle\ \langle \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}',\mathrm{st}')\rangle \\ \qquad\qquad (\mathrm{pc}' = \mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\ \mathrm{k}_2 - \mathrm{k}_1) \wedge \mathrm{st} = \mathrm{st}') \end{array} \tag{7.3}$$

In the formula *shift* is defined as

$$\mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\mathrm{n}) = \begin{cases} \mathrm{pc}, \text{ when } \mathrm{pc} < \mathrm{pc}_1 \\ \mathrm{pc} + \mathrm{n}, \text{ otherwise} \end{cases}$$

For some peephole optimization to be applicable on db_1, pc_1 and il_2 we require that the predicate $peephole(st,pc,db_1,k_1,il_2)$ holds in every state *st*, the ASM can reach. Formally

$$\begin{array}{l} \quad \mathrm{IN}(\mathrm{db}_1,\mathrm{pc}_0,\mathrm{st}_0) \\ \wedge\ \exists\ \mathrm{i}.\ \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db};\mathrm{pc}_0,\mathrm{st}_0)\ \textbf{times}\ \mathrm{i}\rangle\ (\mathrm{pc}_0 = \mathrm{pc}_1 \wedge \mathrm{st}_0 = \mathrm{st}_1) \\ \rightarrow \mathrm{peephole}(\mathrm{st}_1,\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2)) \end{array} \tag{7.4}$$

(7.1) gives a condition for the optimization of a sequence of instructions. It is *local*, since only the instructions at the addresses between pc_1 and $pc_1 + k_1$ are relevant. For program code, that does not contain jump instructions, this condition is already sufficient to assure, that the considered instructions can be replaced by more efficient ones in *every* program. But for programs with jumps we need an additional condition: No instruction in the surrounding program must jump in the middle of the optimized code. This can be formalized with a predicate *notjumpedto*:

$$\begin{array}{l} \quad \mathrm{notjumpedto}(\mathrm{pc}_1,\mathrm{k}_1,\mathrm{db}) \\ \leftrightarrow \forall\ \mathrm{st},\ \mathrm{pc}.\quad \neg\ \mathrm{pc}_1 \leq \mathrm{pc} < \mathrm{pc}_1 + \mathrm{k}_1 \\ \qquad\qquad \rightarrow \langle \mathrm{RULE}(\mathrm{db};\mathrm{pc},\mathrm{pc})\rangle\ \neg\ \mathrm{pc}_1 < \mathrm{pc} < \mathrm{pc}_1 + \mathrm{k}_1 \end{array} \tag{7.5}$$

Now we can prove the following theorem:

Theorem 10 Given a general interpreter formalized as an ASM (as above), a predicate *peephole* and values db_1, pc_1, k_1, il_2 such that (7.1), (7.4) and $notjumpedto(pc_1, k_1, db_1)$ hold, then the modification of db_1 to $repl(db_1,pc_1,k_1,il_2)$ (where *repl* is specified as in (7.2) and (7.3) is a correct and complete refinement of ASM.

For the proof we decompose runs of both the original ASM with code db_1 and of the optimized ASM with code $db_2 = repl(db_1,pc_1,k_1,il_2)$ into 1:1 diagrams as long as $pc \neq pc_1$, and into a $k_1{:}k_2$ diagram for the optimized Code. As the coupling invariant we use the conjunction of the following four formulas.

$$\exists\ \mathrm{pc}_0,\mathrm{st}_0,\mathrm{i}.\quad \mathrm{I}(\mathrm{db}_0,\mathrm{pc}_0,\mathrm{st}_0) \wedge \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_0;\mathrm{pc}_0,\mathrm{st}_0)\ \textbf{times}\ \mathrm{i}\rangle\ (\mathrm{pc}_0 = \mathrm{pc}_1 \wedge \mathrm{st}_0 = \mathrm{st}_1)$$

$$\mathrm{db}_2 = \mathrm{repl}(\mathrm{db}_1,\mathrm{pc}_1,\mathrm{k}_1,\mathrm{il}_2)$$

$$\neg\ \mathrm{pc}_1 < \mathrm{pc} < \mathrm{pc}_1 + \mathrm{k}_1$$

$$\mathrm{pc}' = \mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\mathrm{k}_2 - \mathrm{k}_1) \wedge \mathrm{st}' = \mathrm{st}$$

According to the proof obligations for the equivalence of ASMs from Chapter 6, we have to show that all four formulas are invariant in the following $k_1{:}k_2$ diagram, whenever $pc = pc_1$, and we have to show that they are invariant in the following 1:1 diagram otherwise.

For the first two formulas this is simple. The first is a trivial invariant of the original ASM, which says that each intermediate state is reachable from the initial state.

The second formula is the compiler assumption between the program codes. It is obviously invariant, since it does mention values that are modified by the ASM.

The third formula states, that pc is not *within* the optimized piece of code ($pc = pc_1$ is possible), and the fourth gives the connection between the states (pc,st) and (pc',st') derof the two ASMs.

Their invariance follows from (7.1) for a $k_1{:}k_2$ diagram, since all preconditions are part of the invariant, except $peephole(st,pc_1,db_1,k_1,il_2)$, which follows directly from (7.4): linearity of the instructions implies that at the end of the diagram $pc = pc_1 + k_1$ and $pc' = pc_1 + k_2$, so we have indeed $pc' = shift(pc,pc_1,k_2 - k_1)$.

For a 1:1 diagram the third formula is invariant because we required $notjumpedto(pc_1, k_1, db_1)$ (no jumps into the optimized code), and the invariance of the fourth formula is due to assumption (7.3) for the *repl* function.

Finally, to show all proof obligations defined in Chapter 6 for the equivalence of the ASMs, we have to show that the coupling invariant holds in initial states. The only nontrivial formula of the coupling invariant here is the third, so we just have the requirement that ASM does not start execution within the optimized code. Note that m:0 or 0:n diagrams, which occur for $k_1 = 0$ or $k_2 = 0$, are no problem here, since several successive ones are impossible. Also note, that the coupling invariant trivially implies that both ASMs finish in a state with $st = st'$.

Summarizing, correctness of peephole optimization is a special case of the modularization theorem for ASM refinements, when the optimized code does not contain jump instructions. Jumps in the optimized code will be considered in the section after the next.

7.2 Comparison to the Formalization in PVS

In this section we give a short comparison of our formalization to the one defined in [DvHPR97].

A main technical difference is that [DvHPR97] gives a formalization of the semantics of an interpreter (function *interprete*) and the equivalence of interpreters (predicate $\equiv$) that is specialized for peephole optimization, while we have just instantiated the general notions of ASMs and ASM refinement.

A severe restriction of the formalization in [DvHPR97] is, that only program code without jump instructions is considered. The restriction allows to avoid a program counter pc, and by formalizing program code as a list of instructions, proofs by induction over the length of the list are possible. Such an induction is not possible when jump instructions are present.

The necessary conditions for the correctness of peephole optimization are the same for both formalizations, except that our formalization has the two obvious additional requirements

- The program must not start in the middle of optimized code.

- There must not be jump instructions that point into optimized code.

that result from the generalization to code with jumps.

Some technical points of our definition are less restrictive (but also less concrete) than in [DvHPR97]. We have avoided to define schemes for optimization rules by giving a more precise definition of the *peephole* predicate. We therefore define here, how to specialize our definitions to the ones given in [DvHPR97]:

A rule scheme from [DvHPR97], p. 4, Fig. 1 corresponds to a specialization of the ASM rule to the form

if code(pc,db) = $\mathrm{i_k}$ $\wedge$ admissible($\mathrm{i_k}$)(st)
then pc,st := effect($\mathrm{i_k}$)(pc +1,st)

for every instruction i_k. So it is clear, that the globally defined functions *admissible* and *effect* are defined only to encode the semantics of a deterministic rule application functionally (our formalization avoids this restriction to a deterministic ASM). The implicit restriction, that *pc* is incremented, is given explicitly in our ASM rule. Our predicate *ok(pc,st,db)* corresponds to *admissible(code(pc,db),pc,st)*.

The function *interprete* corresponds to the semantics of the ASM: if the result is the empty set, then our formalization stops in a state *st*, where *ok(pc,st,db)* does not holds. The definition of the "==" in Fig. 4, p. 5 is identical to our definition of equivalence of ASMs, where *IN* and *OUT* are identity on *pc* (modulo *shift*) and *st*.

Our predicate *peephole* is very abstract. [DvHPR97] gives a more concrete definition: It is based on a list of rules $[R_1, \ldots R_n]$ with the form $R_i = (p_i, r_i, c_i)$. Each rule consists of three parts:

- A first list p_i ("patterns") of instructions, that should be replaced.
- A second list r_i ("replacements") of instructions, that will replace the p_i.
- A predicate c_i ("condition"), that characterizes the states, in which the rule is applicable.

This corresponds in our formalization to a definition of n predicates $peephole_1, \ldots, peephole_n$ defined by

$$\mathrm{peephole_i(st,pc,db,k_1,il_2)} : \leftrightarrow \quad \mathrm{instrs(pc,db,k_1)} = \mathrm{p_i} \wedge \mathrm{il_2} = \mathrm{r_i} \wedge \mathrm{c_i(st)}$$

The rules are applied sequentially to the initial program (the correctness of all optimizations is then by transitivity of program equivalence). We thought the definition in [DvHPR97] to be too specific, since there is no pattern matching done between the patterns of a rule and the actual code (it seems that for every instance a new rule has to be given), and the predicates c_i do not mention the code that is executed before *pc* is reached: whether the test for c_i holds, and rule R_i can be applied, can be decided *only* by inspecting all reachable states, which is practically impossible. In contrast, our definition of a *peephole* predicate makes it possible to use arbitrary syntactic conditions in the applicability condition. Also arbitrary patterns and pattern matching are still possible. Since the concrete definition of pattern matching as well as syntactic applicability conditions depend on the concrete program code, we have left the definition of the predicate *peephole* abstract.

7.3 Optimizations of Jump Instructions

In this section we consider optimizations of instructions with jumps. We will not give a generic method for verification, but the given examples should make it obvious, that jump instructions can be easily handled using the modularization theorem. Only the number of commuting diagrams that which to be considered increases with the number of jump instructions.

A special case are the concrete optimizations of a stack machine given in [TvS82], that deal with jump instructions and therefore could not be considered in [DvHPR97]. The optimizations only consider instruction sequences $instrs(pc_1,db_1,k_1)$ and il_2, where only the *last* instruction is a jump. For this case, it is sufficient to generalize correctness condition (7.1) to

$$\begin{array}{ll} & \mathrm{I}(\mathrm{db}_1,\mathrm{st}_0,\mathrm{pc}_0) \\ & \wedge\ \exists\ \mathrm{i}.\ \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc}_0,\mathrm{st}_0)\ \textbf{times}\ \mathrm{i}\rangle\ (\mathrm{pc}_0 = \mathrm{pc}_1 \wedge \mathrm{st}_0 = \mathrm{st}_1) \\ & \wedge\ \mathrm{peephole}(\mathrm{st}_1,\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2) \wedge \mathrm{db}_2 = \mathrm{repl}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\ \mathrm{il}_2) \\ & \wedge\ \mathrm{pc}_1 = \mathrm{pc}_2 \wedge \mathrm{pc}_1 = \mathrm{pc} \wedge \mathrm{st}_1 = \mathrm{st}_2 \\ \rightarrow & \quad \mathrm{k}_1 \neq 0 \wedge \mathrm{linear}(\mathrm{instrs}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1 - 1)) \\ & \wedge\ \mathrm{il}_2 \neq [] \wedge \mathrm{linear}(\mathrm{butlast}(\mathrm{il}_2)) \\ & \wedge\ \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc}_1,\mathrm{st}_1)\ \textbf{times}\ \mathrm{k}_1\rangle \\ & \qquad \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}_2,\mathrm{st}_2)\ \textbf{times}\ \mathrm{k}_2\rangle \\ & \qquad\quad (\mathrm{st}_1 = \mathrm{st}_2 \wedge \mathrm{pc}_2 = \mathrm{shift}(\mathrm{pc}_1,\mathrm{pc},\mathrm{k}_2 - \mathrm{k}_1)) \end{array} \tag{7.6}$$

(*butlast* removes the last element of a list). The new condition is still sufficient to guarantee the commutativity of the k_1:k_2 diagram with unchanged coupling invariant. The only new requirement in the generalized condition is, that the two last instructions jump to the same address (modulo *shift*). That the jump address is not within the optimized code already follows from (7.5).

Finally let us give a simple example for peephole optimization, where not only the last instruction of the optimized sequence is a jump. The example should make it obvious, that we then have to verify several commuting diagrams, that result from decomposing the k_1:k_2 diagram into subdiagrams at every jump instruction.

For the example we assume that it is possible to select an integer value from the state *st* with *get(l,st)* (typically *l* is a location in memory and *get* is memory access). Three typical jump instructions would then be $BZE(l,n)$, $BNZ(l,n)$, and $BRA(n)$ (branch on zero, branch on not zero, branch unconditionally) with ASM rules defined by

if code(pc,db) = BZE(l,n)
then if get(l,st) = 0
  then pc := pc + n
  else pc := pc + 1

if code(pc,db) = BNZ(l,n)
then if get(l,st) = 0
  then pc := pc + 1
  else pc := pc + n

if code(pc,db) = BRA(n)
then pc := pc + n

An obvious peephole optimization then is to replace $il_1 = [BZE(l,2)\ BRA(n)]$ with $il_2 = [BNZ(l,n-1)]$ whenever $n > 0$. If $instr(pc_1,db_1,2) = il_1$ and neither the program start is at $pc_1 + 1$ nor jumps to this address exist, then this is a correct optimization. For the verification we need the same coupling invariant as in the previous section and the proof for the case $pc \neq pc_1$ is unchanged. For the verification of the optimized we now need two commuting diagrams: A 1:1 diagram for the case that $get(l,st) = 0$, and a 2:1 diagram for $get(l,st) \neq 0$. The formal proof, that both diagrams commutate, i.e. that

$$\begin{array}{ll} & \mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \wedge \mathrm{pc} = \mathrm{pc}_1 \wedge \mathrm{get}(\mathrm{l},\mathrm{st}) = 0 \\ \rightarrow & \langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st})\rangle\ \langle \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}',\mathrm{st}')\rangle \\ & \quad \mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \end{array}$$

and

$$\begin{array}{l} \quad \mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \wedge \mathrm{pc} = \mathrm{pc}_1 \wedge \mathrm{get}(\mathrm{l},\mathrm{st}) \neq 0 \\ \rightarrow \langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st}) \rangle \, \langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st}) \rangle \, \langle \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}',\mathrm{st}') \rangle \\ \quad \mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}'). \end{array}$$

hold, is easy.

Chapter 8

Summary of Part I

The first part of this work was considered with the development of tool support for the specification language of ASMs and the definition of generic proof obligation for the correctness of ASM refinements. Three main results were achieved:

First, we defined a natural embedding of the specification language of ASMs into Dynamic Logic, that allows to formalize properties of ASMs, especially the correctness of refinements. With this result, tool supported deduction for ASMs becomes possible.

Second, we developed a theory for the modularization of correctness proofs for ASM refinements. The verified modularization theorems generalize the results known from literature. Data refinement and Peephole optimization from compiler verification are special cases of the theorem.

Third, the results were integrated into the KIV system. The modularization theorems were verified in KIV and several extensions were made to the specification language and the deduction component, to get efficient tool support for ASMs.

Part II

The Prolog-WAM Case Study

Chapter 9

Introduction and Overview

The subject of the Prolog-WAM case study is the correctness proof for the compilation of Prolog programs into byte code of the Warren Abstract Machine (WAM). The WAM (and variants) today is the basis of all popular Prolog implementations.

Our work is based on a systematic presentation of the compilation as 12 ASM refinements in [BR95]. The starting point is a Prolog interpreter, specified as an ASM, that describes the operational semantics of the core of Prolog (clauses with *!, true* and *fail*) as the construction of a search tree. For pure Prolog the semantics is identical to the tree constructed by SLD resolution. The extension of the ASM to full Prolog (in [BR94]) has become an ISO standard for the definition of Prolog semantics.

The first Prolog interpreter, we will call ASM1 in the following, is then stepwise refined to an interpreter ASM13 of byte code of the WAM. In parallel to this transformation the Prolog program is compiled. On intermediate levels the code consists partially of not yet compiled Prolog clauses, partially already of WAM instructions. Each refinement introduces machine concepts like stacks, registers, pointer structures etc.. The refinements are constructed such that they are orthogonal: The compilation of clause selection, of single clauses and of literals are each treated in separate refinement steps. Besides the pure compilation steps there are also refinements that optimize the data representation. The byte code instructions used in the final ASM13 are very simple. They consist each of a number of register transfers that can easily be translated into the assembler code of any processor.

The main goals in the Prolog-WAM case study were:

- The formal specification of the compilation steps and compiler assumptions given in [BR95].
- The formalization of the correctness of refinements.
- To define a suitable proof methodology for the verification of refinement correctness.
- The development of suitable support in the KIV system, that allows the efficient demonstration of the correctness of the refinement steps.
- To formally prove the correctness arguments or to find errors and to remedy them.

Main parts of the theory in Chapters 4 and 6 were developed to achieve the first two goals. Development of suitable proof support required many improvements in KIV, that were summarized in section 3.3. The comparison with the case study in Isabelle in Sect. 20 shows, that the proof support can compete with other systems. Nevertheless the formal verification of an ASM refinement still requires a man month on average. In this work 8 of the 12 refinements have been verified.

A substantial result of the verification was a confirmation of the work done in [BR95]. Until now, no major changes were necessary for the ASMs. Also the ideas for the correctness proofs were correct for all refinements.

Nevertheless even the verification of the first refinement showed, that a formal verification of refinement correctness requires to make explicit a large number of properties, that were only implicitly assumed (compare the first approach at the beginning of Sect. 11.2 with the final coupling invariant at the end). Although many of these properties are easy to find, we found that there is a large gap between the mathematical argument for correctness and a fully formal proof.

Therefore it is not too surprising, that a large number of smaller problems were found in the ASMs as well as in the compiler assumptions, that did not show up in the informal analysis in [BR95]. The most important problems were:

- ASM3 and ASM4 contain a not intended indeterminism, that must removed by a stronger rule test (see 14.2)
- In the switching instructions the backtracking case was missing (see 15.2)
- The unify instruction of ASM9 used the renaming index of the first instead of the second environment (see 17.1)
- The compiler assumptions for several refinements were described correctly in the text, but the formalization had to be made more precise (see 14.2,15.2)
- ASM1 – ASM8 answer the query `?- p(q)` positively, given the two clauses `p(X) :- X.` and `q.`. But in the translation of clauses to code (i.e. in the refinement to ASM9) clause bodies may no longer contain variables or lists (see 17.2).

All problems were relatively easy to correct. Nevertheless the result demonstrates, that even a very careful informal analysis should be complemented by a formal correctness proof, if the goal is a correct compiler.

The following chapters discuss the correctness proofs in detail. They are organized as follows:

The next chapter describes the Prolog interpreter from [BR95]. The following chapters then consist of two sections: the first specifies the refinement of ASM of the previous chapter to a new ASM. This section largely follows [BR95]. Where already the formalization required corrections or deviations, they are explained in this section. The second section then describes the formal verification of the refinement, the experiences learned thereby, and the corrections of ASMs and compiler assumptions that resulted from the verification.

We always have tried, to explain the operations needed in each refinement and in the coupling invariants immediately. If any notations should remain unclear, they can be looked up in the full algebraic specification in KIV given in Appendix E.

In the following we will denote with i/j the refinement from ASMi to ASMj. In every section on the verification of refinement i/j we will also use the convention to name state variables of ASMi (to be precise: state variables that resulted from the translation of ASMi to DL) with $\underline{x}$ and the state variables of ASMj with $\underline{x}'$. We always assume the vectors to be disjoint. The rule (in the sense of section 2.2) of ASMi and ASMj will be named $RULE$ and $RULE'$ and always have the form

RULE(**var** x) **begin**
if ε_1 **then** RULE$_1$(x) **else**
if ε_2 **then** RULE$_2$(x) **else**
⋮
if ε_n **then** RULE$_n$(x) **end**

$RULE_1$, $RULE_2$, ..., $RULE_n$ are rules in the sense of 2.3 and we will use the term "rule" in the following only with this meaning.

Chapter 10

ASM1 : A Prolog Interpreter

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

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

If a unification fails, alternative clauses have to be chosen by backtracking. Due to this the interpreter has to keep a record of the former computation states and the corresponding clause choice alternatives. This history is usually represented as a search tree, that is constructed by the operations above. Each node represents a computation state, and the children of a node are the possible successor states, that can be reached by unification with the different clause heads.

In an ASM we represent a search by a set of *nodes*, connected from leaves to the root by a function *father*. The root node is denoted by $\perp$, *father* is undefined for this node. Information on alternative clauses, which may be tried at a node n, is stored as a list *cands(n)* of candidate nodes. Each node in this list refers via a function *cll* to a clause line in the Prolog program. Suitable initial lists of candidates are constructed with the help of a function *procdef* (for the specification of *procdef* see later on).

The current computation state of the interpreter is stored in a program variable (i.e. a 0-ary dynamic function), the *currnode*. The computation state of a node n could be represented as the result of two functions *glseq*$[n]$ (goal sequent) and *sub*$[n]$.

But to handle the *cut* instruction of Prolog, an extension of this state representation is required. A *cut* updates the *father* of the current node to the *father* of that computation state whose *act* caused the introduction of the considered *cut*. For this we have to 'remember' where a *cut* has been introduced. An uniform solution is to attach the *father* of the (old) *currnode* to each clause body being introduced to the literal sequence. This attachment divides the sequence of literals into subsequents, called goals, each decorated by one node, called the cutpoint of the goal. The resulting (decorated goal) sequence *decglseq* looks as follows

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

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

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

To introduce the rules of ASM1 we will now consider the evaluation of the query `?- p.` on the following Prolog program.

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

which is stored as the value of a constant *db* (database) in the initial algebra of the ASM. Line numbers are shown explicitly in the program for explanatory purposes. The query `?- p.` is stored as the *decglseq* of node A in the initial search tree depicted in Fig. 10.1.

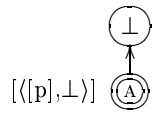


Figure 10.1

The two nodes labeled $\perp$ and A form the initial domain of a dynamic sort *node*, which is extended by the rules of the ASM. Tree structure as stored in the function *father : node* $\rightarrow$ *node* is indicated by the arrow in Fig. 6, so we have *father(A)* $= \perp$. Root node $\perp$ serves only as a marker when to finish search and does not carry information in ASM1. The initial *currnode* is A, as indicated by the double circle. Computed substitutions (attached to the nodes with the *sub* function) are not shown in the figures, since they are always empty in the example.

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

In *call* mode, which is the initial mode, the candidate nodes are computed (for a guard which involves *act*, checks for *decglseq* $\neq$ *[]* and *goal* $\neq$ *[]* are implicitly assumed, and we also omit the obligatory conjunct *stop* $=$ *run*).

call rule
  if is_user_defined(act) $\wedge$ mode = call
  then let[cll$_1$,. . .,cll$_n$] = procdef(act,db)
    extend node
    by tmp$_1$,. . .,tmp$_n$
    with father[tmp$_i$] := currnode
      cll[tmp$_i$] := cll$_i$
      cands := [tmp$_1$,. . .,tmp$_n$]
    endextend
    mode := select

The rule uses the abbreviation *cands* for *cands*[*currnode*], i.e. the candidate nodes of *currnode*. In the following we will also use the analogous abbreviations *father*, *decglseq* and *sub*.

The **extend** construct, by expanding the universe *node*, allocates one node for every clause whose head 'may unify' with the literal *act*. This list of clause lines is computed by *procdef(act,db)* and is assumed to contain at least those clauses, whose heads unify with the activator, and at most those with the same leading predicate symbol as *act*. The use of **extend** with an arbitrary number of allocated nodes is a slight extension of [Gur95]. In DL the extension is realized with a procedure, that traverses the list *procdef(act,db)*. The result of the application of *call* rule is depicted in Fig. 10.2.

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

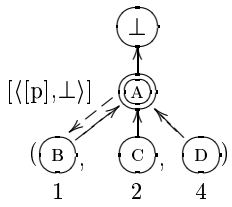


Figure 10.2

select rule
if is_user_defined(act) ∧ mode = select
then if cands = []
then backtrack
else let clau = rename(clause(cll[car(cands)],db),vireg)
let mgu = unify(act,head(clau))
if mgu = failure
then cands := cdr(cands)
else currnode := car(cands)
decglseq(car(cands)) := mgu $\hat{}_d$ [⟨body(clau), father⟩ | cont]
sub[car(cands)] := sub ∘ mgu
cands := cdr(cands)
vireg := vireg +1
mode := call

where

backtrack ≡
if father = ⊥
then stop := halt
subst := failure
else currnode := father
mode := select

This rule causes backtracking if there are no (more) alternatives to select. Otherwise, by repeated application, it removes all candidates whose heads do not unify with *act*. When the first candidate is reached, for which a most general unifier *mgu* exists (function *clause* selects the clause at a clause line[1], and variable index *vireg* is used to rename the implicitly universal quantified clause variables to new instances), this node becomes the new *currnode*. A new *decglseq* is computed by replacing the activator of the old *decglseq* with the body of the selected clause. As a cutpoint the *father* of the old *currnode* is attached to this new goal. The *mgu* is applied to the resulting *decglseq* (with the infix operation $\hat{}_d$) and composed (with ∘) with the old substitution *sub*.

The result of applying the *select* rule in our example is shown in Fig. 10.3. The value of the *mode* variable is now *call* again. Since the activator *fail* is not user defined, *fail* rule is applied.

fail rule
if act = fail **then backtrack**

It sets *currnode* to A again. Note that node B is not formally deallocated. It remains in the carrier set of *node*. Again in *select* mode, the next candidate node for A, node C, is selected. Its *decglseq* is computed as $[\langle [q, !, true], \bot\rangle\ , \langle [], \bot\rangle\ ]$. Subsequently *call* rule allocates one new candidate node E for the only appropriate clause q. After selection of node E ASM1 reaches the state shown in Fig. 10.4.

[1] since *clause* clearly depends on the Prolog program, we have added an argument *db* compared to [BR95]

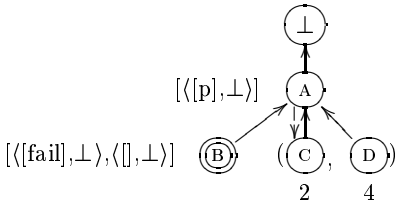


Figure 10.3

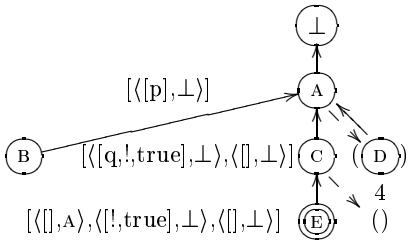


Figure 10.4

The now empty *goal* is removed by the *goal success* rule.

goal success rule
if decglseq $\neq$ [] $\wedge$ goal = []
then decglseq := cdr(decglseq)

Then the activator is a cut, which is removed from *decglseq* by *cut* rule.

cut rule
if act = !
then father := ctpt
decglseq := cont

The rule sets the *father* of the current node E to the cutpoint *ctpt* of the current *decglseq*, which here is the root node $\perp$ (see Fig. 10.5). This "cuts" the alternative D at node A. The *cut* rule is the only one that uses *ctpt*. For the activator *true* ASM1 then executes the following rule.

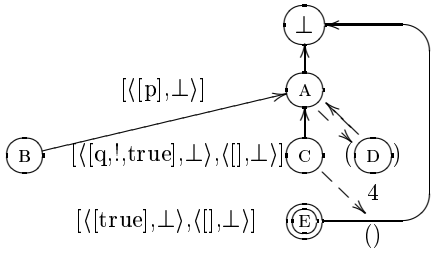


Figure 10.5

true rule
if act = true **then** decglseq := cont

Finally, with another two applications of *goal success* rule, *decglseq(E)* becomes empty. This means that the initial query is completely solved. Therefore *query success* rule sets the answer

substitution *subst* to *sub(currnode)* (here, of course, the empty substitution), and finishes the execution by setting *stop* to *halt*.

query success rule
if decglseq(currnode) = []
then stop := halt
subst := sub

If we consider a variant of our example program, where we replace clause `p :- q,!,true` with `p :- q,!,r`, we would also arrive at the situation of Fig. 10.5. But now *call* rule would allocate a node F with an empty list of candidates, since no clauses for predicate `r` are given. *select* mode, finding no more alternatives, would backtrack from nodes F and E. Since the father of E is the root node $\perp$, execution would finally stop with *stop* = *halt* and *subst* = *failure*.

Chapter 11

1/2: From Search Trees to Stacks

11.1 Definition of ASM2

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

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

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

Third, instead of providing a list of candidate nodes, ASM2 attaches the *first* candidate directly via the *cll*-function. This is possible if we assume that clauses whose head starts with the same predicate are stored in successive clause lines followed by a special *null* marker. The "compiled" representation db_2 of our example Prolog program for ASM2 thus has to look like

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

A new $procdef_2$ function is needed, such that $procdef_2(act,db_2)$ now yields the first clause line whose head may unify with the activator *act*.

For $act = p$ we get $procdef_2(p,db_2) = 1$, 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 $compile_{12}$, which is used as an axiom in the equivalence proof for 1/2.

$$\begin{aligned} & \mathrm{db}_2 = \mathrm{compile}_{12}(\mathrm{db}) \\ \rightarrow\ & \langle \mathrm{CLLS\#}(\mathrm{procdef}_2(\mathrm{act},\mathrm{db}_2),\mathrm{db}_2),\mathrm{db}_2;\mathrm{col})\rangle \\ & \quad \mathrm{mapclause}(\mathrm{procdef}(\mathrm{act},\mathrm{db}),\mathrm{db}) = \mathrm{mapclause}'(\mathrm{col},\mathrm{db}_2) \end{aligned}$$

Procedure *CLLS#*[1] collects consecutive line numbers, until a *null* is reached, and functions *mapclause* and *mapclause'* select the clauses at each line number. Note that in contrast to [BR95] (p. 17) we have *not* assumed that the literals were sorted in the original database, and that the equality *procdef(act,db) = col* of clause lines holds. Instead we only require the equality of the clauses. This weakening of the compiler assumption is necessary, otherwise it can not be fulfilled by any implementation of the *procdef* function that selects clauses more precisely than looking only at the leading predicate symbol. Note, that with the stronger assumption the three calls $procdef_2$(`p(f(X))`,db_2), $procdef_2$(`p(g(X))`,db_2) and $procdef_2$(`p(X)`,db_2) can not return three different results, since the three clause lists, which can be collected at these addresses end with

[1] A procedure, not a function is used, to make sure that the specification does not become inconsistent with a db_2 that does not contain a *null* marker. See the same argument for *STACK#* in the following section, p. 71

the same *null* marker, that marks the end of the clauses for `p` (so all three lists must be end pieces, not arbitrary sublists of the clause list for `p`). Our weaker assumption can be implemented for any definition of *procdef* by duplicating code. The duplicated code can be removed later on, when the abstract code selection with *procdef* is replaced with switching instructions (see Sect. 15.1).

Instead of allocating a candidate list, ASM2 simply assigns *procdef'(act,db)* to *cllreg*. Removing a candidate from *cands* now corresponds to incrementing *cllreg*. If the clause at *cllreg* becomes *null*, no more candidates are available.

Since ASM2 no longer needs to allocate a current node *currnode*, a new node must be created in *select* mode, to save the current register contents to a node. The new *call* and *select* rule therefore are

call rule
if is_user_defined(act) $\wedge$ mode = call
then cllreg := procdef$_2$(act,db$_2$)
mode := select

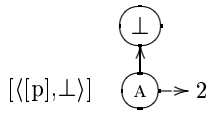
select rule
if is_user_defined(act) $\wedge$ mode = select
then if clause(cllreg,db$_2$) = null
then backtrack
else let cla = rename(clause(cllreg,db$_2$),vireg)
let mgu = unify(act, head(cla))
if mgu = failure
then cllreg := cllreg +1
else let tmp = new(s)
s := s $\cup$ {tmp}
breg := tmp
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
cll[tmp] := cllreg +1
decglseqreg := mgu $\hat{}_d$ [$\langle$body(cla),breg$\rangle$ | cont]
subreg := subreg $\circ$ mgu
vireg := vireg +1
mode := call

where

backtrack $\equiv$
if breg = $\perp$
then stop := halt
subst := failure
else decglseqreg := decglseq[breg]
subreg := sub[breg]
breg := b[breg]
cllreg := cll[breg]
mode := select

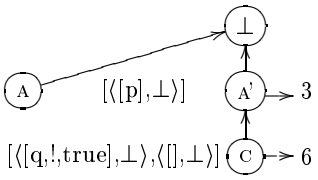
All other rule of ASM1 are unchanged, except that *father* is renamed to *b* and abbreviations *decglseq*, *father* and *sub* (for *decglseq*[*currnode*] etc.) have to be replaced with the registers *decglseqreg*, *breg* and *subreg*.

In our example program ASM2 now runs through the states shown in Fig. 11.1 and Fig. 11.2. The corresponding states in ASM1 were those in Fig. 10.3 and Fig. 10.4.



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

Figure 11.1



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

Figure 11.2

Dashed arrows now point to the *cll* of a node. Since the values attached to the *currnode* are now stored in registers, allocation of nodes corresponding to B and D is avoided. On the other hand, when node A is visited by backtracking (by executing *fail* rule in the state shown in Fig. 10.1), its computation state is moved to registers, and the following *select* rule allocates a new, similar choicepoint *A'*. Removing this redundancy is the subject of the next refinement.

In ASM2, the nodes which may be visited in the future are always reachable from *breg* via the b function. They form a stack, but note that there may still be abandoned nodes in the *node* universe, which are no longer reachable (here A). This causes one of the problems in the verification of the refinement from ASM1 to ASM2. The tuple of values *decglseq(n)*, *sub(n)*, *cll(n)* and *b(n)* attached to a stack node n is usually called a *choicepoint*.

11.2 Equivalence Proof 1/2

In this section we will describe the formal verification of the first refinement with KIV. The main focus of this section is not the application of the general theory for the verification of ASMs we developed in the first part (we have data refinement with 1:1 diagrams here), but on the practical problems that arise in a formal, system-supported correctness proof, which consists mainly in the incremental development of a suitable coupling invariant. We will show exemplarily for this refinement, that

- the informal correspondence between the states of the ASMs given in [BR95] is by far not sufficient for a formal proof.
- a lot of additional properties must be formulated, that are not foreseeable at the beginning of the verification, but which are necessary to guarantee the correctness of the refinement.
- the efficient verification of ASM refinements requires a system with very good support for an *incremental* verification of goals.

To assure the last point, a lot of details had to be improved in the KIV system. Some of them were described in Sect. 3.4.

The following description of the verification unfortunately requires to confront the reader with a lot of details. Only the consideration of these details leads to the detection of hidden assumptions, which ultimately *guarantee* the correctness of the refinement. The reader who is not interested in the details may just have a look at the 9 initial properties as given in [BR95] at the beginning of the following subsection, and compare them to final coupling invariant shown at the end. This should give an impression about the work needed to translate an informal mathematical argument to a complete, formal proof.

The Initial Coupling Invariant The refinement from ASM1 to ASM2 does not change the control structure of the interpreter. One rule application of ASM1 corresponds to one rule application of ASM2, i.e. we have the case of data refinement. For the proof obligations from Chapter 6 this means, that we can choose $ndtype(\underline{x}, \underline{x}')$ to be constantly mn, and that by choosing $i = j = 1$ in the proof obligation (6.5) we can simplify it to

$$\begin{array}{l} \quad \text{INV}(\underline{\text{x}},\underline{\text{x}}'),\ \text{stop} = \text{run},\ \text{stop}' = \text{run} \\ \vdash \langle \textbf{if}\ \text{stop} = \text{run}\ \textbf{then}\ \text{RULE}\rangle \\ \quad \langle \textbf{if}\ \text{stop}' = \text{run}\ \textbf{then}\ \text{RULE}'\rangle\ \text{INV}(\underline{\text{x}},\underline{\text{x}}') \end{array} \tag{11.1}$$

The proof now splits into 5 cases for each of the 5 rules of the two ASMs. The other proof obligations (6.10), (6.8), (6.9) and (6.11) are all trivial, since *INV* will contain the formula *stop* = *stop'*. So the "only" critical point for a successful formal proof is to find a coupling invariant INV(x,x'), such that formula (11.1) is provable for each corresponding pair of rules.

Some rough indication how such a formula *INV* might look like is already given in [BR95], p.17f. There an auxiliary function F is proposed, which maps the nodes in the stack of ASM2 to corresponding nodes in the search tree of ASM1 (see Fig. 11.3).

[Sch94] pointed out that *F* cannot be given statically, but has to be defined by induction on the number of rule applications. This requires a formalism, where a dynamic function can be updated by proof steps.

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

$\exists$ F:

1 decglseq[currnode] = decglseqreg

2 sub[currnode] = subreg

3a mapclause(map(cll, cands[currnode]),db)
= mapclause'(clls(cllreg,db_2),db_2)

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]]),db)
= mapclause'(clls(cll'[n],db_2),db_2)

7b every(father, cands[F[n]], F[n])

8 father[F[n]] = b[n]

9 F[⊥] = ⊥

In the formulas *every*(*father*,*cands*[n], n) means, that n is the father node of every node in *cands*[n].

The equations 1 and 5 actually do not hold. Although the goals are identical, cutpoints have to be mapped by F. Therefore already [Sch94] defines a function F_d with the axioms

$$F_d(F,[]) = []$$

$$F_d(F,[\langle goal,ctpt\rangle \mid dgl]) = [\langle goal, F(ctpt)\rangle \mid F_d(F, dgl)]$$

and replaces 1 and 5 by

1 decglseq[currnode] = F_d(F, decglseqreg)

5 decglseq[F[n]] = F_d(F, decglseq′[n])

He also adds the obvious equations

10 stop = stop′ ∧ mode = mode′ ∧ vireg = vireg′

Formulas 1 – 10 formed our first version of the coupling invariant, with which we started the formal verification with the KIV system.

Development of the Correct Coupling Invariant We found that the first version of the coupling invariant was not sufficient to prove the correctness. Instead a dozen iterations were necessary to find the correct one. The failed proof attempts took much more time than the successful verification with the correct invariant. We give a rough overview over the search and explain, how hidden assumptions were detected during proof attempts. Adding these assumptions to the coupling invariant and attempting a new proof revealed further gaps, which required new modifications in the coupling invariant. An evolutionary proof process resulted.

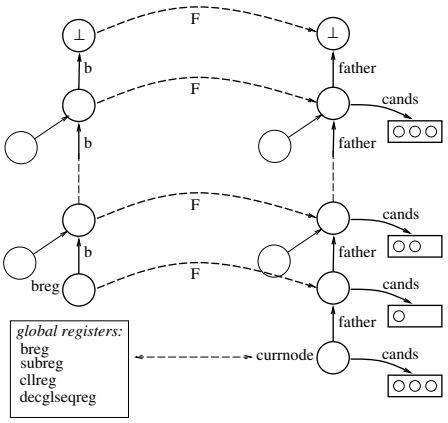


Figure 11.3

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

$$F[breg] = F[\bot] \rightarrow breg = \bot \tag{11.2}$$

This formula holds (compare Fig. 11.3), but how to deduce it? A short look at the visualized proof tree shows that this proof situation arose by trying to guarantee that in the backtracking case ASM2 stops (with failure) if *and only if* ASM1 stops! The "if" direction is trivial but for the "only if" direction we must prove (11.2).

What we need is the *injectivity* of F, as can also be seen in Fig. 11.3. We therefore add

11 F injon s

to *INV*, where *injon* is defined as

$$\text{F injon s} \equiv \forall\ \text{n,n}_1.\ \text{n} \in \text{s} \wedge \text{n}_1 \in \text{s} \wedge \text{F[n]} = \text{F[n}_1\text{]} \rightarrow \text{n} = \text{n}_1$$

Thereby we make it available in all proof situations. On the other hand it is now necessary to prove that injectivity is invariant in all rules.

Characterization of the Stack Unfortunately, it is too strong, to assume the injectivity of F. A proof attempt now fails, with a goal that requires to prove injectivity of $F[new(s') \leftarrow currnode]$. We are not able to show, that *select* rule keeps the injectivity of F invariant. (after *select* rule the new node $new(s')$ must be mapped to *currnode*). A detailed analysis shows, that there are indeed situations, where this is impossible. Figure 11.4 shows such a situation, in which two different nodes of ASM2 are mapped to the same node of ASM1.

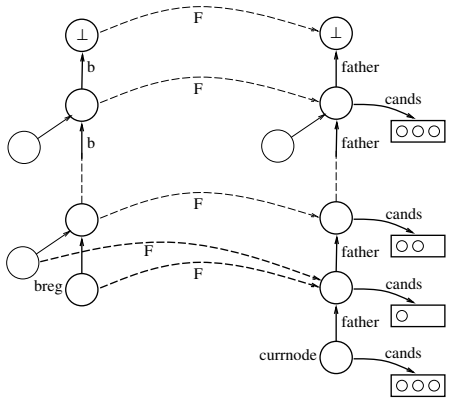


Figure 11.4

The problem arises because there are abandoned nodes that are no longer in the stack (i.e. reachable following the function b from *breg*) but still present in the set of allocated nodes. The function F is still defined on such nodes, violating injectivity. But on the smaller set of stack nodes injectivity holds. What we need is a logical characterization of the stack nodes. Then we can restrict injectivity of F to the stack.

A characterization of the stack is also necessary to restrict other still missing properties of F to stack nodes. One other such property can be derived from another unprovable goal in the same proof.

$$\text{cands[currnode} \leftarrow \text{x][F[n]]} = \text{cands[F[n]]}$$

Here it must be proved, that a modification of the candidates *cands[currnode]* does not modify the candidates of any node in the codomain of F. To prove this we need:

12 F[n] $\neq$ currnode

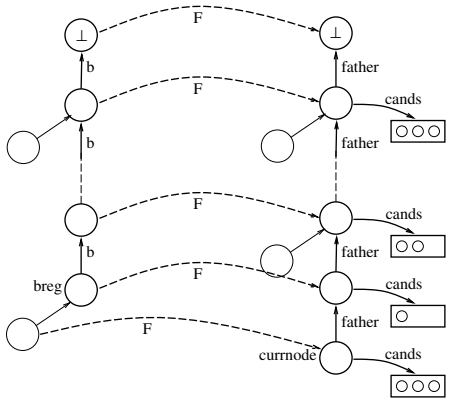


Figure 11.5

But this formula is also not true for abandoned nodes, as can be seen in Fig. 11.5, that shows a pair of states after backtracking. Only that *currnode* is not in the image of *stack* nodes is true.

An important problem with the formal definition of stack nodes is, that the simple approach that defines a function *stackof* with

$$\text{stackof(b,}\bot\text{)} = [],$$

$$\text{breg} \neq \bot \rightarrow \text{stackof(b,breg)} = [\text{breg} \mid \text{stackof(b,b[breg])}]$$

is incorrect. It leads to an inconsistent specification, since it is possible to construct dynamic functions, that cyclically connect nodes (for an arbitrary function b and a node $n \neq \bot$ define $b' := b[n \leftarrow n]$. Then using the axioms above, it is easy to prove $stackof(b', n) = [n|stackof(b', n)]$, contradicting the list axiom $x \neq [a|x]$).

A correct approach to characterize the list of stack nodes is, to use the program *STACK#* below. Its termination guarantees, that the stack does not contain cycles.

```
STACK#(n, b; var stack)
begin
if n = ⊥ then stack := [] else
  begin STACK#(b[n], b; stack); stack := [n | stack] end
end
```

Figure 11.6 : Characterization of cycle free Stacks

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

$$\exists \text{ F: } \varphi \wedge \langle \text{STACK\#(breg, b; stack)} \rangle \ (\forall \text{ n. } \text{n} \in \text{stack} \rightarrow \psi(\text{n})) \tag{11.3}$$

It says now, that (for suitable F) φ holds and that B-LIST# terminates with a list *stack* as result, such that ψ holds for all its elements.

Cutpoints Proving equivalence between the two *cut* rules with this version of *INV* shows another difficulty: ψ must be guaranteed for the new stack shortened by execution of the cut. This stack starts with a new *breg*, which was set to the first cutpoint of *decglseqreg*. Now, of course, the

new stack would inherit ψ from the old one, if we knew that it is a part of the old one. But this can not be deduced from the current *INV*. We have to assert that the cutpoints in the current decorated goal sequence are elements of the current stack. We therefore define a new predicate *cutptsin* (written infix) with axioms

$$\begin{array}{l} [] \text{ cutptsin stack,} \\ [\langle \text{goal,ctpt} \rangle \mid \text{dgl}] \text{ cutptsin stack} \leftrightarrow \text{ctpt} \in \text{stack} \wedge \text{dgl cutptsin stack} \end{array} \tag{11.4}$$

and add:

decglseqreg cutptsin stack

to the coupling invariant. In this version, the definition of *cutptsin* simply checks whether all cutpoints of the first argument are elements of the second. Because the decorated goal sequence *decglseq*[*n*] of every node in the stack can potentially become the *decglseqreg* (by backtracking), we also have to add

decglseq′[n] cutptsin (stack from b[n])

where function *from* (again written infix) is axiomatized with

$$\begin{array}{l} [] \text{ from n} = [], \\ \text{n} \neq \text{n}' \rightarrow [\text{n}|\text{l}] \text{ from n}' = \text{l from n}', \\ [\text{n}|\text{l}] \text{ from n} = [\text{n}|\text{l}] \end{array}$$

With the new formulas *INV* is now

$$\begin{array}{l} \exists \text{ F.} \quad \varphi \\ \quad \wedge \langle \text{STACK\#(breg, b; stack)} \rangle \\ \qquad (\quad \text{decglseqreg cutptsin stack} \\ \qquad \wedge (\forall \text{ n.} \quad \text{n} \in \text{stack} \\ \qquad\qquad \rightarrow \quad \psi(\text{n}) \\ \qquad\qquad\quad \wedge \text{ decglseq'[n] cutptsin (stack from b[n])} \end{array}$$

Still, this invariant is not strong enough. The proof fails because when the *cut* rule is applied, we have not made sure, that the cutpoints in *decglseqreg* other than the first remain in the stack that has been *shortened* by the cut. This is true only because the cutpoints point into the stack *in the right ordering* (see Fig. 11.7). Therefore the axioms (11.4) for *cutptsin* must be strengthened to

$$\begin{array}{l} [] \text{ cutptsin stack,} \\ \quad [\langle \text{goal,ctpt} \rangle \mid \text{dgl}] \text{ cutptsin stack} \\ \leftrightarrow \quad \text{ctpt} \in \text{stack} \\ \quad \wedge \text{ dgl cutptsin (stack } \textbf{from ctpt}) \end{array}$$

INV is syntactically unchanged. Fortunately all proofs up to this point used only lemmas for *cutptsin* that remain valid for the new axiomatization. Therefore, no proof needs to be redone (and this fact is checked by the "correctness management" of KIV).

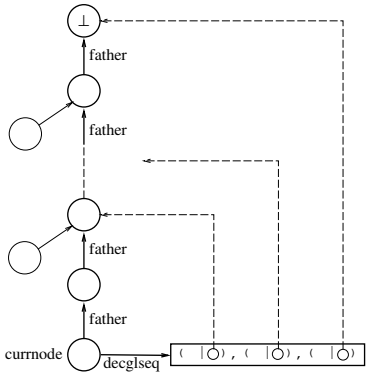


Figure 11.7

More Properties The coupling invariant is still not complete. Several further proof attempts revealed that it is necessary to make properties about the structure of the search tree of ASM1 explicit. Some of these properties are (informally): no candidate is in the range of *F*, no candidate list has duplicates, the intersection of different candidate lists is empty, and so on. Altogether 12 proof attempts were made with different coupling invariants (not counting different versions due to typing errors) until the final coupling invariant shown below was reached. *All* of the properties listed were actually needed to complete the proof.

$\mathrm{INV}_{12} \equiv$

$$
\begin{array}{l}
\exists\ \mathrm{F}.\quad \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{mode} = \mathrm{mode}' \wedge \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{sub}[\mathrm{currnode}] \\
\quad \wedge\ \mathrm{F}[\bot] = \bot \wedge \mathrm{F}[\mathrm{breg}] = \mathrm{father}[\mathrm{currnode}] \wedge \bot \neq \mathrm{currnode} \\
\quad \wedge\ \mathrm{F_d}(\mathrm{F},\ \mathrm{decglseqreg}) = \mathrm{decglseq}[\mathrm{currnode}] \\
\quad \wedge\ \bot \in \mathrm{s}' \wedge \bot \in \mathrm{s} \wedge \mathrm{currnode} \in \mathrm{s} \\
\quad \wedge\ (\quad \mathrm{mode} = \mathrm{select} \\
\qquad \rightarrow\quad \langle \mathrm{CLLS\#}(\mathrm{cllreg},\ \mathrm{db_2};\mathrm{col})\rangle \\
\qquad\qquad \mathrm{mapclause}'(\mathrm{col},\ \mathrm{db_2}) = \mathrm{mapclause}(\mathrm{map}(\mathrm{cll},\ \mathrm{cands}[\mathrm{currnode}]),\ \mathrm{db}) \\
\qquad\quad \wedge\ \mathrm{every}(\mathrm{father},\ \mathrm{cands}[\mathrm{currnode}],\ \mathrm{currnode}) \\
\qquad\quad \wedge\ \neg\ \mathrm{currnode} \in \mathrm{cands}[\mathrm{currnode}] \wedge \neg\ \bot \in \mathrm{cands}[\mathrm{currnode}] \\
\qquad\quad \wedge\ \mathrm{cands}[\mathrm{currnode}] \subseteq \mathrm{s} \wedge \mathrm{nodups}(\mathrm{cands}[\mathrm{currnode}])) \\
\quad \wedge\ \langle \mathrm{STACK\#}(\mathrm{breg},\ \mathrm{b};\ \mathrm{stack})\rangle \\
\qquad\quad (\quad \mathrm{decglseqreg\ cutptsin\ stack} \wedge \mathrm{candsdisjoint}(\mathrm{F},\ \mathrm{cands},\ \mathrm{stack}) \\
\qquad\quad \wedge\ \mathrm{F\ injon\ stack} \\
\qquad\quad \wedge\ \mathrm{nocands}(\mathrm{F},\ \mathrm{cands},\ \mathrm{stack}) \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\qquad\quad \wedge\ \forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad \rightarrow\quad \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}[\mathrm{F}[\mathrm{n}]] \wedge \mathrm{F}[\mathrm{b}[\mathrm{n}]] = \mathrm{father}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\quad \wedge\ \mathrm{F_d}(\mathrm{F},\ \mathrm{decglseq}'[\mathrm{n}]) = \mathrm{decglseq}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\quad \wedge\ \langle \mathrm{CLLS\#}(\mathrm{cll}'[\mathrm{n}],\mathrm{db_2};\mathrm{col})\rangle \\
\qquad\qquad\qquad\quad \mathrm{mapclause}'(\mathrm{col},\mathrm{db_2}) = \mathrm{mapcl}(\mathrm{map}(\mathrm{cll},\ \mathrm{cands}[\mathrm{F}[\mathrm{n}]]),\ \mathrm{db}) \\
\qquad\qquad\quad \wedge\ \mathrm{every}(\mathrm{father},\ \mathrm{cands}[\mathrm{F}[\mathrm{n}]],\ \mathrm{F}[\mathrm{n}]) \\
\qquad\qquad\quad \wedge\ \mathrm{F}[\mathrm{n}] \neq \mathrm{currnode} \wedge \mathrm{F}[\mathrm{n}] \in \mathrm{s} \wedge \mathrm{nodups}(\mathrm{cands}[\mathrm{F}[\mathrm{n}]]) \\
\qquad\qquad\quad \wedge\ \mathrm{cands}[\mathrm{F}[\mathrm{n}]] \subseteq \mathrm{s} \wedge \neg\ \mathrm{currnode} \in \mathrm{cands}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\quad \wedge\ (\quad \mathrm{mode} = \mathrm{select} \\
\qquad\qquad\qquad \rightarrow\quad \neg\ \mathrm{F}[\mathrm{n}] \in \mathrm{cands}[\mathrm{currnode}] \\
\qquad\qquad\qquad\quad \wedge\ \mathrm{disjoint}(\mathrm{cands}[\mathrm{F}[\mathrm{n}]],\ \mathrm{cands}[\mathrm{currnode}])) \\
\qquad\qquad\quad \wedge\ \mathrm{decglseq}'[\mathrm{n}]\ \mathrm{cutptsin}\ (\mathrm{stack\ from\ b}[\mathrm{n}]))
\end{array}
$$

Chapter 12

2/3: Reuse of Choicepoints

12.1 Definition of ASM3

Although ASM2 allocates fewer nodes that ASM1, there are still two more possibilities to reduce their number, that are exploited in the optimizations to ASM3 and ASM4.

In this section we first describe the reuse of choicepoints. We follow [BR95], Chapter 1.3. The optimization can be explained most easily by looking at the example of the previous section: When the first alternative for activator `p` is tried, ASM2 allocates a new node A, and sets the values *decglseq*[A], *sub*[A] and *cll*[A] of the new choicepoint.

Since the first alternative does not lead to a solution, the interpreter executes a *backtrack* instruction, which removes the node A from the stack. Thereby the whole choicepoint becomes inaccessible. The subsequent select rule for the second alternative then pushes a new choicepoint *A'* on the stack. This choicepoint gets the same values as the one for the first alternative, except that *cll(A')* has been incremented (see Fig. 11.2, p. 67 in Sect. 11.2).

The optimization done in ASM3 avoids deallocation and reallocation of choicepoints. Instead it *reuses* the existing choicepoint. The optimization is achieved by replacing the removal of a choicepoint in the else-branch of backtracking with the assignment *mode := retry*, which activates a new rule, *retry* rule. This rule combines the effects of the else-branch of *backtrack* and of *select*. It is executed instead of *select* rule for every alternative except the first. It removes a choicepoint (i.e. to set *breg* to *b(breg)*) only on execution of the last alternative. Otherwise it reuses the old choicepoint by incrementing *cll(breg)*. The old *select* rule, which allocates a new choicepoint is now only called for the first alternative clause, and is renamed to *try* rule. The test whether any alternative exists, can now be done already in the call rule instead of the *try* rule. To avoid code duplication the common parts of *try* and *retry* rule (unification with the activator, incrementing *vireg* etc.) are moved to a new *enter* rule, which is activated with *mode* := *enter*. Altogether these transformations result in the following set of rules:

call rule
    if mode = call ∧ is_user_defined(act)
    then if clause(procdef$_2$(act,db$_2$)) = null
        then backtrack
        else cllreg := procdef$_2$(act,db$_2$)
            ctreg := breg
            mode := try

enter rule
    if mode = enter
    then let cla = rename(clause(cllreg,db$_2$),vireg)
        let mgu = unify(act, hd(cla))
        if mgu = nil

```
        then backtrack
        else decglseqreg := mgu ^d [<bdy(cla),ctreg> | cont]
             subreg := subreg ∘ mgu
             vireg := vireg +1
             mode := call

goal success rule
    if goal = [] ∧ decglseqreg ≠ []
    then decglseqreg := cdr(decglseqreg)

query success rule
    if decglseqreg = [] then stop := halt
    subst := subreg

try rule
    if mode = try
    then mode := enter
        let tmp = new(s)
        s := s ∪ {tmp}
        breg := tmp
        b[tmp] := breg
        decglseq[tmp] := decglseqreg
        sub[tmp] := subreg
        cll[tmp] := cllreg +1

retry rule
    if mode = retry
    then if clause(cll[breg],db2) = null
        then deep-backtrack
        else decglseqreg := decglseq[breg]
             subreg := sub[breg]
             cllreg := cll[breg]
             ctreg := b[breg]
             mode := enter

cut rule
    if act = ! then father := cutpt
    decglseqreg := cont

fail rule
    if act = fail then backtrack

where

backtrack ≡
    if breg = ⊥
    then stop := halt
        subst := failure
    else mode := retry
```

It should be noted, that *enter* rule uses a new register *ctreg* to set the cutpoint *ctpt* of the new *decglseqreg*. This is necessary, since after a *retry* rule we must now use *b*[*breg*] instead of *breg* as the value of *ctpt*. *call* rule and *retry* rule set *ctreg* appropriately.

12.2 Equivalence Proof 2/3

The description of the optimization from ASM2 to ASM3 suggests not to look at single rules in the verification, but to look for corresponding states, and to define groups of rules which keep this correspondence invariant. Two obviously corresponding states are the ones, when both ASMs are in *call* mode. In these states the values of the registers and the state of the choicepoint stack are the same (modulo renaming of stack nodes). Only little more complicated is the correspondence, when ASM3 executes a *retry* and ASM2 executes the corresponding *select*. In this case the register contents of ASM2 agree with the content of the topmost choicepoint of ASM3, and the remainder of ASM3 stack is identical to the ASM2 stack. If one writes *regs*, *stack* resp. *regs'*, *stack'* for the registers and the stack of ASM2 resp. ASM3, a first attempt for the coupling invariant is

$$\text{INV23(regs,stack,regs}',\text{stack}') \equiv \text{CINV} \vee \text{RINV}$$

where

$$\text{CINV} \equiv \text{mode} = \text{call} \wedge \text{mode}' = \text{call} \wedge \text{regs} = \text{regs}' \wedge \text{stack} = \text{stack}',$$

$$\text{RINV} \equiv \text{mode} = \text{select} \wedge \text{mode}' = \text{retry} \wedge \text{stack}' = \text{push(regs,stack)}$$

An analysis, which rule sequences lead from corresponding states to corresponding states results in the commuting diagrams shown in Fig. 12.1.

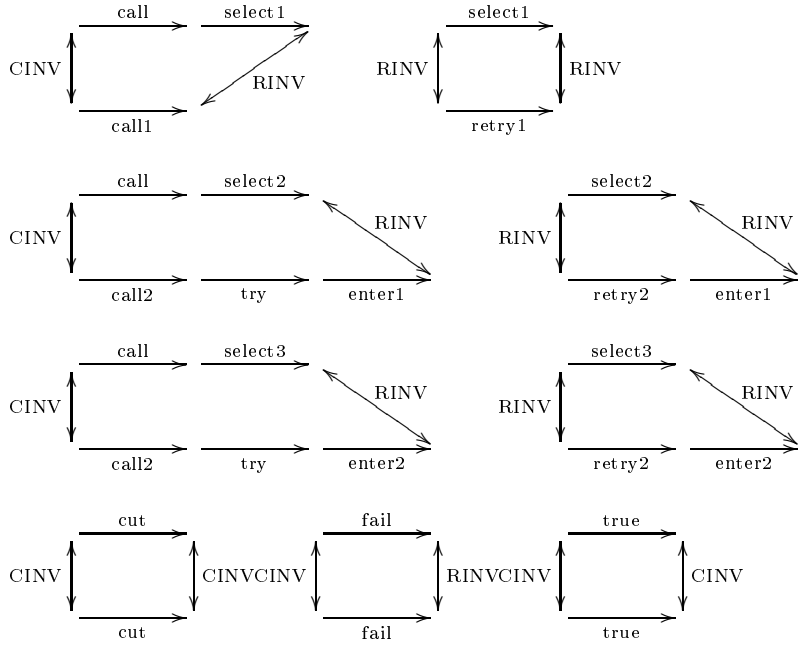


Figure 12.1 : Commuting Diagrams for the Refinement 2/3

select1, *select2* and *select3* are the three subcases of the *select* rule, *retry1* etc. are defined similarly. The theory developed in Chapter 6 now shows, that the proof of commutativity for all given diagrams is sufficient, to prove the equivalence of ASM2 and ASM3 (after a case distinction over all possible pairs of rules, just instantiate the quantified variables i and j in proof obligation (6.5) according to the size of each diagram). The commuting diagrams as well as the first approach for a coupling invariant agree with the ones given [BR95].

Since ASM3 allocates fewer nodes that ASM2, it is obvious that for the formal verification to go through, we again need a mapping F between the nodes. This again causes some of the problems that already showed up in the first refinement, namely injectivity of F on the current stack, and the *cutptsin* property.

A new property that was not needed in the verification of 1/2 is, that each *decglseq*[*n*] is not empty, and its first goal starts with a user defined literal (we again write *goal*[*n*] and *act*[*n*] for these components). This property is necessary to make sure that the rule that is applied after backtracking can only be *retry*, and not *goal success*.

Using the theory from Chapter 6 simplifies verification enormously, since it is completely unnecessary to define a coupling invariant for intermediate states of the diagrams (see also the comparison to Isabelle in Sect. 20).

A first attempt, to prove that all diagrams commute, was successful within 2 weeks, This first attempt used a preliminary version of the theory, which allowed the use of arbitrary commuting diagrams. It still required a separate correctness and completeness proof with two different coupling invariants, as well as a proof of the generic modularization theorem for the concrete instance (as we have now seen). 8 attempts were necessary, to find the two coupling invariants.

A second attempt with the full theory was successful to prove the equivalence of ASM2 and ASM3 in a few hours. Of course the time for the successful second attempt was shortened by the fact that a successful proof already existed. Somewhat more realistic is the comparison of interactions in both proofs: instead of 234 interactions only 75 were necessary to prove the commutation of all diagrams following coupling invariant.

$$
\begin{array}{l}
\mathrm{INV}_{23} \equiv \\
\quad \mathrm{stop} = \mathrm{stop}' \\
\wedge\ (\mathrm{stop} = \mathrm{success} \rightarrow \mathrm{subreg} = \mathrm{subreg}') \\
\wedge\ \bot \in \mathrm{s} \wedge \bot \in \mathrm{s}' \\
\wedge\ (\qquad \mathrm{stop} \neq \mathrm{run} \\
\qquad\quad (*\ \mathrm{CINV}\ *) \\
\quad \vee \quad \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{mode} = \mathrm{call} \wedge \mathrm{mode}' = \mathrm{call} \\
\qquad \wedge\ \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{subreg}' \\
\qquad \wedge\ (\exists\ \mathrm{F}. \quad \mathrm{F}[\bot] = \bot \wedge \mathrm{breg} = \mathrm{F}[\mathrm{breg}'] \\
\qquad\qquad\quad \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseqreg}') = \mathrm{decglseqreg} \\
\qquad\qquad\quad \wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}) \rangle \\
\qquad\qquad\qquad (\quad \langle \mathrm{STACK\#}(\mathrm{breg}, \mathrm{b}; \mathrm{stack}_0) \rangle\ \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) = \mathrm{stack}_0 \\
\qquad\qquad\qquad\ \wedge\ \mathrm{F\ injon\ stack} \wedge \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) \subseteq \mathrm{s} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\qquad\qquad\qquad\ \wedge\ \mathrm{decglseqreg}'\ \mathrm{cutptsin\ stack} \\
\qquad\qquad\qquad\ \wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad\qquad \rightarrow \quad \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}[\mathrm{F}[\mathrm{n}]] \wedge \mathrm{cll}'[\mathrm{n}] = \mathrm{cll}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseq}'[\mathrm{n}]) = \mathrm{decglseq}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{decglseq}'[\mathrm{n}]\ \mathrm{cutptsin\ cdr(stack\ from\ n)} \\
\qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}'[\mathrm{n}] \neq [] \\
\qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{n}])))) \\
\qquad\quad (*\ \mathrm{RINV}\ *) \\
\quad \vee \quad \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{mode} = \mathrm{select} \wedge \mathrm{mode}' = \mathrm{retry} \\
\qquad \wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \wedge \mathrm{decglseqreg} \neq [] \wedge \mathrm{goal} \neq [] \\
\qquad \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}) \wedge \mathrm{breg}' \neq \bot \\
\qquad \wedge\ \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{sub}'[\mathrm{breg}'] = \mathrm{subreg} \wedge \mathrm{cll}'[\mathrm{breg}'] = \mathrm{cllreg} \\
\qquad \wedge\ (\exists\ \mathrm{F}.\ \langle \mathrm{STACK\#}(\mathrm{b}'[\mathrm{breg}'], \mathrm{b}'; \mathrm{stack}) \rangle \\
\qquad\qquad\quad (\quad \langle \mathrm{STACK\#}(\mathrm{breg}, \mathrm{b}; \mathrm{stack}_0) \rangle\ \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) = \mathrm{stack}_0 \\
\qquad\qquad\quad\ \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseq}'[\mathrm{breg}']) = \mathrm{decglseqreg} \wedge \mathrm{F}[\bot] = \bot \\
\qquad\qquad\quad\ \wedge\ \mathrm{F\ injon\ stack} \wedge \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) \subseteq \mathrm{s} \wedge \mathrm{stack} \subseteq \mathrm{s}' \wedge \mathrm{breg}' \in \mathrm{s}' \\
\qquad\qquad\quad\ \wedge\ \mathrm{decglseq}'[\mathrm{breg}']\ \mathrm{cutptsin\ stack} \\
\qquad\qquad\quad\ \wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad \rightarrow \quad \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}[\mathrm{F}[\mathrm{n}]] \wedge \mathrm{cll}'[\mathrm{n}] = \mathrm{cll}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseq}'[\mathrm{n}]) = \mathrm{decglseq}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{decglseq}'[\mathrm{n}]\ \mathrm{cutptsin\ cdr(stack\ from\ n)} \\
\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}[\mathrm{n}] \neq [] \\
\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}[\mathrm{n}])))))
\end{array}
$$

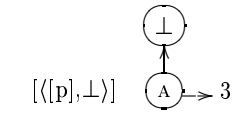
Chapter 13

3/4: Determinacy Detection

13.1 Definition of ASM4

In the refinement of ASM2 to ASM3 we have removed the unnecessary deallocation and reallocation of choicepoints. But there is another possibility for optimization, namely choicepoints with an empty list of candidates ("empty choicepoints").

As an example in Fig. 11.2, p. 67 from Sect. 11.2 both choicepoints A' (in ASM3 A is reused) and C point to an empty list of clauses, i.e. $clause(cll[A'],db_2) = clause(cll[C],db_2) = null$. If such an empty choicepoint is visited in *retry* rule, *deep-backtrack* is called and the choicepoint is simply removed. This behavior can be optimized by avoiding the creation of empty choicepoints altogether with look-ahead tests ("'determinacy detection"'). For the *try* rule this means, that a choicepoint need not be created when $procdef_2(act,db_2)$ gives only one clause. In the *retry* rule a choicepoint can be removed altogether instead of modifying it, when the stored alternatives become empty. The test for an empty choicepoint becomes obsolete. The state of ASM2 from Fig. 11.2 then corresponds to the state of ASM4 shown in Fig. 13.1.



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

Figure 13.1

The modified *try*- and *retry* rule of ASM4 are

```
try rule
    if mode = try
    then mode := enter
         if clause'(cllreg +1, db2) ≠ null
         then let tmp = new(s)
              s := s ∪ {tmp}
              b[tmp] := breg
              decglseq[tmp] := decglseqreg
              sub[tmp] := subreg
              cll[tmp] := cllreg +1
              breg := tmp

retry rule
    if mode = retry
    then decglseqreg := decglseq[breg]
         subreg := sub[breg]
         cllreg := cll[breg]
         ctreg := b[breg]
         mode := enter
            /* look ahead guard */
         if clause(cll[breg] +1,db2) ≠ null
         then cll[breg] := cll[breg] +1
         else breg := b[breg]
```

13.2 Equivalence Proof 3/4

To verify the equivalence between ASM3 and ASM4 a bijection F between the nonempty choicepoints of ASM3 and ASM4 is needed. Whether the function is defined to map nonempty choicepoints of ASM3 to ones of ASM4 or the other way round is not too important, it only determines which of the two stacks has to be computed with a call to *STACK#* (the other stack then is the image under F). To be consistent with [BR95] we have chosen to map the stack of ASM3 to the one of ASM4.

As the critical point in the definition of the coupling invariant it remains to define a correspondence between the cutpoints, To this purpose we use a program *F#* that maps each cutpoint of ASM3 to the next one below it in the stack that is nonempty. Program *G#* applies *F#* to all cutpoints of a *decglseq*. Applying first *G#* and then F (with F_d) on a *decglseq* of ASM3 then gives the corresponding *decglseq* of ASM4. Again a first-order definition is not possible since inconsistency due to cyclic pointer structures has to be avoided. Figure 13.2 graphically shows the correspondence between the two choicepoint stacks. Empty choicepoints are shown as a "o".

The formal definition of the procedures *F#* and *G#* is

```
F#(n,b,cll,db2;var n0)
begin
if n = ⊥
then n0 := n
else if clause(cll[n],db2) = null
     then F#(b[n],b,cll,db2;n0)
     else n0 := n
end


G#(decglseqreg,b,cll,db2;var decglseqreg0)
begin
if decglseqreg = [] then decglseqreg0 := [] else
```

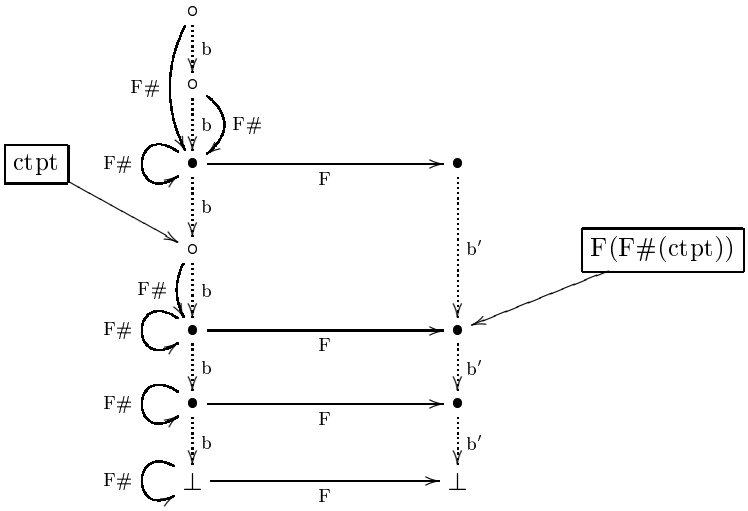


Figure 13.2 : Corresponding Choicepoints in ASM3 and ASM4

var ctpt_0, cont_0 **in**
begin
F#(ctpt,b,cll,db_2;ctpt_0);
G#(cont,b,cll,db_2;cont_0);
decglseqreg_0 := [⟨act, ctpt_0⟩ | cont_0]
end
end

It corrects and simplifies the definitions of F and G given in [BR95].

As a first approach for a coupling invariant the considerations above suggest

$\text{INV}_{34} \equiv$
$\exists$ F.
stop = stop′ ∧ vireg = vireg′ ∧ subreg = subreg′
∧ cllreg = cllreg′ ∧ F[⊥] = ⊥ ∧ mode = mode′
∧ ⟨F#(breg, b, cll, db_2; breg_0)⟩ F[breg_0] = breg′
∧ ⟨STACK#(breg,b;stack)⟩
⟨G#(decglseqreg,b,cll,db_2;bf var decglseqreg_0)⟩
F_d(F,decglseqreg_0) =decglseqreg′
∧ ∀ n. n ∈ stack
→ sub[n] = sub′[n] ∧ cll[n] = cll′[n]
∧ ⟨F#(b[n], b, cll, db_2; n_0)⟩
F[n_0] = b′[F[n]]

The two conjuncts with calls to $F\#$ and the formula $F[\bot] = \bot$ describe the construction of the ASM4 stack from the ASM3 stack. Most of the rules of ASM3 correspond to the same rule in ASM4. Only applications of the *retry* rule, that remove an empty choicepoint with *deep-backtrack* have no counterpart in ASM4. We have a 1:0 diagram for this case and 1:1 diagrams otherwise. Therefore the function *ndt* from Chapter 6 no longer has the constant value *mn*. Instead we have to define[1] *ndt* by

stop = run ∧ decglseqreg ≠ [] ∧ goal ≠ []
∧ mode = retry ∧ clause(cll[breg],db_2) = null
⊃ ndt(x,x′) = m0 ; ndt(x,x′) = mn

[1] $A \supset B; C$ abbreviates $(A \to B) \wedge (\neg A \to C)$, see Appendix B.

In the definition, as usual $\underline{x}$ =*decglseqreg, decglseq, stop, ...* and $\underline{x}'$ = *decglseqreg'*, *decglseq'*, *stop'*, ... denote the vectors of all dynamic functions of ASM3 and ASM4 (translated to program variables). To apply the modularization theorem from Chapter 6, we also need to define a function *exec0n* that bounds the number of successive triangular 1:0 diagrams, i.e. of successive calls to *deep-backtrack*. Such a bound is obviously given by the size of the ASM3 stack (computed with #). With this instance, proof obligation (6.6) from Chapter 6 becomes

$$\begin{aligned}
&\text{stop} = \text{run} \wedge \text{INV}_{34} \wedge \text{decglseqreg} \neq [] \wedge \text{goal} \neq [] \\
&\wedge\ \text{mode} = \text{retry} \wedge \text{clause(cll[breg],db}_2) = \text{null} \\
&\wedge\ \langle \text{STACK\#(breg,b;stack)} \rangle\ \#(\text{stack}) = \text{m} \\
\rightarrow\ &\langle \text{RULE}_3 \rangle\ (\quad \text{INV}_{34} \\
&\qquad \wedge\ (\langle \text{STACK\#(breg,b;stack)} \rangle\ \#(\text{stack}) < \text{m} \vee \text{stop} = \text{failure}))
\end{aligned}$$

The disjunct $ndt(\underline{x}, \underline{x}') \neq m0$ in the postcondition has been strengthened to $stop = failure$, since this is the only case, where ASM3 does not reduce the size of the stack.

It should be noted, that the precondition of the proof obligation does *not* include $stop' = run$. Just on the contrary proof obligation (6.9) from Chapter 6 now requires to prove that

$$\text{stop} = \text{run} \wedge \text{stop}' \neq \text{run} \wedge \text{INV}_{34} \rightarrow \text{ndt}(\underline{\text{x}},\underline{\text{x}}') = \text{m0}$$

holds. This results in the main problem for the verification: it must be made sure that INV_{34} holds, when ASM4 has already terminated, while ASM3 still has to remove empty choicepoints. This situation of *asynchronous termination* complicates the definition of the coupling invariant. In it we do not have $stop = stop'$, and also $mode = mode'$ is violated. So we have to weaken these properties in the coupling invariant to

$$\begin{aligned}
&(\text{stop}' \neq \text{failure} \rightarrow \text{stop} = \text{stop}' \wedge \text{mode} = \text{mode}') \\
\wedge\ &(\quad \text{stop}' = \text{failure} \wedge \text{stop} \neq \text{failure} \\
&\quad \rightarrow \text{mode} = \text{retry} \wedge \text{breg}' = \perp)
\end{aligned}$$

Together with the property

$$\langle \text{F\#(breg, b, cll, db}_2\text{; breg}_0) \rangle\ \text{F}[\text{breg}_0] = \text{breg}'$$

already present in the invariant, it is guaranteed that in the critical case, where ASM4 has stopped, all choicepoints in the stack of ASM3 are empty.

As always this approach for the coupling invariant is still insufficient for the equivalence proofs. Like in 1/2 and 2/3 we additionally need the injectivity of F, but this time only for *nonempty* choicepoints. Also the *cutptsin* property and the existence of $act[n]$ for every choicepoint n are required. Finally we need to mention a number of preconditions for single rule applications like $mode' = retry \rightarrow breg' \neq \perp$, and a characterization of *ctreg* and *ctreg'* in terms of *breg* and *breg'*. These properties were easy to find, and after 2 weeks of work and 5 iterations the following, correct coupling invariant was found.

$$\begin{aligned}
&\text{INV}_{34} \equiv \\
&\exists\ \text{F}. \\
&\qquad (\text{mode} = \text{try} \rightarrow \text{ctreg} = \text{breg} \wedge \text{clause}'(\text{cllreg}, \text{db}_2) \neq \text{null}) \\
&\quad \wedge\ (\quad \text{mode} = \text{enter} \\
&\qquad \rightarrow \quad \text{breg} \neq \perp \wedge \text{ctreg} = \text{b[breg]} \wedge \text{subreg} = \text{sub[breg]} \\
&\qquad\quad \wedge\ \text{clause}'(\text{cllreg}, \text{db}_2) \neq \text{null} \wedge \text{cllreg+1} = \text{cll[breg]} \\
&\qquad\quad \wedge\ \text{decglseqreg} = \text{decglseq[breg]})
\end{aligned}$$

$\wedge$ (mode$'$ = retry $\rightarrow$ breg$'$ $\neq$ $\perp$)
$\wedge$ (mode$'$ = try $\rightarrow$ ctreg$'$ = breg$'$ $\wedge$ clause$'$(cllreg$'$, db$_2$) $\neq$ null)
$\wedge$ (mode$'$ = enter $\rightarrow$ clause$'$(cllreg$'$, db$_2$) $\neq$ null)
$\wedge$ (mode$'$ = enter $\wedge$ clause$'$(cllreg$'$+1, db$_2$) $\neq$ null
$\rightarrow$ breg$'$ $\neq$ $\perp$ $\wedge$ ctreg$'$ = b$'$[breg$'$])
$\wedge$ (mode$'$ = enter $\wedge$ clause$'$(cllreg$'$+1, db$_2$) = null $\rightarrow$ ctreg$'$ = breg$'$)

$\wedge$ F[$\perp$] = $\perp$ $\wedge$ $\perp$ $\in$ s $\wedge$ $\perp$ $\in$ s$'$ $\wedge$ breg $\in$ s $\wedge$ ctreg $\in$ s
$\wedge$ vireg = vireg$'$ $\wedge$ subreg = subreg$'$ $\wedge$ cllreg = cllreg$'$
$\wedge$ (mode = retry $\rightarrow$ breg $\neq$ $\perp$ $\wedge$ decglseqreg $\neq$ [] $\wedge$ goal $\neq$ [])
$\wedge$ (decglseqreg$'$ = [] $\vee$ goal = [] $\rightarrow$ mode = call)
$\wedge$ (stop$'$ $\neq$ failure $\rightarrow$ mode = mode$'$ $\wedge$ stop = stop$'$)
$\wedge$ (stop$'$ = failure $\wedge$ stop $\neq$ failure
$\rightarrow$ stop = run $\wedge$ mode = retry $\wedge$ breg$'$ = $\perp$)
$\wedge$ $\langle$F#(breg, b, cll, db$_2$; n$_0$)$\rangle$ F[n$_0$] = breg$'$
$\wedge$ $\langle$G#(decglseqreg, b, cll, db$_2$; decglseqreg$_0$)$\rangle$
F$_d$(F, decglseqreg$_0$) = decglseqreg$'$
$\wedge$ $\langle$STACK#(breg, b; stack)$\rangle$
(stack $\subseteq$ s $\wedge$ (mode $\neq$ retry $\rightarrow$ decglseqreg cutptsin stack)
$\wedge$ ($\forall$ n. n $\in$ stack
$\rightarrow$ decglseq[n] cutptsin cdr(stack from n)
$\wedge$ decglseq[n] $\neq$ [] $\wedge$ goal[n] $\neq$ [])
$\wedge$ ($\forall$ n. n $\in$ stack $\wedge$ clause$'$(cll[n], db$_2$) $\neq$ null
$\rightarrow$ F[n] $\in$ s$'$ $\wedge$ F[n] $\neq$ $\perp$ $\wedge$ decglseq[n] $\neq$ [] $\wedge$ goal $\neq$ []
$\wedge$ $\langle$F#(b[n], b, cll, db$_2$; n$_0$)$\rangle$ F[n$_0$] = b$'$[F[n]]
$\wedge$ $\langle$G#(decglseq[n], b, cll, db$_2$; decglseqreg$_0$)$\rangle$
F$_d$(F, decglseqreg$_0$) = decglseq$'$[F[n]]
$\wedge$ cll[n] = cll$'$[F[n]] $\wedge$ sub[n] = sub$'$[F[n]]
$\wedge$ ($\forall$ n$_1$. n$_1$ $\in$ stack $\wedge$ clause$'$(cll[n$_1$], db$_2$) $\neq$ null
$\wedge$ n $\neq$ n$_1$
$\rightarrow$ F[n] $\neq$ F[n$_1$])))

With hindsight this invariant could be simplified by merging some of the 1:1 diagrams which deterministically are successors of each others. This is the case for the rule sequences *call* (second case that does not backtrack) *try*, *enter* (which gives a 3:3 diagram) and *retry*, *enter* (2:2 diagram). Using larger diagrams would reduce the number of states, in which the coupling invariant must hold. Specifically all conjuncts with one of the preconditions *mode* = *try*, *mode*$'$ = *try*, *mode* = *enter* or *mode*$'$ = *enter*, i.e. the first 11 lines of the invariant, could be removed.

Chapter 14

4/5: Linear Compilation of Predicate Structure

14.1 Definition of ASM5

The first three refinement steps can be viewed as an optimization of the first ASM which do not change the representation of the Prolog program. In contrast, the refinement from ASM4 to ASM5 compiles the predicate structure of Prolog. For the first time instructions are introduced, which will also be present in the final WAM. We will deviate in this section from [BR95] insofar, as the code of ASM5 will first contain *linear* chains, not the more complex *nested* chains, which we will define in ASM6 (a precise definition of "chains" will be given below). The reason is, that the refinement 4/5 allows to study the typical problems of a compilation step, without having to consider the problems of m:n diagrams simultaneously.

The general idea of the refinement step is to move control over the rule to be executed from the *mode*-Variable to the actual code. To do this, *cllreg* no longer points to the line of a clause, but to an address, where *instructions* are stored. *cllreg* becomes a program counter, and is therefore renamed to *preg*. Similarly the clause line *cll[n]* stored in choicepoints becomes a code pointer *p[n]*.

The instruction stored at *preg* is now the result of a function *code*, that replaces *clause*. Checks for the value of *mode* are replaced by checks on the type of the instruction *code*(*preg*,db_5), where db_5 is the database of ASM5. Possible instructions may at this stage still be clauses (they are replaced by finer-grained instructions in the refinements 8/9 and 9/10), but additionally we now have the control instructions *try_me_else*, *retry_me_else* and *trust_me*, which replace the rules *try* and *retry* (then and else case).

To understand the effect of the control instructions, consider the following example clauses for a predicate `p`:

```
p(X)    :- body1.
p(f(X)) :- body2.
p(g(X)) :- body3.
p(g(X)) :- body4.
```
(14.1)

In the refinement of ASM4 to ASM5 they are translated to the code fragment (labels L1 – L4 are symbolic addresses):

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: retry_me_else(L4)
    p(g(X)) :- body3.
L4: trust_me
    p(g(X)) :- body4.
```
(14.2)

On a query `?- p(X)`, call rule of ASM5 (called when *preg* is at a special *start* address) will set *preg* to the start address L1 of the clauses for `p` (a special address *failcode* is used as the result of the *procdef* function, when no clauses are available for an activator).

call rule
if is_user_defined(act) ∧ preg = start
then ctreg := breg
if code(procdef_5(act,db_5)) = failcode
then backtrack
else preg := procdef_5(act,db_5)

where

backtrack ≡
if breg = ⊥
then stop := failure
else preg := p[breg]

Execution of `try_me_else(L2)` at address `L1` with the *try_me* rule will have the same effect, that *try* rule in ASM4 had.

try_me rule
if code(preg,db_5) = try_me_else(N)
then let tmp = new(s)
s := s ∪ {tmp}
breg := tmp
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
p[tmp] := N
preg := preg +1

The address for alternative clauses stored in the choicepoint is `L2` and execution continues with the next address. The clause there is executed with *enter* rule, which has the same effect as in ASM4. Since it must activate *call* rule on successful invocation, it sets *preg* := *start*.

enter rule
if is_user_defined(act) ∧ code(preg,db_5) = clause
then let cla = rename(clause,vi)
let mgu = unify(act, hd(cla))
if mgu = nil
then backtrack
else decglseqreg := mgu $\hat{}_d$ [<bdy(cla),ctreg> | cont]
subreg := subreg ∘ unify
vi := vi +1
preg := start

When *preg* is set to `L3` or to `L5` by backtracking, the *retry_me* rule resp. the *trust_me* rule are executed. They correspond to the then- and the else-branch of *retry* rule of ASM4. The case distinction is no longer done at run time, but at compile time.

retry_me rule
if code(preg,db_5) = retry_me_else(N)
then decglseqreg := decglseq[breg]
subreg := sub[breg]
ctreg := b[breg]
p[breg] := N
preg := preg +1

trust_me rule
if code(preg,db_5) = trust_me
then decglseqreg := decglseq[breg]
ctreg := b[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

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

$$\begin{array}{l} \quad db_5 = \text{compile}_{45}(db_2) \\ \rightarrow [\text{CLLS\#}(\text{procdef}_2(\text{act},db_2),db_2,db_2;\text{col}_1)] \\ \quad \langle \text{L-CHAIN\#}(\text{procdef}_5(\text{act},db_5),db_5;\text{col}_2)\rangle \\ \qquad \text{mapclause}'(\text{col}_1,db_2) = \text{mapclause}'(\text{col}_2,db_5) \end{array} \tag{14.3}$$

$procdef_2$ and db_2 are the *procdef* function and the Prolog program that have been used in the ASM2, ASM3 and ASM4. $procdef_5$ is the new *procdef*-function for ASM5 and db_5 is the compiled Prolog program. The procedure *L-CHAIN#* terminates, iff the code fragment stored at $procdef_5(act,db_5)$ is a linear chain, and delivers the clauses contained in it. As for *stackof* (see p. 71 in Sect. 11.2) a definition a first-order function *l-chain* instead of the procedure is *not sufficient* to characterize linear chains. By the termination of the procedure cyclic chains have to be ruled out as possible results of the compilation. A precise definition of the *L-CHAIN#* program is given in appendix D.1.

14.2 Equivalence Proof 4/5

A precise analysis of the refinement from ASM4 to ASM5 shows that it does not just replace *mode* with instructions. Also the test $clause(procdef_2(act,db_2)) = null$ is moved from *try rule* (ASM4) to *call rule* (ASM5). This modification can also be done in ASM4. Just replace *try rule* and *call rule* with

call rule
if stop = run $\wedge$ mode = call
$\wedge$ is_user_defined(act)
then if clause(procdef$_2$(act,db_2)) = null
then backtrack

```
        else cllreg := procdef2(act,db2)
             ctreg := breg
             if clause(procdef2(act,db2)+1, db2) ≠ null
             then mode := try
             else mode := enter


try rule
    if stop = run ∧ mode = try
    then mode := enter
         let tmp = new(s)
         s := s ∪ {tmp}
         breg := tmp
         b[tmp] := breg
         decglseq[tmp] := decglseqreg
         sub[tmp] := subreg
         cll[tmp] := cllreg +1
```

If we call the result ASM4a, then the refinement of ASM4a to ASM5 only contains 1:1 diagrams.

In the verification of the refinement from ASM4 to ASM4a we must consider a 2:1 and a 2:2 diagram for the case where *mode* = *call* and no backtracking happens, depending on whether $clause(procdef_2(act,db_2)) = null$ holds. Otherwise the verification is trivial, since obviously identity suffices as coupling invariant.

The verification of 4a/5 was the subject of the diploma thesis of Wolfgang Ahrendt at the university of Karlsruhe ([Ahr95]). Details are also given in [SA98].

About one month of work and 9 iterations were necessary to find the correct coupling invariant. The complexity of the proofs is about the same as for the refinement 1/2. The main problem in the development of the coupling invariant is to transform the compiler assumption into suitable connections between the choicepoints. E.g. in the case *mode* = *retry* we must have that for each choicepoint n the code chain of ASM5 at $pc[n]$ starts with a *retry_me_else* or *trust_me* and contains the same clauses as the clause list of ASM4 starting with $cll[n]$. Formally we have to add

$$\langle \text{CLLS\#}(\text{cll[n]},\ \text{db}_2;\ \text{col}_1)\rangle$$
$$\quad \langle \text{L-CHAIN-RETRY-ME\#}(\text{p[n]},\ \text{db}_5;\ \text{col}_2)\rangle$$
$$\quad\quad \text{mapcode}(\text{col}_2,\ \text{db}_5) = \text{mapclause}'(\text{col}_1,\ \text{db}_2))$$

to the coupling invariant. The use of a subprocedure (here *C-CHAIN-RETRY-ME#*) of the procedure *L-CHAIN#* used in the compiler assumption is typical for compilation steps (for the definition of *L-CHAIN#* see appendix D.1). To have a simple coupling invariant, it is recommendable to structure the procedures in the compiler assumptions according to the structure of ASM runs.

The most important result of the formal verification of 4a/5 was that an unintended indeterminism was revealed in ASM3 and ASM4. The problem was found when verifying 4a/5, since this refinement was verified before refinements 2/3 and 3/4.

To see the problem, consider again the *fail rule* from ASM3 (p. 76), that is also used in ASM4. The obvious *intention* of the rule is that retry rule should be executed afterwards.

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

Although the error is easy to correct (the conjunct *mode* = *call* must be added to the guard of *fail* rule), we think this is a typical error that is very difficult to find even by intensive inspection (and, of course, we *had* to inspect the code thoroughly before we could make an attempt to define a

coupling invariant). A reader will always unconsciously resolve the indeterminism in the intended way. Nevertheless, an implementation is blind for intentions, and will possibly resolve the conflict in the wrong way (and ours did!).

The coupling invariant required for successful verification is:

$$
\begin{array}{l}
\mathrm{INV}_{45} \equiv \\
\quad \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{subreg}' \wedge \mathrm{breg} = \mathrm{breg}' \\
\wedge\ \mathrm{ctreg} = \mathrm{ctreg}' \wedge \mathrm{decglseqreg} = \mathrm{decglseqreg}' \wedge \mathrm{s} = \mathrm{s}' \wedge \mathrm{breg} \in \mathrm{s} \wedge \mathrm{ctreg} \in \mathrm{s} \\
\wedge\ \mathrm{decglseqreg\ ctpelem\ s} \\
\wedge\ (\mathrm{mode} = \mathrm{call} \rightarrow \mathrm{preg} = \mathrm{start}) \\
\wedge\ (\mathrm{mode} = \mathrm{retry} \rightarrow \mathrm{breg} \neq \bot \wedge \mathrm{preg} = \mathrm{p}[\mathrm{breg}']) \\
\wedge\ (\mathrm{mode} = \mathrm{enter} \rightarrow \mathrm{code}(\mathrm{preg}, \mathrm{db}_5) = \mathrm{mkcl}(\mathrm{the\_clau}(\mathrm{clause}'(\mathrm{cllreg}, \mathrm{db}_2)))) \\
\wedge\ (\quad \mathrm{mode} = \mathrm{try} \\
\quad \rightarrow \quad \mathrm{is\_user\_defined}(\mathrm{act}) \\
\qquad \wedge\ \langle \mathrm{CLLS\#}(\mathrm{cllreg}, \mathrm{db}_2; \mathrm{col}_1) \rangle \\
\qquad\quad \langle \text{L-CHAIN-TRY-ME\#}(\mathrm{preg}, \mathrm{db}_5; \mathrm{col}_2) \rangle \\
\qquad\qquad \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_5) = \mathrm{mapclause}'(\mathrm{col}_1, \mathrm{db}_2)) \\
\wedge\ (\mathrm{decglseqreg} = [] \vee \mathrm{goal} = [] \vee \mathrm{act} = ! \vee \mathrm{act} = \mathrm{true} \rightarrow \mathrm{mode} = \mathrm{call}') \\
\wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{s} \wedge \mathrm{n} \neq \bot \\
\qquad \rightarrow \quad \mathrm{b}[\mathrm{n}] \in \mathrm{s} \wedge \mathrm{decglseq}[\mathrm{n}]\ \mathrm{ctpelem\ s} \wedge \mathrm{sub}[\mathrm{n}] = \mathrm{sub}'[\mathrm{n}] \\
\qquad\quad \wedge\ \mathrm{b}[\mathrm{n}] = \mathrm{b}'[\mathrm{n}] \wedge \mathrm{decglseq}[\mathrm{n}] = \mathrm{decglseq}'[\mathrm{n}] \wedge \mathrm{decglseq}[\mathrm{n}] \neq [] \\
\qquad\quad \wedge\ \mathrm{goal} \neq [] \wedge \mathrm{is\_user\_defined}(\mathrm{act}[\mathrm{n}]) \\
\qquad\quad \wedge\ \langle \mathrm{CLLS\#}(\mathrm{cll}[\mathrm{n}], \mathrm{db}_2; \mathrm{col}_1) \rangle \\
\qquad\qquad \langle \text{L-CHAIN-RETRY-ME\#}(\mathrm{p}[\mathrm{n}], \mathrm{db}_5; \mathrm{col}_2) \rangle \\
\qquad\qquad\quad \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_5) = \mathrm{mapclause}'(\mathrm{col}_1, \mathrm{db}_2))
\end{array}
$$

Chapter 15

5/7: Structured Compilation of Predicate Structure

15.1 Definition of ASM6 and ASM7

In ASMs 1–5 the problem, how to determine "relevant" clauses, which have a head that unifies with an activator, was encoded into the under-specified *procdef* function. In ASM7 this under-specification is removed by defining instruction sequences that select relevant clauses.

A concrete definition of the *procdef* function has to be between two extremes:

- A simple implementation, in which *procdef*(*act,db*) returns all clauses, which have a head that starts with the leading predicate symbol of *act*. This solution is inefficient, since it leads to a linear search in clauses, and causes a lot of (expensive) failed unification attempts. Consider e.g. a collection of facts `p(`c_1`), ..., p(`c_n`)` in a database.

- An elaborate solution, which selects exactly those clauses, which unify with the activator. Such a solution is possible using "discrimination nets" (see e.g. [Gra96]). It encodes the whole unification into clause selection.

The solution taken in the WAM is a compromise between both extremes. It uses the simple *procdef* function in the *call* rule and additional *switching* instructions, that select relevant "groups" of clauses depending on the leading function symbol of some argument of *act*. If e.g. the activator is of the form `p(t1,f(t2))`, then a switching instruction could select a group of clauses which have as second argument either a variable or `f`. Clauses with a second argument, that starts with a function symbol different from `f` would not be considered.

Before switching instructions can be introduced, first "grouping" of clauses must be made possible. This is done in ASM6 by allowing instruction sequences that form *nested* chains. Nested chains are defined like linear chains, but at each position where a linear chain contains a clause, a nested chain may contain another (nested) chain. Such an inner chain can be used to group similar clauses together, so that they can be skipped as a whole with a switching instruction in ASM7.

If we look at the example program (14.1) from Sect. 14.1, then we could for example group the last two clauses. The resulting code shown in Fig. 15.1 has a subchain for the two clauses starting at label `L4`.

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: trust_me                                                    (15.1)
L4: try_me_else(L5)
    p(g(X)) :- body3.
L5: trust_me
    p(g(X)) :- body4.
```

Allowing nested instead of linear chains requires only a minimal change in the ASM code. In the *retry_me_else* and *trust_me* instructions we can no longer load *ctreg* with $b[breg]$, since the cutpoint of the currently active goal need no longer be the father of *breg*. Instead *all* choicepoints that were constructed for the current goal have to be ignored. The number of these choicepoints is equal to the nesting depth of the chain the ASM currently works on. For the *trust_me* at `L5` it is 2, the correct value that should be assigned to *ctreg* in the rule therefore should be $ctreg := b[b[breg]]$. The *trust_me* at `L3` should set *ctreg* to $b[breg]$. To solve the problem, there are two alternatives. [BR95] leaves open which one to choose by not giving a concrete definition for the *restore_cutpoint* statement. The first solution is to add an additional argument to each *retry_me_else* and *trust_me* instruction, which records its current depth in the chain. The second solution is to store the correct *ctreg* within the choicepoint. We have chosen the second one, since according to [AK91] it is the one usually adopted. An additional component *ct* is added to each choicepoint and the new *try_me_else*, *retry_me_else* are *trust_me* rule are:

try_me rule
if code(preg,db_7) = try_me_else(N)
then let tmp = new(s)
s := s ∪ {tmp}
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
p[tmp] := N
breg := tmp
ct[tmp] := ctreg
preg := preg +1

retry_me_else rule
if code(preg,db_7) = retry_me_else(N)
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := N
preg := preg +1

trust_me rule
if code(preg,db_7) = trust_me
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

After ASM6 has made grouping instructions together possible, ASM7 allows to put switching instructions at the front of chains or subchains. There are three types:

- `switch_on_term(i,Lv,Lc,Ll,Ls)` jumps to address `Lv`, `Lc`, `Ll` or `Ls`, if the i^{th} argument *arg(act,i)* of the activator is a variable, a constant a list or a function term (a structure).
- `switch_on_struct(i,N,T)` assumes, that it has been already assured, that *arg(act,i)* is a structure. The address to jump to is found by looking up the leading function symbol in a table of triples (*f*,*j*,`L`). If *arg(act,i)* is a function term with leading function symbol f and j subterms, the instruction jumps to `L`. The selection of the jump address is encoded into an abstract function *hashs*. For the case described we have

  $$\text{hashs}(\text{arg}(\text{act,i}),\text{N},\text{T},\text{db}_7) = \texttt{L}$$

- `switch_on_const(i,N,T)` assumes similar to `switch_on_struct` that *arg(act,i)* is a constant and branches according to a table at address `T` that stores N pairs (*c*,`L`). For the abstract function *hashc* we have analogously

  $$\text{hashs}(\text{arg}(\text{act,i}),\text{N},\text{T},\text{db}_7) = \texttt{L}$$

  whenever *arg(act,i)* = *c*.

In our example we could add at `L4` the following switching instructions:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: trust_me
L4: switch_on_term(L7,failcode,failcode,L6)                    (15.2)
L6: switch_on_struct(1,1,T)
L7: try_me_else(L5)
    p(g(X)) :- body3.
L5: trust_me
    p(g(X)) :- body4.
```

Address T should contain a list with one element (*g*,1,`L7`). *failcode* is a special address, that leads to backtracking. This address must be returned by *hashs* and *hashc*, when the function or constant symbol is not found in the table. The ASM instructions for switching are

switch_on_term rule
if code(preg, db_7) = switch_on_term(i, N_s, N_c, N_v, N_l)
then let x_i = arg(act,i)
if is_struct(x_i) **then** preg := N_s **else**
if is_const(x_i) **then** preg := N_c **else**
if is_var(x_i) **then** preg := N_v **else**
if is_list(x_i) **then** preg := N_l;
if preg = failcode **then backtrack**

switch_on_constant rule
if code(preg, db_7) = switch_on_constant(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashc(table, tabsize, constsym(x_i), db_7);
if preg = failcode **then backtrack**

switch_on_structure rule
if code(preg, db_7) = switch_on_structure(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashs(table, tabsize, funct(x_i), arity(x_i),db_7);
if preg = failcode **then backtrack**

Note that the *failcode* address is used in the examples given in [BR95], but that the call of backtracking is missing in the ASM rules of appendix 2. In the rules given in [AK91] for *switch_on_struct* and *switch_on_const* the call is defined, but in the *switch_on_term* it is also realized only by the assumption never given explicitly, that *failcode* is the address of the backtracking routine.

To allow the use of clauses in several chains, ASM6 additionally introduces instructions *try*(L), *retry*(L) and *trust*(L). Their effect is identical to the one *try_me_else*(L), *retry_me_else*(L) and *trust_me*, except that the role of L and *preg* +1 as address of the choicepoint to create resp. address to continue the computation are exchanged.

try rule
if code(preg,db$_7$) = try(N)
then let tmp = new(s)
s := s ∪ {tmp}
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
p[tmp] := preg +1
breg := tmp
ct[tmp] := ctreg
preg := N

retry rule
if code(preg,db$_7$) = retry(N)
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := preg +1
preg := N

trust rule
if code(preg,db$_7$) = trust(N)
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := N

In our example above a meaningful use of the new instructions would be

```
    switch_on_term(L2,failcode,failcode,L1)
L1: switch_on_struct(1,1,T)
L2: try_me_else(L4)
    p(X)    :- body1.
L3: retry_me_else(L6)
L4: p(f(X)) :- body2.
L5: retry_me_else(L8)
L6: p(g(X)) :- body3.
L7: trust_me
L8: p(g(X)) :- body4.
L9: try(L6)
    trust(L8)
```
(15.3)

where the table T now has entries (g,1,L6) and (f,1,L9). With an activator `p(f(X))` ASM7 would execute the first two switching instructions. The last one would jump to `L9`. There, by execution of the *try* and *trust* the clauses at `L6` and `L8` would be tried.

Finally it should be remarked, that the code schemes given are only two of many possible ones. The compiler assumption of 5/7 allows a great number of alternatives, among others the variants "one-level switching" and "two-level switching" discussed in [AK91].

The compiler assumption

$$\begin{array}{l} \mathrm{db}_6 = \mathrm{compile}_{56}(\mathrm{db}_5) \rightarrow [\text{L-CHAIN\#}(\mathrm{procdef}_5(\mathrm{act},\mathrm{db}_5),\mathrm{db}_5;\ \mathrm{col}_1)] \\ \quad \langle \text{CHAIN\#}(\mathrm{procdef}_6(\mathrm{act},\mathrm{db}_6),\mathrm{db}_6;\ \mathrm{col}_2)\rangle \\ \qquad \mathrm{mapcode}(\mathrm{col}_1,\ \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_2,\ \mathrm{db}_6) \end{array} \tag{15.4}$$

for 5/6 is similar to the one for 4/5. By the introduction of switching instructions in ASM7 selection of relevant clauses for one leading predicate symbol is then moved from the *procdef* function to the switching instructions. Only the starting address for one leading predicate symbol must still be selected by a *procdef* function. The selection can now be done by a table lookup, abstractly encoded into a dynamic function $procdef_7$, which is a result of the compilation step from ASM6 to ASM7. Therefore we have for $compile_{67}(db_6) := \langle procdef_7, db_7\rangle$:

$$\begin{array}{l} [\text{CHAIN\#}(\mathrm{procdef}_6(\mathrm{act},\mathrm{db}_6),\mathrm{db}_6;\ \mathrm{col}_1)] \\ \quad \langle \text{S-CHAIN\#}(\mathrm{act},\ \mathrm{procdef}_7[\mathrm{id}(\mathrm{act})],\mathrm{db}_7;\ \mathrm{col}_2)\rangle \\ \qquad \mathrm{mapcode}(\mathrm{col}_1,\ \mathrm{db}_6) = \mathrm{mapcode}(\mathrm{col}_2,\ \mathrm{db}_7) \end{array} \tag{15.5}$$

In the compiler assumption *id* selects the leading predicate symbol of a literal including its arity. We have introduced selection of the leading predicate symbol in the refinement 6/7, since it seemed to be the logical consequence of the refinement idea for clause selection given in [BR95], p. 27. In [BR95] selection of the leading predicate symbol is done, without mentioning the change, only in the final ASM (the WAM).

The programs *CHAIN#* and *S-CHAIN#* in the compiler assumption characterize nested chains and nested chains with switching. A concrete definition of these programs is given in appendix D.2. The definition is significantly more complex that the definition given in [BR95], because cyclic chains have to be avoided. Also the fact, that switching instructions are allowed only *at the beginning* of subchains had to be made precise.

15.2 Equivalence Proof 5/7

An informal argument for the equivalence of ASM5, ASM6 and ASM7 is that they all try the same candidate clauses. To be a little more precise, all 3 ASMs go through the same sequence of *call* and *enter* rules with the same activators *act* and the same candidate nodes (in the remaining chain starting with *preg*). Unfortunately this informal argument, which is also given in [BR95], is far away from a formal proof. Although it suggest to decompose the commuting diagram into subdiagrams with corners at states where *preg* = *start* and *is_clause*(*code*(*preg*,*db*)), it does neither give a hint how to set up a correspondence between states, nor how to prove the commutativity of the subdiagrams.

To make the verification manageable, we therefore had to solve the following three problems, that will be discussed in the following sections:

- Define a precise correspondence between the choicepoint stacks.
- Given the correct correspondence between choicepoints, define another one for the cutpoints stored in the *decglseq*'s. This results in a first approach to define the coupling invariant.
- Finally verify the subdiagrams. These now have no fixed size any longer as in all previous refinements. Their size now depends on the number of instructions in the code chains. We discuss two methods two verify diagrams with datastructure-dependent size.

Trying to solve the first problem, one immediately finds that it is easier to verify the refinement 5/7 than to verify 6/7. In the first case the *one* choicepoint that is allocated for an activator in ASM5 must be compared with the corresponding set of choicepoints in ASM7 (like for 5/6), for the second case two sets of choicepoints must be compared. We have first verified refinement 5/6, quasi as a "preliminary study" for the problems that will occur in 5/7. We will discuss the three problems described above first for the refinement 5/6 and will then show how much the solutions developed for 5/6 had to be changed for 5/7.

Correspondence of Choicepoint Stacks To model the correspondence of choicepoint stacks we first used for 5/6 as well as for 5/7 a dynamic function $H : node \rightarrow nodelist$ that given an ASM5 choicepoint returns the corresponding ones of ASM6 resp. ASM7. The function is used existentially quantified in the coupling invariant just like function F was used in the verification of 1/2 (see Sect. 11.2). Appending of all the (nonempty) lists $H[n]$ for all stack nodes n of ASM5 should give the stacks of ASM6 resp. ASM7. The (remainder of a) chain starting at $p[n]$ (computed with *CHAIN-RET#*) should contain the same clauses as can be computed by appending the clauses that are stored in the chains $p'[n']$ for $n' \in H[n]$ (these clauses are computed with the program *APP-CHAINS-RET#*). Also the $sub[n]$ and the goals in $decglseq[n]$ should be identical to $sub[n']$ and $decglseq[n']$. Formalized this can be written as:

$$
\begin{array}{l}
\langle \text{STACK\#(breg,b;stack)} \rangle \\
\quad (\quad \langle \text{STACK\#(breg}',\text{b}';\text{stack}') \rangle \ \text{stack}' = \text{H}_1(\text{H,stack}) \\
\quad\ \wedge \forall \ \text{n.} \ \text{n} \in \text{stack} \\
\qquad\qquad \rightarrow \langle \text{L-CHAIN-RET\#(p[n], db}_5;\text{col}_1) \rangle \\
\qquad\qquad\quad \langle \text{S-APP-CHAINS-RET\#(decglseq}',\text{p,H[n],db}_7;\text{col}_2) \rangle \\
\qquad\qquad\qquad \text{mapclause(col}_1,\text{db}_5) = \text{mapclause(col}_2,\text{db}_7)
\end{array}
$$

Now it turns out, that this formula is a correct description of the correspondence of ASM5 and ASM6, but insufficient for 5/7. The reason is, that in ASM7 choicepoints n are possible, for which the chain starting at $p[n]$ does contain *no* clauses at all (i.e. a suitable call to *S-CHAIN-RET#* computes an empty list of clauses). For such a choicepoint, which we call *empty* in the following, there is *no corresponding choicepoint in ASM5*.

An example for such an empty choicepoint can be constructed for the following example program, where we assume that table T contains the two entries $(f, 1, L5)$ and $(g, 1, L7)$:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: trust_me
    switch_on_term(L4,failcode,failcode,L3)
L3: switch_on_struct(1,2,T)                                    (15.6)
L4: try_me_else(L6)
L5: p(f(X)) :- body2.
L6: trust_me
L7: p(g(X)) :- body3.
```

For an activator `p(h(c))` an empty choicepoint n is present while the first clause is considered. During this $p[n]$ points to `L2` (allocated in the *try_me_else* instruction). But execution of the instructions at `L2` will lead to backtracking in the *switch_on_struct*) instruction, without any clause being considered. Nevertheless the empty choicepoint is present, while *body1* is executed. On the other hand, in ASM5 no choicepoint is constructed for the activator `p(h(c))`, since the code of ASM5 consists according to the compiler assumption

$$\langle \text{L-CHAIN\#(procdef}_5(\text{act,db}_5),\ \text{db}_5;\ \text{col}_1) \rangle\ \text{col}_1 = [\texttt{p(X) :- body1}]$$

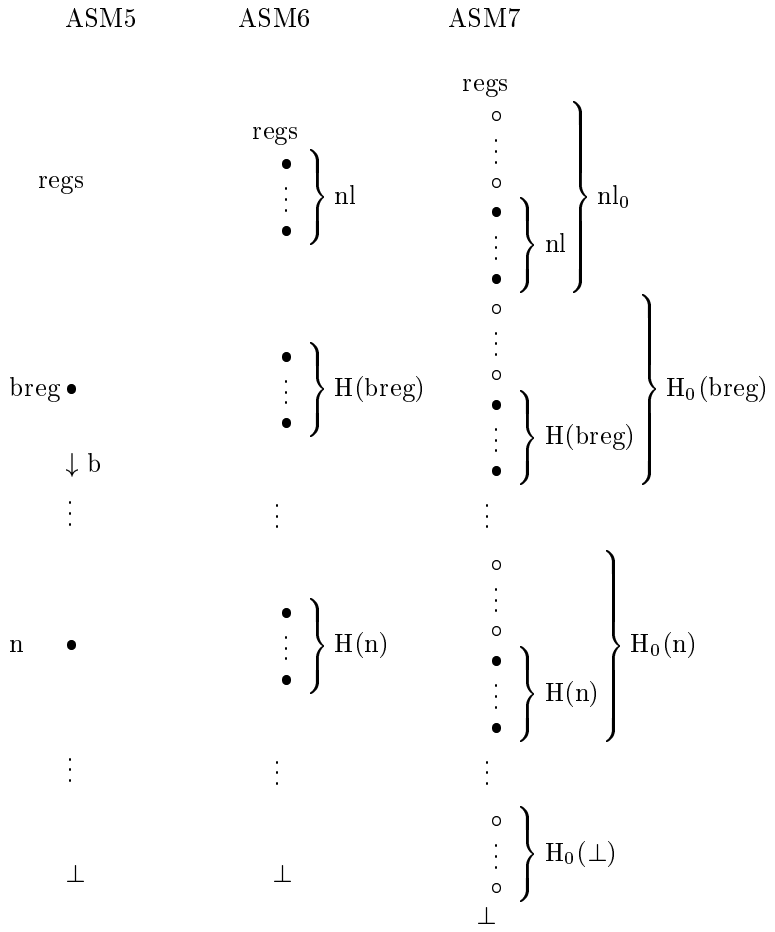


Figure 15.1

of only the first clause. Summarizing, the image of the ASM5 stack under H is not the whole ASM7 stack, but between the images $H[n]$ and $H[b[n]]$ of two successive choicepoints there may be an arbitrary number of empty choicepoints.

Figure 15.1 depicts the situation graphically. Empty choicepoints are represented as '∘'. *regs* are the current values of the registers *decglseqreg*,*subreg* and *cllreg*. The figure shows, that the contents of ASM5 registers not only correspond to the registers of ASM6 resp. ASM7, but also to an additional list *nl* of choicepoints. It is also shown that we have formalized the problem of empty choicepoints using an additional function H_0 and an additional list nl_0. It should be noted that at the lower end of an ASM7 stack there may also be a list $H_0(\bot)$ of empty choicepoints. This causes the problem of asynchronous termination just as in the refinement 3/4.

Correspondence of Cutpoints For the refinement 5/6 a cutpoint *ctpt* of ASM5 is simply mapped to $car(H[ctpt])$, the topmost corresponding Cutpoint in ASM6. $H_d(H,decglseq[n])$ maps all cutpoints of $decglseq[n]$ in this way.

We made a similar assumption, that *ctpt* should be mapped to $car(H_0[ctpt])$ also in our first proof attempt for 5/7. But a thorough analysis why it failed showed, that the cutpoint of ASM7 corresponding to *ctpt* maybe located *anywhere* between $H[ctpt]$ and $H[b[ctpt]]$ or may be the first element of $H[b[ctpt]]$. There is even an exception for $b[ctpt] = \bot$: then the corresponding cutpoint may be in $H_0[\bot]$ or it may be $\bot$ itself. The formal definition of similarity between *decglseq*'s of ASM5 and ASM7 is therefore ($cdr([])$ ist defined as [] here):

$$
\begin{array}{l}
\mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [],\ []),\\
\neg\ \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [\langle \mathrm{goal},\mathrm{ctpt}\rangle,\mathrm{dgl}],[]),\\
\neg\ \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [],[\langle \mathrm{goal}',\mathrm{ctpt}'\rangle,\ \mathrm{dgl}']),\\
\quad \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [\langle \mathrm{goal},\mathrm{ctpt}\rangle,\ \mathrm{dgl}],[\langle \mathrm{goal}',\mathrm{ctpt}'\rangle,\ \mathrm{dgl}'])\\
\leftrightarrow \quad \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ \mathrm{dgl},\ \mathrm{dgl}') \wedge \mathrm{goal} = \mathrm{goal}'\\
\quad \wedge\ (\mathrm{ctpt} = \bot \supset \mathrm{ctpt}' \in \mathrm{H}_0[\bot] \vee \mathrm{ctpt}' = \bot;\\
\qquad\qquad\qquad \mathrm{ctpt}' \in \mathrm{H}_0[\bot] \wedge \mathrm{ctpt}' \notin \mathrm{cdr}(\mathrm{H}[\mathrm{ctpt}])
\end{array}
$$

Diagrams with Datastructure Dependent Size The commuting diagrams in the refinements 5/6 and 5/7 are no longer diagrams of some type $m{:}n$ with some constants m, n (e.g. $m = 1$, $n = 2$). Instead n is determined by the number of instructions, that have to be executed until the next clause is reached. That n is finite, is implicitly guaranteed by the termination of the *CHAIN#* resp. *S-CHAIN#* program from the compiler assumption, but for a formal (inductive) argument we need an explicit size n. An explicit definition is easy for 5/6, since the number of instructions in a chain corresponds directly to the number of clauses stored in the chain. For ASM7 this is not the case, since empty chains of arbitrary length are possible. Therefore appendix D.3 defines a procedure *S-COUNT#* which explicitly counts the remaining instructions in the chain. The termination of *S-COUNT#* should be intuitively clear, since it follows the same recursion structure than *S-CHAIN#*. But for a formal proof we need the new proof principle of *induction over the recursion depth of procedures*, that was described in Sect. 3. It allows to prove the termination of *S-COUNT#* (as well as the termination of all auxiliary procedures mentioned in appendix D.3) easily.

To prove the commutation of diagrams of datastructure dependent size, we then have 2 alternatives, that we will discuss in the following. Either we can recursively decompose them, or we can prove auxiliary lemmata for each single ASM.

Recursive Decomposition of Diagrams This technique was applied in the verification of 5/6. It interprets each $m{:}n$ (sub)diagram with a datastructure dependent n as a refinement, and decomposes it, using the modularisation theorem recursively into smaller (subsub)diagrams. This aproach seems natural here, since the coupling invariant $WINV_{56}$ for two intermediate states during the execution of such a diagram can be defined just by generalizing the case from the coupling invariant INV_{56}, in which both ASMs are directly at a clause: For 5/6 the requirement that $is\_clause(code(preg,db_5)) \wedge is\_clause(code(preg',db_6))$ is generalied to the the weaker requirement, that the instruction sequences currently executed lead to the same clause. The weaker invariant $WINV_{56}$ for subdiagrams now holds in *all* intermediate states. It decomposes the diagrams shown in Fig. 15.2 in 1:0 and 0:1 subdiagrams.

Pairs of states which correspond according to $WINV_{56}$ are connected by dashed lines. *call1* and *call2* denote the first resp. second case of the call rule. The suffix "(a)" denotes the subcase of backtracking, where $breg = \bot$, in which the ASM therefore finishes its computation with result *failure*. The suffixes "(A)" and "(B)" divide the successful case of call rule into the subcase, where only one clause is tried and into the subcases, where several clauses are to explore (in the latter case the subsequent instruction must be a *try_me_else* or a *try*). *ret** denotes an arbitrary number of *retry*, *retry_me*, *trust* or *trust_me* instructions, and *tr** an arbitrary number of *try* or *try_me* instructions. The resulting subdiagrams of the recursive application of the modularisation theorem are shown in Fig. 15.3.

Compared to an immediate decomposion of the whole proof in the smaller subdiagrams the aproach has the advantage that proofs are more modular, and coupling invariants are somewhat smaller. These advantages should in general be compared to the necessity to define *two* coupling invariants INV_{56} and $WINV_{56}$ simultaneously. The disadavantage is not too much of a problem here, since the relation

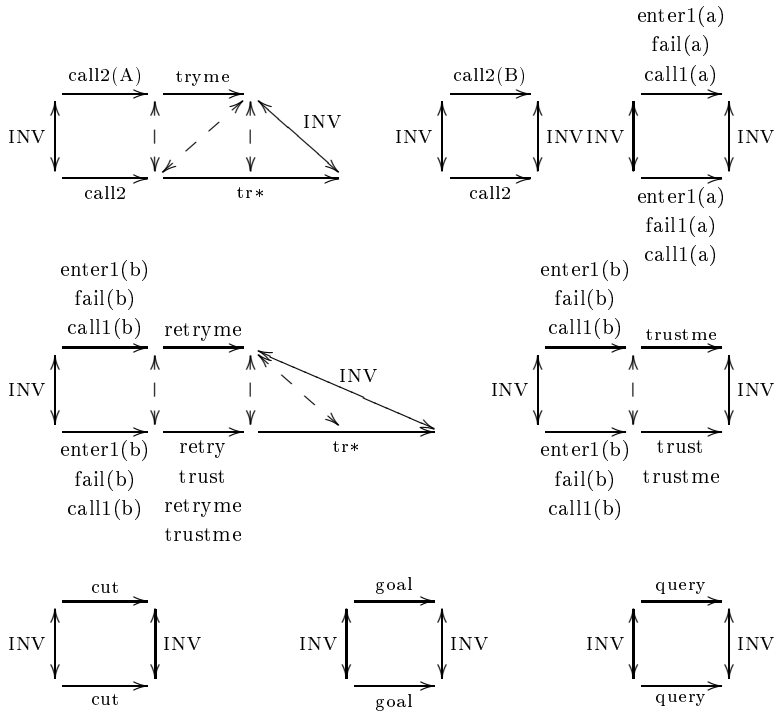


Figure 15.2 : Commuting Diagrams for the Refinement 5/6

$$
\begin{array}{l}
\quad \text{preg} \neq \text{start} \\
\rightarrow (\quad \text{INV}_{56} \\
\qquad \leftrightarrow \quad \text{WINV}_{56} \wedge \text{is\_clause}(\text{code}(\text{preg},\text{db}_5)) \\
\qquad\quad \wedge \text{is\_clause}(\text{code}(\text{preg}',\text{db}_6)))
\end{array}
\tag{15.7}
$$

must hold, which given $WINV_{56}$ is sufficient to construct INV_{56} for the case where *preg* $\neq$ *start* (the case *preg* = *start* is relatively easy). The refinement could be verified in 2 weeks and with 8 iterations. The generalisation of INV_{56} to $WINV_{56}$ was no real problem. The following two coupling invariants were used:

$\text{HINV}_{56} \equiv$
$\exists$ h. $\quad \bot \in \text{s} \wedge \bot \in \text{s}' \wedge \text{h}[\bot] = [\bot] \wedge \text{ctreg} \in \text{s} \wedge \text{ctreg}' \in \text{s}'$
$\wedge$ stop = stop$'$ $\wedge$ vireg = vireg$'$ $\wedge$ (h[breg] $\neq$ [] $\rightarrow$ car(h[breg]) = breg$'$)
$\wedge$ ($\neg$ (is_retry_me(code(preg$'$, db$_6$)) $\vee$ is_retry(code(preg$'$, db$_6$))
$\vee$ is_trust_me(code(preg$'$, db$_6$)) $\vee$ is_trust(code(preg$'$, db$_6$)))
$\rightarrow$ ctreg$'$ = car(h[ctreg]))
$\wedge$ subreg = subreg$'$ $\wedge$ hdg(h, decglseqreg) = decglseqreg$'$
$\wedge$ (preg = start $\leftrightarrow$ preg$'$ = start)
$\wedge$ (decglseqreg = [] $\vee$ goal = [] $\vee$ act = ! $\vee$ act = true $\rightarrow$ preg = start)
$\wedge$ (is_clause(code(preg, db$_5$)) $\wedge$ ctreg $\neq$ breg
$\rightarrow$ ctreg = b[breg] $\wedge$ breg $\neq \bot$
$\wedge$ decglseq[breg] = decglseqreg $\wedge$ sub[breg] = subreg)
$\wedge$ (preg $\neq$ start
$\rightarrow$ is_clause(code(preg, db$_5$)) $\wedge$ is_clause(code(preg$'$, db$_6$))
$\wedge$ code(preg, db$_5$) = code(preg$'$, db$_6$)) $\wedge$ ctreg$'$ = car(h[ctreg])
$\wedge \langle$STACK#(breg, b; stack)$\rangle$
(stack $\subseteq$ s $\wedge$ decglseqreg cutptsin stack

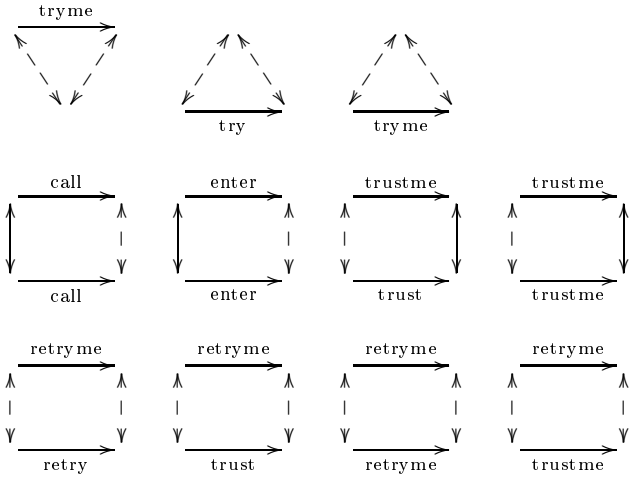


Figure 15.3 : Subdiagrams for the Refinement 5/6

$$
\begin{array}{l}
\quad \wedge\ \langle \text{STACK\#}(\text{breg}', \text{b}'; \text{stack}')\rangle \\
\qquad\quad (\text{stack}' = \text{hl}(\text{h}, \text{stack}) \wedge \text{stack}' \subseteq \text{s}') \\
\quad \wedge\ (\forall\ \text{n}.\quad \text{n} \in \text{stack} \\
\qquad\quad \rightarrow\quad \text{decglseq}[\text{n}] \neq [] \wedge \text{goal}[\text{n}] \neq [] \\
\qquad\qquad \wedge\ \text{is\_user\_defined}(\text{act}[\text{n}]) \wedge \text{h}[\text{n}] \neq [] \\
\qquad\qquad \wedge\ \text{decglseq}[\text{n}]\ \text{cutptsin}\ \text{cdr}(\text{stack from n}) \\
\qquad\qquad \wedge\ (\forall\ \text{n}_0.\quad \text{n}_0 \in \text{h}[\text{n}] \\
\qquad\qquad\qquad\quad \rightarrow\quad \text{sub}[\text{n}] = \text{sub}'[\text{n}_0] \\
\qquad\qquad\qquad\qquad \wedge\ \text{hdg}(\text{h}, \text{decglseq}[\text{n}]) = \text{decglseq}'[\text{n}_0] \\
\qquad\qquad\qquad\qquad \wedge\ \text{ct}[\text{n}_0] = \text{car}(\text{h}[\text{b}[\text{n}]])) \\
\qquad\qquad \wedge\ \langle \text{L-CHAIN-RETRY-ME\#}(\text{p}[\text{n}], \text{db}_5; \text{col})\rangle \\
\qquad\qquad\qquad \langle \text{APP-CHAINS-RET\#}(\text{p}', \text{h}[\text{n}], \text{db}_6; \text{col}_2)\rangle \\
\qquad\qquad\qquad\quad \text{mapcode}(\text{col}, \text{db}_5) = \text{mapcode}(\text{col}_2, \text{db}_6))) \\
\quad \wedge\ \text{STACKINV}_{56}(\text{true})
\end{array}
$$

$$
\begin{array}{l}
\text{WINV}_{56} \equiv \\
\exists\ \text{h}.\quad \bot \in \text{s} \wedge \bot \in \text{s}' \wedge \text{h}[\bot] = \bot +_{sl} [] \wedge \text{ctreg} \in \text{s} \wedge \text{ctreg}' \in \text{s}' \\
\quad \wedge\ \text{stop} = \text{run} \wedge \text{stop} = \text{stop}' \wedge \text{vireg} = \text{vireg}' \\
\quad \wedge\ (\text{h}[\text{breg}] \neq [] \rightarrow \text{car}(\text{h}[\text{breg}]) = \text{breg}') \\
\quad \wedge\ (\quad \neg\ (\quad \text{is\_retry\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_retry}(\text{code}(\text{preg}', \text{db}_6)) \\
\qquad\quad \vee\ \text{is\_trust\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_trust}(\text{code}(\text{preg}', \text{db}_6))) \\
\qquad \rightarrow\ \text{ctreg}' = \text{car}(\text{h}[\text{ctreg}])) \\
\quad \wedge\ \text{subreg} = \text{subreg}' \wedge \text{hdg}(\text{h}, \text{decglseqreg}) = \text{decglseqreg}' \\
\quad \wedge\ \text{preg} \neq \text{start} \wedge \text{preg}' \neq \text{start} \\
\quad \wedge\ \text{decglseqreg} \neq [] \wedge \text{goal} \neq [] \wedge \text{act} \neq\ ! \wedge \text{act} \neq \text{true} \\
\quad \wedge\ (\text{is\_try\_me}(\text{code}(\text{preg}, \text{db}_5)) \rightarrow \text{is\_user\_defined}(\text{act}) \wedge \text{ctreg} = \text{breg}) \\
\quad \wedge\ (\quad \text{is\_clause}(\text{code}(\text{preg}, \text{db}_5)) \wedge \text{ctreg} \neq \text{breg} \\
\qquad \rightarrow\quad \text{ctreg} = \text{b}[\text{breg}] \wedge \text{breg} \neq \bot \\
\qquad\quad \wedge\ \text{decglseq}[\text{breg}] = \text{decglseqreg} \wedge \text{sub}[\text{breg}] = \text{subreg}) \\
\quad \wedge\ (\text{is\_clause}(\text{code}(\text{preg}, \text{db}_5)) \rightarrow \text{ctreg}' = \text{car}(\text{h}[\text{ctreg}])) \\
\quad \wedge\ (\quad \text{is\_try\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_try}(\text{code}(\text{preg}', \text{db}_6)) \\
\qquad \rightarrow\ \text{is\_user\_defined}(\text{act}')) \\
\quad \wedge\ (\quad \text{is\_retry\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_retry}(\text{code}(\text{preg}', \text{db}_6)) \\
\qquad\quad \vee\ \text{is\_trust\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_trust}(\text{code}(\text{preg}', \text{db}_6))
\end{array}
$$

$$
\begin{array}{l}
\quad \to \quad \mathrm{breg}' \neq \bot \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{breg}'] \wedge \mathrm{ctreg}' = \mathrm{car}(\mathrm{h}[\mathrm{b}[\mathrm{breg}]]) \\
\quad\quad \wedge (\mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)))) \\
\wedge (\quad \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \\
\quad \to \quad (\quad \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\quad\quad \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\quad\quad \wedge \mathrm{breg} \neq \bot \wedge \mathrm{preg} = \mathrm{p}[\mathrm{breg}] \\
\quad\quad \wedge \langle \text{L-CHAIN-RETRY-ME\#}(\mathrm{preg}, \mathrm{db}_5; \mathrm{col})\rangle \\
\quad\quad\quad \langle \text{APP-CHAINS-RET\#}(\mathrm{p}', \mathrm{h}[\mathrm{breg}], \mathrm{db}_6; \mathrm{col}_2)\rangle \\
\quad\quad\quad\quad \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_6)) \\
\wedge (\quad \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \\
\quad \to \quad (\mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\quad\quad \wedge \langle \text{L-CHAIN-TRY-ME\#}(\mathrm{preg}, \mathrm{db}_5; \mathrm{col})\rangle \\
\quad\quad\quad \langle \text{CHAIN-REC\#}(\mathrm{preg}', \mathrm{db}_6; \mathrm{col}_1)\rangle \\
\quad\quad\quad\quad \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_1, \mathrm{db}_6)) \\
\wedge (\mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \wedge \neg\, \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\quad \to \quad (\mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\quad\quad \wedge \mathrm{breg} \neq \bot \wedge \mathrm{decglseqreg} = \mathrm{decglseq}[\mathrm{breg}] \\
\quad\quad \wedge \mathrm{subreg} = \mathrm{sub}[\mathrm{breg}] \wedge \mathrm{ctreg} = \mathrm{b}[\mathrm{breg}] \\
\quad\quad \wedge \langle \text{L-CHAIN-RETRY-ME\#}(\mathrm{p}[\mathrm{breg}], \mathrm{db}_5; \mathrm{col})\rangle \\
\quad\quad\quad \langle \text{CHAIN-REC\#}(\mathrm{preg}', \mathrm{db}_6; \mathrm{col}_1)\rangle \\
\quad\quad\quad\quad \langle \text{APP-CHAINS-RET\#}(\mathrm{p}', \mathrm{h}[\mathrm{breg}], \mathrm{db}_6; \mathrm{col}_2)\rangle \\
\quad\quad\quad\quad\quad \mathrm{the\_cl}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) +_{cli} \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) \\
\quad\quad\quad\quad\quad = \mathrm{mapcode}(\mathrm{col}_1 \odot_{col} \mathrm{col}_2, \mathrm{db}_6)) \\
\wedge (\quad \mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\quad \to \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5))) \\
\wedge (\mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \to \mathrm{code}(\mathrm{preg}, \mathrm{db}_5) = \mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\wedge (\quad \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \\
\quad \vee \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5))) \\
\wedge \mathrm{STACKINV}_{56}(\neg\, \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)))
\end{array}
$$

$$
\begin{array}{l}
\mathrm{STACKINV}_{56} \equiv \\
\langle \text{STACK\#}(\mathrm{breg}, \mathrm{b}; \mathrm{stack})\rangle \\
\quad (\quad \mathrm{stack} \subseteq \mathrm{s} \wedge (\mathrm{cond} \to \mathrm{decglseqreg\ cutptsin\ stack}) \\
\quad \wedge \langle \text{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}')\rangle \\
\quad\quad (\mathrm{stack}' = \mathrm{hl}(\mathrm{h}, \mathrm{stack}) \wedge \mathrm{stack}' \subseteq \mathrm{s}') \\
\quad \wedge (\forall\, \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\quad\quad \to \quad \mathrm{decglseq}[\mathrm{n}] \neq [] \wedge \mathrm{goal}[\mathrm{n}] \neq [] \wedge \mathrm{is\_user\_defined}(\mathrm{act}[\mathrm{n}]) \\
\quad\quad\quad \wedge \mathrm{decglseq}[\mathrm{n}]\ \mathrm{cutptsin}\ \mathrm{cdr}(\mathrm{stack\ from\ n}) \\
\quad\quad\quad \wedge (\forall\, \mathrm{n}_0. \quad \mathrm{n}_0 \in \mathrm{h}[\mathrm{n}] \\
\quad\quad\quad\quad \to \quad \mathrm{sub}[\mathrm{n}] = \mathrm{sub}'[\mathrm{n}_0] \wedge \mathrm{ct}[\mathrm{n}_0] = \mathrm{car}(\mathrm{h}[\mathrm{b}[\mathrm{n}]]) \\
\quad\quad\quad\quad\quad \wedge \mathrm{hdg}(\mathrm{h}, \mathrm{decglseq}[\mathrm{n}]) = \mathrm{decglseq}'[\mathrm{n}_0] \\
\quad\quad\quad \wedge (\quad \mathrm{n} \neq \mathrm{breg} \\
\quad\quad\quad\quad \vee \quad \neg\, \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\quad\quad\quad\quad\quad \wedge \neg\, \mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\quad\quad\quad\quad\quad \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \\
\quad\quad\quad\quad \to \quad \mathrm{h}[\mathrm{n}] \neq [] \\
\quad\quad\quad\quad\quad \wedge \langle \text{CHAIN-RETRY-ME-FL\#}(\mathrm{p}[\mathrm{n}], \mathrm{db}_5; \mathrm{col})\rangle \\
\quad\quad\quad\quad\quad\quad \langle \text{APP-CHAINS-RET\#}(\mathrm{p}', \mathrm{h}[\mathrm{n}], \mathrm{db}_6; \mathrm{col}_2)\rangle \\
\quad\quad\quad\quad\quad\quad\quad \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_6))))
\end{array}
$$

Auxiliary Theorems for the ASMs If one analyzes the equivalence proof 5/6 it becomes obvious, that in the proofs of 0:1 diagrams a lot of properties of ASM5 are shown to be invariant in ASM6, that are encoded only implicitly via the correspondence to ASM6. An alternative is, to

prove auxiliary theorems that are concerned with the execution of chains in ASM6 alone.

We have worked out a proof for the refinement 5/7 first using the technique of recursive decomposition of diagrams. We found, that the generalization of INV_{57} to $WINV_{57}$ is a *very* hard problem: The final $WINV_{57}$ has 4 times the size of $WINV_{56}$. To find the correct version and to verify 5/7 took 2 months and 20 iterations. Therefore we have tried the technique of auxiliary theorems too. It lead to much smaller proofs, as can be seen from the statistics at the end of this section. For complex refinements we therefore prefer this technique although it adds to the problem of finding a suitable coupling invariant the problem to find suitable auxiliary theorems, which are not only provable but als fit into the overall proof.

As auxiliary theorems for ASM7 we first formulated, that execution of some arbitrary chain leads to one of the following results:

- If the chain is empty and *breg* $= \bot$, the run of ASM7 is terminated with *stop* = *failure*.
- If the chain is empty and *breg* $\neq \bot$, then ASM7 will reach a state, in which the instructions of the chain have been completely executed, and the chain has just been left by backtracking, i.e. *decglseqreg, subreg, ctreg, vireg* and the stack are still unchanged and *preg* points to the topmost stack element *p*[*breg*].
- If the chain is nonempty, then a state is reached, in which the first clause has been reached, i.e. *decglseqreg, subreg, ctreg, vireg* are unchanged, *preg* points to the first clause of the chain. A number of choicepoints have been pushed on the stack, which all contain *decglseqreg, subreg* and *ctreg*, and whose chains contain appended exactly the clauses of the original chain except the first.

As a formula this can be written as Lemma *chain7*:

$$
\begin{array}{l}
\quad \text{decglseq}' = \text{decglseq}'_0 \wedge \text{sub}' = \text{sub}'_0 \wedge \text{ct} = \text{ct}_0 \wedge \text{p}' = \text{p}'_0 \\
\wedge\ \text{b}' = \text{b}'_0 \wedge \text{vireg}' = \text{vireg}'_0 \wedge \text{stop}' = \text{run} \wedge \text{s}'_0 \subseteq \text{s}' \wedge \bot \in \text{s}'_0 \\
\wedge\ \text{decglseqreg}'_0 \neq [] \wedge \text{goal}'_0 \neq [] \wedge \text{is\_user\_defined}(\text{act}'_0) \\
\wedge\ \langle \text{STACK\#}(\text{breg}', \text{b}'; \text{stack}')\rangle\ \text{stack}' = \text{stack} \wedge \text{stack} \subseteq \text{s}' \\
\wedge\ (\quad \text{is\_retry}(\text{code}(\text{preg}', \text{db}_7)) \vee \text{is\_retry\_me}(\text{code}(\text{preg}', \text{db}_7)) \\
\quad \vee \text{is\_trust}(\text{code}(\text{preg}', \text{db}_7)) \vee \text{is\_trust\_me}(\text{code}(\text{preg}', \text{db}_7)) \\
\quad \supset \quad \text{stack} \neq [] \wedge \text{preg}' = \text{p}'[\text{car}(\text{stack})] \wedge \text{decglseqreg}' \neq [] \\
\quad \wedge\ \text{goal}' \neq [] \wedge \text{decglseqreg}'_0 = \text{decglseq}'[\text{car}(\text{stack})] \\
\quad \wedge\ \text{subreg}'_0 = \text{sub}'[\text{car}(\text{stack})] \wedge \text{ctreg}'_0 = \text{ct}[\text{car}(\text{stack})] \\
\quad \wedge\ \text{stack}_0 = \text{cdr}(\text{stack})\ ; \\
\quad\quad \text{subreg}'_0 = \text{subreg}' \wedge \text{decglseqreg}'_0 = \text{decglseqreg}' \\
\quad \wedge\ \text{ctreg}'_0 = \text{ctreg}' \wedge \text{stack}_0 = \text{stack}) \\
\wedge\ \langle \text{S-ANY-CHAIN\#}(\text{act}'_0, \text{preg}', \text{db}_7; \text{col})\rangle \\
\quad\quad \text{col} = \text{col}_0 \\
\rightarrow\ \exists\ \text{kappa}. \\
\quad\quad \langle \textbf{loop} \\
\quad\quad\quad \textbf{if}\ \text{stop}' = \text{run}\ \textbf{then} \\
\quad\quad\quad\ \text{RULE}'(\text{mkco3res}(\text{db}_7, \text{procdeftab}); \text{s}', \text{vireg}', \text{stop}', \text{breg}', \\
\quad\quad\quad\quad\quad \text{ctreg}', \text{sub}', \text{subreg}', \text{decglseq}', \text{decglseqreg}', \text{p}', \\
\quad\quad\quad\quad\quad \text{preg}', \text{b}', \text{ct}) \\
\quad\quad\quad \textbf{times}\ \text{kappa}\rangle \\
\quad\quad\quad (\quad \text{col}_0 = [] \\
\quad\quad\quad\ \supset \text{stack}_0 = [] \supset \text{stop}' = \text{failure} \wedge \text{breg}' = \bot\ ; \\
\quad\quad\quad\quad\quad \text{preg}' = \text{p}'[\text{car}(\text{stack}_0)] \wedge \text{decglseqreg}' = \text{decglseqreg}'_0 \\
\quad\quad\quad\quad \wedge\ \text{subreg}' = \text{subreg}'_0 \wedge \text{ctreg}' = \text{ctreg}'_0 \\
\quad\quad\quad\quad \wedge\ \text{vireg}' = \text{vireg}'_0 \wedge \text{s}'_0 \subseteq \text{s}' \wedge \text{stop}' = \text{run} \\
\quad\quad\quad\quad \wedge\ \langle \text{STACK\#}(\text{breg}', \text{b}'; \text{stack})\rangle \\
\quad\quad\quad\quad\quad\quad (\quad \text{stack} = \text{stack}_0 \wedge \text{stack} \subseteq \text{s}'
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}]))\ ; \\
\quad \mathrm{decglseqreg}' = \mathrm{decglseqreg}'_0 \wedge \mathrm{subreg}' = \mathrm{subreg}'_0 \\
\wedge\ \mathrm{ctreg}' = \mathrm{ctreg}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \\
\wedge\ \mathrm{stop}' = \mathrm{run} \wedge \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \wedge \mathrm{preg}' = \mathrm{car}(\mathrm{col}_0) \\
\wedge\ (\exists\ \mathrm{nl}.\quad \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\qquad\qquad (\mathrm{stack} = \mathrm{append}(\mathrm{nl}, \mathrm{stack}_0) \wedge \mathrm{stack} \subseteq \mathrm{s}') \\
\qquad \wedge\ \langle \text{S-APP-CHAINS-RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}, \mathrm{db}_7; \\
\qquad\qquad \mathrm{col})\rangle\ \mathrm{col} = \mathrm{cdr}(\mathrm{col}_0) \\
\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl} \\
\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseqreg}'_0 \\
\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{subreg}'_0 \wedge \mathrm{ct}[\mathrm{n}] = \mathrm{ctreg}'_0) \\
\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack}_0 \\
\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\quad \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}])))
\end{array}
$$

The proof is by induction on the number of instructions in the chain. Using the lemma it can be proved, that if ASM7 does backtracking and the stack contains a number of empty choicepoints at its top, then a state is reached where all empty choicepoints have been removed. Formally this is lemma *emptychains7*:

$$
\begin{array}{l}
\qquad \mathrm{decglseq}' = \mathrm{decglseq}'_0 \wedge \mathrm{sub}' = \mathrm{sub}'_0 \wedge \mathrm{ct} = \mathrm{ct}_0 \wedge \mathrm{p}' = \mathrm{p}'_0 \\
\wedge\ \mathrm{b}' = \mathrm{b}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \bot \in \mathrm{s}'_0 \\
\wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \\
\wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}')\rangle\ \mathrm{stack}' = \mathrm{stack} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\wedge\ (\quad \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \\
\quad \vee\ \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7))) \\
\wedge\ \mathrm{stack} = \mathrm{append}(\mathrm{nl},\mathrm{stack}_0) \wedge \mathrm{stack} \neq [] \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack})] \\
\wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl} \\
\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}'[\mathrm{n}] \neq [] \\
\qquad\quad \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{n}])) \\
\wedge\ \langle \text{S-APP-CHAINS-RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}, \mathrm{db}_7; \mathrm{col})\rangle\ \mathrm{col} = [] \\
\rightarrow\ \exists\ \mathrm{kappa}.\ \langle \textbf{loop} \\
\qquad\qquad \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then} \\
\qquad\qquad\ \mathrm{RULE}'(\mathrm{mkco3res}(\mathrm{db}_7, \mathrm{procdeftab}); \mathrm{s}', \mathrm{vireg}', \mathrm{stop}', \mathrm{breg}', \\
\qquad\qquad\qquad \mathrm{ctreg}', \mathrm{sub}', \mathrm{subreg}', \mathrm{decglseq}', \mathrm{decglseqreg}', \mathrm{p}', \\
\qquad\qquad\qquad \mathrm{preg}', \mathrm{b}', \mathrm{ct}) \\
\qquad\qquad \textbf{times}\ \mathrm{kappa}\rangle \\
\qquad\qquad (\mathrm{stack}_0 = [] \supset \mathrm{stop}' = \mathrm{failure} \wedge \mathrm{breg}' = \bot\ ; \\
\qquad\qquad\quad \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack}_0)] \\
\qquad\qquad\quad \wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \\
\qquad\qquad\quad \wedge\ \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \mathrm{stop}' = \mathrm{run} \\
\qquad\qquad\quad \wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\qquad\qquad\qquad (\quad \mathrm{stack} = \mathrm{stack}_0 \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\qquad\qquad\qquad\ \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad\quad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}])))
\end{array}
$$

Finally we need a lemma which combines *chain7* and *emptychains7*, called *nextclause7*, which states that backtracking in a stack of choicepoints leads to the first nonempty choicepoint, and that its chain is reduced to a clause and new choicepoints:

$$
\begin{array}{l}
\quad \mathrm{decglseq}' = \mathrm{decglseq}'_0 \wedge \mathrm{sub}' = \mathrm{sub}'_0 \wedge \mathrm{ct} = \mathrm{ct}_0 \wedge \mathrm{p}' = \mathrm{p}'_0 \\
\wedge\ \mathrm{b}' = \mathrm{b}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \bot \in \mathrm{s}'_0 \\
\wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \\
\wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}')\rangle\ \mathrm{stack}' = \mathrm{stack} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\wedge\ (\quad \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \\
\quad \vee\ \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7))) \\
\wedge\ \mathrm{stack} = \mathrm{append}(\mathrm{nl}, \mathrm{stack}_0) \wedge \mathrm{stack}_0 \neq [] \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack})] \\
\wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl} \\
\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}'[\mathrm{n}] \neq [] \wedge \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{n}])) \\
\wedge\ \langle \mathrm{S\text{-}APP\text{-}CHAINS\text{-}RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}, \mathrm{db}_7; \mathrm{col})\rangle\ \mathrm{col} = [] \\
\wedge\ \langle \mathrm{S\text{-}CHAIN\text{-}RET\#}(\mathrm{act}'[\mathrm{car}(\mathrm{stack}_0)], \mathrm{p}'[\mathrm{car}(\mathrm{stack}_0)], \mathrm{db}_7; \mathrm{col})\rangle \\
\qquad \mathrm{col} = \mathrm{col}_0 \\
\wedge\ \mathrm{col}_0 \neq [] \wedge \mathrm{decglseq}'[\mathrm{car}(\mathrm{stack}_0)] \neq [] \\
\wedge\ \mathrm{goal}'[\mathrm{car}(\mathrm{stack}_0)] \neq [] \\
\wedge\ \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{scar}(\mathrm{stack}_0)]) \\
\rightarrow \exists\ \mathrm{kappa}.\ \langle \textbf{loop} \\
\qquad\qquad \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then} \\
\qquad\qquad \mathrm{RULE}'(\mathrm{mkco3res}(\mathrm{db}_7, \mathrm{procdeftab}); \mathrm{s}', \mathrm{vireg}', \mathrm{stop}', \mathrm{breg}', \\
\qquad\qquad\qquad \mathrm{ctreg}', \mathrm{sub}', \mathrm{subreg}', \mathrm{decglseq}', \mathrm{decglseqreg}', \mathrm{p}', \\
\qquad\qquad\qquad \mathrm{preg}', \mathrm{b}', \mathrm{ct}) \\
\qquad\qquad \textbf{times}\ \mathrm{kappa}\rangle \\
\qquad\qquad (\quad \mathrm{decglseqreg}' = \mathrm{decglseq}'_0[\mathrm{car}(\mathrm{stack}_0)] \\
\qquad\qquad \wedge\ \mathrm{subreg}' = \mathrm{sub}'_0[\mathrm{car}(\mathrm{stack}_0)] \wedge \mathrm{ctreg}' = \mathrm{ct}_0[\mathrm{car}(\mathrm{stack}_0)] \\
\qquad\qquad \wedge\ \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \mathrm{stop}' = \mathrm{run} \\
\qquad\qquad \wedge\ \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \wedge \mathrm{preg}' = \mathrm{car}(\mathrm{col}_0) \\
\qquad\qquad \wedge\ (\exists\ \mathrm{nl}_1.\quad \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\qquad\qquad\qquad\qquad (\quad \mathrm{stack} = \mathrm{append}(\mathrm{nl}_1, \mathrm{cdr}(\mathrm{stack}_0)) \\
\qquad\qquad\qquad\qquad \wedge\ \mathrm{stack} \subseteq \mathrm{s}') \\
\qquad\qquad\qquad \wedge\ \langle \mathrm{S\text{-}APP\text{-}CHAINS\text{-}RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}_1, \mathrm{db}_7; \\
\qquad\qquad\qquad\qquad \mathrm{col})\rangle\ \mathrm{col} = \mathrm{cdr}(\mathrm{col}_0) \\
\qquad\qquad\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl}_1 \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseqreg}' \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{subreg}' \wedge \mathrm{ct}[\mathrm{n}] = \mathrm{ctreg}') \\
\qquad\qquad\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{cdr}(\mathrm{stack}_0) \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}])))
\end{array}
$$

With these lemmas we can then decompose the commuting diagrams of 5/7 as shown in Fig. 15.4.

CINV is the case in the coupling invariant in which *preg* = *start* holds, *EINV* is the case where the next instruction is a clause. In the case *FINV* both ASMs have finished their run. The most complicated proof is the one, in which backtracking is called (the 7 diagrams in the lower half of Fig. 15.4). The figure hints, that the proofs of the first 5 diagrams can be merged into one. It is sufficient to use the coupling invariant as precondition, and to replace the two calls to rules of ASM5 and ASM7 by calls to the corresponding backtrack program. The last two of the 7 diagrams can be reduced to the proof of the diagram directly above them, by applying lemma *chain7* first (to remove the empty chain in ASM7).

The total effort for the verification of 5/7 by recursive decomposition of diagrams was 17009 proof steps and 1521 interactions. The proof using auxiliary lemmas was done within a week and required only 7473 proof steps and 1351 interactions.

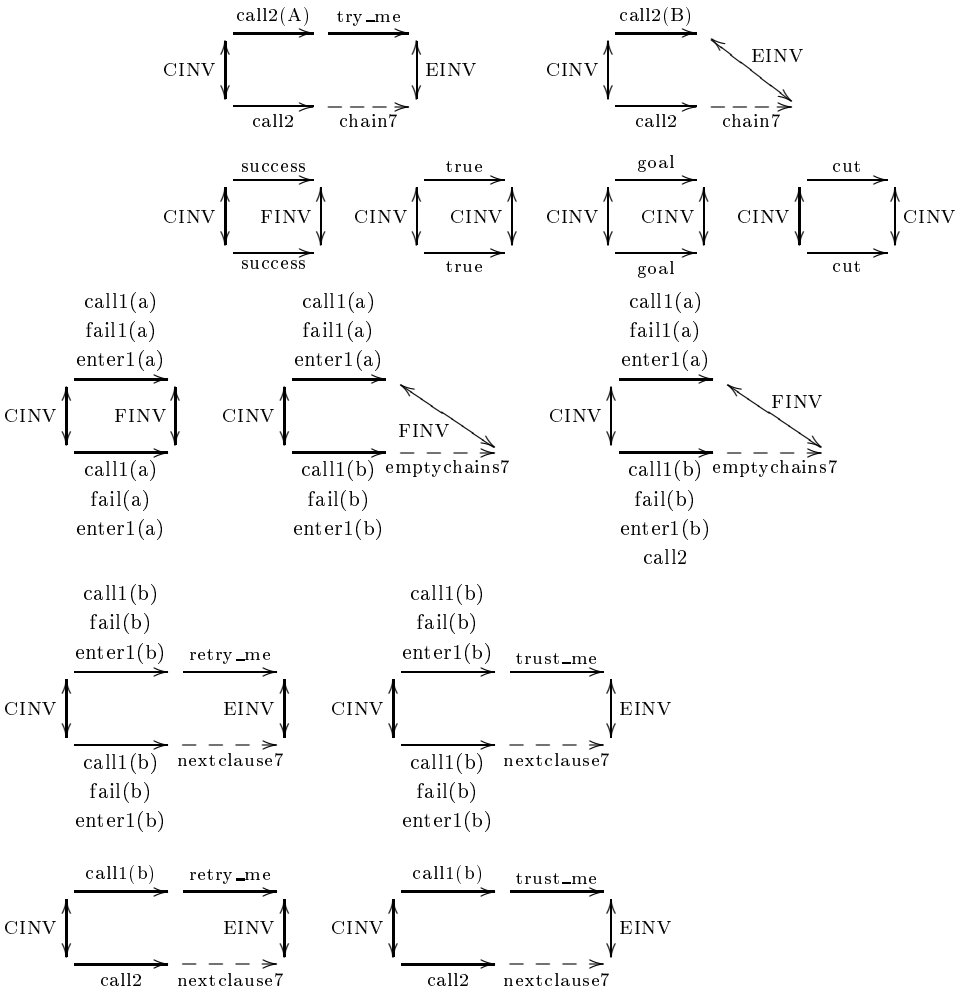


Figure 15.4 : Commuting Diagrams for the Refinement 5/7

Chapter 16

7/8: Environments and Stack Sharing

16.1 Definition of ASM8

After we have completed the compilation of predicate structure with ASM7, refinement 7/8 now prepares the compilation of single clauses. A first step in this direction is to transform the data-structure of *decglseq*'s, such that the goals contained in them are directly accessible and can later on be replaced with pointers into the code of clause bodies. To make this possible, it is necessary to delay the application of substitutions to goals. Instead substitutions are applied to literals when the literal becomes a new activator. With this approach all goals become end pieces of clause bodies. Although goals still contain renamed variables and can therefore not be replace by pointers to code immediately (this will be changed in the refinement 8/9, when the clauses are compiled), dispensing with the immediate application of substitutions in *enter rule* causes old and new *decglseqreg* to have a large common part. By restructuring, the information contained in the common part can now be shared and stored only once.

Sharing is achieved as follows: Instead of storing [$\langle goal_1,\ ctpt_1\rangle$, $\langle goal_2, ctpt_2\rangle$, $\langle goal_3, ctpt_3\rangle$, ...] in *decglseqreg*, $goal_1$ is accessible in ASM8 in a new register *goalreg* directly. For the rest of the informations an *environment* is allocated. Formally an environment is an element of a dynamic sort *envnode*, similar to a choicepoint, that is stored in a register *ereg* (again, similar to *breg*). Dynamic functions *cutpt* and *cg* attach the current cutpoint and the second goal (the "continuation goal") to the environment: $cutpt[ereg] = ctpt_1$ and $cg[ereg] = goal_2$. The rest of the information ($ctpt_2$, $goal_3$, etc.) can be reached via a function $ce : envnode \rightarrow envnode$ (the "continuation environment").

With the re-encoding of the information stored in *decglseqreg* a similar re-encoding for the data stored in $decglseqreg[n]$ for each choicepoint n becomes necessary. Instead of $decglseqreg[n]$ ASM8 used two new functions $goal[n]$ and $e[n]$ for this purpose, which correspond to *goalreg* and *ereg*.

Changing the representation of the data in the *decglseq*'s rises the question, whether environments have to be put on a separate (environment) stack. This is not the case, it is possible to store environments and choicepoints on the same stack, and to introduce a genuine stack discipline, that overwrites abandoned stack frames destructively. By that, sort *envnode* becomes equal to sort *node*.

In [BR95] the new stack discipline is introduced in two steps: First, ASM8 contains a common, but not destructively modified stack, and ASM9 then replaces allocation of new stack nodes with overwriting. This two-step approach seemed disadvantageous for verification to us, since the intermediate level requires to introduce an additional dynamic function *tos*, which has to return the maximum of two nodes relative to a dynamic stack chaining function $-$ (see p. 32 in [BR95]). The definition of such a function is possible, but elaborate. It would be only needed in ASM8, and can be avoided by going directly to the stack representation of ASM9. Our solution therefore does

not divide introduction of a destructively modified stack over two refinements, but includes it in refinement 7/8. The *Hiding Lemma* thereby is needed only in the verification of this refinement.

To introduce the destructively modified stack, we add a total order $\ll$ on stack nodes, and define functions $+1$ and -1 to increment and decrement them. Thereby, the role of stack nodes becomes one of addresses. Allocation of stack nodes is no longer done with the function *new* relative to a set of allocated nodes, but simply by incrementing the pointer to the top element of the stack. To make an environment or a choicepoint inaccessible, we now simply decrement the pointer to the topmost stack frame. Allocation of a new stack frame will then overwrite the inaccessible one. Abandoned nodes, which have been allocated but are not in the current stack are no longer possible in ASM8. The statement of the *Hiding Lemma* is now, that when new nodes (environment nodes as well as choicepoint nodes) are always allocated at $max(breg,ereg)+1$, then the environment nodes $e[n]$, $ce[e[n]]$, ... belonging to a choicepoint n will always be below n (so the choicepoints "'hides"' them from being overwritten). The same will also hold for the choicepoints $cutpt[n']$, $cutpt[b[n']]$ stored in an environment or a choicepoint n'. For ASM8 we have the following rules:

backtrack $\equiv$
    if breg $= \bot$ **then** stop := failure
        else preg := p[breg]

call rule
    let act = subreg $\hat{}_t$ car(goalreg)
    if preg = start $\wedge$ is_user_defined(act)
    then if procdef_7(act,db_7) = failcode
        then backtrack
        else preg := procdef_7(act,db_7)
            ctreg := breg

cut rule
    let act = subreg $\hat{}_t$ car(goalreg)
    if act = !
    then breg := cutpt[ereg]
        goalreg := rest(goalreg)

enter rule
    if is_clause(code(preg, db_7))
    then let cla = rename(clause(code(preg, db_7)), vireg)
        let act = subreg $\hat{}_t$ car(goalreg)
        let mgu = unify(act, hd(cla))
        if mgu = nil
        then backtrack
        else let tmp = max(ereg,breg)+1
            ce[tmp] := ereg
            ereg := tmp
            cg[tmp] := rest(goalreg)
            cutpt[tmp] := ctreg
            goalreg := bdy(cla)
            subreg := subreg $\circ$ mgu
            vireg := vireg +1
            preg := start

fail rule
    let act = subreg $\hat{}_t$ car(goalreg)
    if act = fail
    then backtrack

goal success rule
if goalreg = [] ∧ ¬ ereg = ⊥
then goalreg := cg[ereg]
ereg := ce[ereg]

query success rule
if goalreg = [] ∧ ereg = ⊥
then stop := success

retry rule
if code(preg,db_7) = retry(N)
then ereg := e[breg]
goalreg[breg] := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := preg +1
preg := N

retry_me_else rule
if code(preg,db_7) = retry_me_else(N)
then ereg := e[breg]
goalreg := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := N
preg := preg +1

switch_on_constant rule
let act = subreg ^$_t$ car(goalreg)
if code(preg, db_7) = switch_on_constant(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashc(table, tabsize, constsym(x_i), db_7);
if preg = failcode **then backtrack**

switch_on_structure rule
let act = subreg ^$_t$ car(goalreg)
if code(preg, db_7) = switch_on_structure(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashs(table, tabsize, funct(x_i), arity(x_i), db_7);
if preg = failcode **then backtrack**

switch_on_term rule
let act = subreg ^$_t$ car(goalreg)
if code(preg, db_7) = switch_on_term(i, N_s, N_c, N_v, N_l)
then let x_i = arg(act,i)
if is_struct(x_i) **then** preg := N_s **else**
if is_const(x_i) **then** preg := N_c **else**
if is_var(x_i) **then** preg := N_v **else**
if is_list(x_i) **then** preg := N_l;
if preg = failcode **then backtrack**

true rule
let act = subreg ^$_t$ car(goalreg)
if act = true
then goalreg := rest(goalreg)

trust rule
if code(preg,db$_7$) = trust(N)
then ereg := e[breg]
goalreg := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := N

trust_me rule
if code(preg,db$_7$) = trust_me
then ereg := e[breg]
goalreg := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

try rule
if code(preg,db$_7$) = try(N)
then let tmp = max(ereg,breg) +1
b[tmp] := breg
e[tmp] := ereg
goal[tmp] := goalreg
sub[tmp] := subreg
p[tmp] := preg +1
breg := tmp
ct[tmp] := ctreg
preg := N

try_me rule
if code(preg,db$_7$) = try_me_else(N)
then let tmp = max(ereg,breg)+1
b[tmp] := breg
e[tmp] := ereg
goal[tmp] := goalreg
sub[tmp] := subreg
p[tmp] := N
breg := tmp
ct[tmp] := ctreg
preg := preg +1

16.2 Equivalence Proof 7/8

Verification of 7/8 poses 3 main problems: first, we must make precise the connection between the *decglseq*'s and the components of ASM8. Here we found, that a modification of the *query success* rule was necessary, to keep the 1:1 correspondence of rules. Second we have to make the correctness of stack sharing explicit in the coupling invariant. Third, delaying substitutions resulted in an additional compiler assumption necessary for the correctness of the refinement.

Correspondence of Environment and *decglseq*'s To verify 7/8 we first have to make precise the initialization of environments, the connection between *decglseq*'s from ASM7 and the components of ASM8, and the termination criterion in ASM8. All three points are tightly connected,

since the initial environment strongly influences the coupling invariant as well as the guard of *query success* rule. The ASM rules that were shown in the previous section already contain the necessary modifications compared to [BR95].

For the initialization we have set *ereg* to $\perp$. The function *ce* as well as *cutpt* have to map $\perp$ to $\perp$. The initialization of *cg* is arbitrary, and *goalreg* has to be initialized with the query. With this initialization we can compute *decglseqreg* and *decglseq*[n] from ASM7, using the components of ASM8:

⟨STACK#(ereg,ce;estack)⟩
  decglseqreg = subreg ^$_d$ [⟨goalreg, cutpt[ereg]⟩ |
                  decglseqof(cutpt, cg, ce, estack)]

⟨STACK#(e[n],ce;estack′)⟩ decglseq[F[n]]
  = sub[F[st]] ^$_d$ F$_d$(F,[⟨goal[n],cutpt[e[n]]⟩ |
                  decglseqof(cutpt, cg, ce, estack′)])

Like in the refinements 1/2, 2/3 etc. the choicepoint of ASM8, that corresponds to a choicepoint *st* of ASM is computed as *F*[*st*] with a dynamic function *F*. *estack* and *estack′* are the environment stacks starting at *ereg* resp. *e*[*n*]. These lists of stack nodes can be computed with the same program *STACK#* (see the definition in Sect. 11.2), that was used for choicepoints. The function *decglseqof* collects the information at the corresponding nodes:

decglseqof(cutpt,cg,ce,[]) = []

decglseqof(cutpt,cg,ce,[n | estack])
  = [⟨cg[n],cutpt[ce[n]]⟩ | decglseqof(cutpt,cg,ce,estack)]

Until now our definitions seem to agree with those given in [BR95]. Only the initialization of *ereg* with $\perp$ was added, the connection between the registers was formalized, and the definition of function *G* (p.32 f), that would have to be realized as a program, was decomposed into calls of *STACK#* and *decglseqof*. But our definition of the termination criterion for *query success* will deviate from [BR95], where the rule test is defined (using our notation) as

goalreg = $\perp$ ∧ ⟨STACK#(ereg,ce;estack)⟩ ∀ n ∈ estack. goal[n] = []

We have deviated, although it *is* correct, to finish the computation when all goals on the stack are empty. Nevertheless the test is very expensive since all *goal*[*n*] must be looked at (and the test has to be done each time an empty goal is reached to decide whether *goal success* or *query success* rule should be applied). Also the optimisation removes all applications of *goal success* rule at the end of a computation, violating the proposed 1:1 correspondence of ASM rules. Also the following ASM9 does not look at several stack frames, so the optimisation is not used in ASM9. Therefore we use

goalreg = $\perp$ ∧ ereg = $\perp$

as the rule test of *query success*. This corresponds to a test *decglseqreg* = [⟨[], *ctpt*⟩] in ASM7. This means that the last applications of *goal success* and *query success* in ASM7 have been replaced by an application of *query success*. Therefore we have a 2:1 diagram for this case. The 2:1 diagram cannot be avoided, since from the connection of *decglseqreg* to the components of ASM8 shown above (which will be part of the coupling invariant) it is clear that there is no possibility to represent a state corresponding to *decglseqreg* = [] in ASM8.

Stack Sharing The most delicate task in setting up the coupling invariant is to make the stack sharing of ASM8 explicit. The coupling invariant must assure, that allocation of choicepoints and environments never overwrites still relevant old ones. To save lengthy calls to the *STACK#* program in the following we denote with *estack* the current stack of environements (a list starting starting with *ereg*), with *bstack* the current stack of choicepoints (starting with *breg*) and with *estack*[*n*] the stack of environments starting with the environemnt $e[n]$ of choicepoint n. Then we need first need the following obvious properties:

- The choicepoint stack *bstack* and the environment stack *estack* are disjoint (formalized as *disjoint*(*estack*,*bstack*)).
- the choicepoint stack *bstack* is also disjoint to the environement stack of every choicepoint.
- The choicepoints in *bstack* are strictly monotone decreasing with respect to $\ll$ (formalised as *ordered*(*bstack*)).
- The environments in *estack* and *estack*[*n*] are decreasing too.
- The environment $e[n]$ of each choicepoint n is below the choicepoint (this is the content of the "Hiding Lemma").

Unfortunately these propertoes are not sufficient for a successful verification. We found, that a number of other properties are necessary, that are not obvious at first. The two most important are.

- *breg* is never below *cutpt*[*ereg*]
- *ct*[*n*] is never above the choicepoint n, and never below *cutpt*[$e[n]$]

Two other simple properties are that no states are below $\perp$, and the *cutptsin* properties we already needed in previous refinements.

Delaying Substitutions Delaying the application of substitutions to goals as far as possible seems to be a harmless transformation at first glance. But if one tries to prove the equivalence of the two *enter* rules of ASM7 and ASM8, then one encounters the problem, that *the substitutions applied to activators of ASM7 and ASM8 are different*! To understand this, look at a situation where an activator is unified with the head of a clause $H :- B$ that has been renamed with *vireg*. Let us assume, that the computed substitutions in *subreg* and *subreg'* as well as both activators *act* and *act'* are equal. Then both ASM7 and ASM8 will compute the same *mgu*. Both will then compute a new goal, consisting of literal B. ASM7 instantiates B immediately with *mgu*, while ASM8 will only compute the new substitution *subreg* $\circ$ *mgu*. When now B becomes itself the activator later on, ASM8 will instantiate it with this composed substitution, and not only with *mgu*. For both activators to be equal, we must have

$$(\text{subreg} \circ \text{mgu}) \hat{}_d \text{ B} = \text{mgu} \hat{}_d \text{ B}$$

This is the case, since the application of *subreg* has no effect on B: the clause $H :- B$, and so especially B were renamed with a new index *vireg*, that was not used previously. Therefore *subreg* should contain no variables which were renamed with the index *vireg* at this point.

To formalise this argument we have defined predicates $cl <_{cvi}$ *vireg*, $L <_{tvi}$ *vireg*, $dgl <_{dvi}$ *vireg* and *subreg* $<_{svi}$ *vireg*, which state that clause *cl*, decorated goal list *dgl*, literal L and substitution *subreg* do not contain variables renamed with index *vireg*. The proof, that *subreg* has no effect on literal B then can be reduced to the goal, that the renaming function *rent* obeys

$$\text{rent(L,vireg)} <_{tvi} \text{vireg} + 1$$

But for a natural definition of renaming, that is homomorphic over the datatypes mentioned (for which e.g. *rent*(*f*(*t*),*vireg*) = *f*(*rent*(*t*,*vireg*)) holds) this goal can be proved only if **the literal to rename does not contain renamed variables** already. Therefore we need

Compiler Assumption for the Refinement 7/8: The original Prolog program does not contain renamed variables.

The assumption is realized in reality simply by giving renamed variables no readable representation. Nevertheless the formal verification makes this implicit assumption explicit.

We define the new compiler assumption for the original database *db* of ASM1:

$$\text{mapclause(procdef(lit,db),db)} <_{clvi} 0$$

With the previous compiler assumptions it easy to propagate it to the database db_7 of ASM7.

As the coupling invariant we finally reach after 12 attempts and one man month of work the following formula.

$$
\begin{array}{l}
\text{INV}_{78} \equiv \\
\exists\ \text{F.}\quad \text{F}[\bot] = \bot \wedge \bot \in \text{s} \\
\quad \wedge\ (\text{stop} = \text{run} \rightarrow \text{decglseqreg} \neq []) \\
\quad \wedge\ \text{stop} = \text{stop}' \wedge \text{preg} = \text{preg}' \wedge \text{vireg} = \text{vireg}' \wedge \text{subreg} = \text{subreg}' \\
\quad \wedge\ \text{ctreg} = \text{F}[\text{ctreg}'] \wedge \text{breg} = \text{F}[\text{breg}'] \wedge \text{ce}[\bot] = \bot \wedge \text{cutpt}[\bot] = \bot \\
\quad \wedge\ \neg\ \text{breg}' \ll \bot \wedge (\text{breg}' \neq \bot \rightarrow \text{b}'[\text{breg}'] \ll \text{breg}') \wedge \neg\ \text{ereg} \ll \bot \\
\quad \wedge\ \text{subreg}' <_{svi} \text{vireg}' \wedge \neg\ \text{breg}' \ll \text{cutpt}[\text{ereg}] \\
\quad \wedge\ (\quad \text{preg}' \neq \text{start} \wedge \text{stop}' = \text{run} \\
\qquad \rightarrow \text{goalreg} \neq [] \wedge \text{is\_ret}(\text{code}(\text{preg}', \text{db}_7)) \\
\qquad\quad \supset \text{breg} \neq \bot \wedge \text{preg}' = \text{p}'[\text{breg}'] ; \\
\qquad\qquad \neg\ \text{breg}' \ll \text{ctreg}' \wedge \neg\ \text{ctreg}' \ll \text{cutpt}[\text{ereg}] \\
\qquad\quad \wedge \langle \text{S-CHAIN-REC\#(act, preg, db}_7\text{; col)}\rangle\ \text{tt})) \\
\quad \wedge\ \langle \text{STACK\#(breg}', \text{b}'\text{; stack}')\rangle \\
\qquad (\quad \langle \text{STACK\#(breg, b; stack)}\rangle\ (\text{F}_\text{l}(\text{F, stack}') = \text{stack} \wedge \text{stack} \subseteq \text{s}) \\
\qquad\ \wedge\ \text{F injon stack}' \wedge \text{ordered(stack}') \\
\qquad\ \wedge\ (\quad \text{stop} = \text{run} \\
\qquad\quad \rightarrow \langle \text{STACK\#(ereg, ce; estack)}\rangle \\
\qquad\qquad \langle \text{decglseqreg}' := [\langle \text{goalreg, cutpt[ereg]}\rangle\ | \\
\qquad\qquad\qquad \text{decglseqof(cutpt, cg, ce, estack)}] \\
\qquad\qquad\ (\quad \text{decglseqreg} = \text{subreg}\ \hat{}_d\ \text{F}_\text{d}(\text{F, decglseqreg}') \\
\qquad\qquad\ \wedge\ \text{decglseqreg}' <_{dvi} \text{vireg}' \\
\qquad\qquad\ \wedge\ \text{disjoint(estack, stack}') \wedge \text{ordered(estack)} \\
\qquad\qquad\ \wedge\ (\text{preg}' = \text{start} \supset \text{decglseqreg}'\ \text{cutptsin stack}' ; \\
\qquad\qquad\qquad \neg\ \text{is\_ret}(\text{code}(\text{preg}', \text{db}_7)) \\
\qquad\qquad\qquad \rightarrow \quad \text{decglseqreg}'\ \text{cutptsin stack}'\ \text{from ctreg}' \\
\qquad\qquad\qquad\ \wedge\ (\text{ctreg}' = \bot \vee \text{ctreg}' \in \text{stack}')))) \\
\qquad\ \wedge\ \forall\ \text{n. STACKINV}_{78}),
\end{array}
$$

where

$$
\begin{array}{l}
\text{is\_ret(instr)} \leftrightarrow \quad \text{is\_retry(instr)} \vee \text{is\_retry\_me(instr)} \\
\qquad\qquad\quad \vee\ \text{is\_trust(instr)} \vee \text{is\_trust\_me(instr)}
\end{array}
$$

$\text{STACKINV}_{78} \equiv$
$\quad \text{n} \in \text{stack}'$
$\rightarrow \quad \text{sub}[\text{F}[\text{n}]] = \text{sub}'[\text{n}] \wedge \text{p}[\text{F}[\text{n}]] = \text{p}'[\text{n}] \wedge \text{ct}[\text{F}[\text{n}]] = \text{F}[\text{ct}'[\text{n}]]$
$\quad \wedge \text{b}[\text{F}[\text{n}]] = \text{F}[\text{b}'[\text{n}]] \wedge (\text{ct}'[\text{n}] \neq \bot \rightarrow \text{ct}'[\text{n}] \in \text{cdr}(\text{stack}' \text{ from n}))$
$\quad \wedge \neg\ \text{n} \ll \text{ct}'[\text{n}] \wedge \text{e}[\text{n}] \ll \text{n} \wedge \neg\ \text{e}[\text{n}] \ll \bot \wedge \neg\ \text{ct}'[\text{n}] \ll \text{cutpt}[\text{e}[\text{n}]]$
$\quad \wedge \neg\ \text{breg}' \ll \text{n} \wedge \text{goal}[\text{n}] \neq [] \wedge \text{sub}'[\text{n}] <_{svi} \text{vireg}'$
$\quad \wedge \langle \text{S-CHAIN-RET\#}(\text{act}(\text{F}[\text{n}]), \text{p}[\text{F}[\text{n}]], \text{db}_7; \text{col}) \rangle\ \text{tt}$
$\quad \wedge \langle \text{STACK\#}(\text{e}[\text{n}], \text{ce}; \text{estack}) \rangle$
$\qquad \langle \text{decglseqreg}' := [\langle \text{goal}[\text{n}], \text{cutpt}[\text{e}[\text{n}]] \rangle \mid$
$\qquad\qquad\qquad\qquad \text{decglseqof}(\text{cutpt}, \text{cg}, \text{ce}, \text{estack})] \rangle$
$\qquad\quad (\quad \text{decglseqreg}' \text{ cutptsin stack}' \text{ from ct}'[\text{n}]$
$\qquad\quad \wedge \text{decglseqreg}' <_{dvi} \text{vireg}'$
$\qquad\quad \wedge \text{disjoint}(\text{estack}, \text{stack}' \text{ from n}) \wedge \text{ordered}(\text{estack})$
$\qquad\quad \wedge \text{decglseq}[\text{F}[\text{n}]] = \text{sub}[\text{F}[\text{n}]]\ \hat{}_d\ \text{F}_\text{d}(\text{F}, \text{decglseqreg}')$

Chapter 17

8/9: Compilation of Clauses

17.1 Definition of ASM9

In the refinement from ASM8 to ASM9 clauses are decomposed into instructions for every literal. The memory db_9 of ASM9 now stores instead of a clause `p :-` $\mathtt{q}_1$`,` $\ldots\mathtt{q}_n$ an instruction sequence

$$\begin{array}{l} \text{allocate} \\ \text{unify(p)} \\ \text{call}(\mathtt{q}_1) \\ \ldots \\ \text{call}(\mathtt{q}_n) \\ \text{deallocate} \\ \text{proceed} \end{array} \tag{17.1}$$

For the case where *preg* was *start* in ASM8, *preg′* of ASM9 now takes over the role of *goalreg* (when $preg \neq start$, *preg* and *preg′* are equal). $goalreg = [q_i, \ldots q_n]$ now corresponds to a situation, in which *preg′* points to the instruction $call(q_i)$. The situation in ASM8, in which *preg* points to a clause and *enter* rule is executed corresponds to the situation in which *preg′* points to *allocate*. Execution of the *enter* rule is replaced with execution of the 2 instructions *allocate* and *unify*(*p*). Similarly the execution of *goal success* (an empty *goalreg* in ASM8 corresponds to *preg′* pointing to *deallocate*) is replaced by execution of *deallocate* and *proceed*. Splitting *enter* and *goal success* into two instructions is not strictly necessary for this refinement, but introduces instructions used in the WAM, that can be optimized in later refinements.

To be able to remove *goalreg*, it must be taken care that the renaming of variables (with *vireg*) done in the *enter* rule when *goalreg* is set, must now be postponed to the actual use of the activator. It is therefore necessary, to store the renaming index with a dynamic function *vi* in the current environment and in the environments of choicepoints.

Replacing the use of *goalreg* with *preg* makes it necessary to also replace the current goal *cg* in choicepoints with a pinter *cp* into the program code.

To complete the definition of the compilation, we finally have to define how a query $q_i, \ldots q_n$ is compiled. The result is:

$$\begin{array}{l} \text{call}(\mathtt{q}_1) \\ \ldots \\ \text{call}(\mathtt{q}_n) \\ \text{null} \end{array} \tag{17.2}$$

Instead of the instruction *null*[1] [BR95] uses the instruction *proceed*. The applicability test for *query success* rule there is

$$\text{code(preg,db}_9) = \text{proceed} \wedge \text{code(cpreg,db}_9) = \text{proceed}$$

This is not correct, when the last literal of a query is either *!* or *true*, since both instructions do not increment *cpreg*, but leave it on the current instruction. This would result in an infinite loop by repeated execution of the last instruction. There are two alternatives to our solution:

- Both the *cut* and the *true* rule finally set *cpreg* to *preg*. This solution is inefficient, since setting *cpreg* is unneccessary during regular execution.
- The compiler removes literals *true* and *!* at the end of a query, since they have no effect anyway. Although this solution is possible for the two constructs, it is problematic insofar, as an extension of Prolog by other built-in constructs (such as *assert*) would mean that the problem would have to be reconsidered.

It should also be noted, that the two alternatives cause two irregularities compared to ours:

- An empty query must either be handled specially by initialisation of *cpreg* with *preg*, or it must be completely forbidden (in our solution, no special treatment is necessary, *cpreg* need not be initialized). In the first case we have an additional 1:1 diagram to verify for the empty query.
- The rule mapping given in [BR95] that maps *goal success* to *deallocate* and *proceed* (1:2 diagram) is *not* correct for this solution. Instead (assuming a nonempty query) in both solutions the *final two* applications of *goal success* and *query success* of ASM8 correspond to *deallocate* and *query success* in ASM9. In the second solution, we also get additional 1:0 diagrams resulting from the removal of *true* und ! literals.

In [AK91] the question of successful termination is not even considered. A query seems to be compiled solely to a sequence of *call* instructions, and the end of the computation seems to be defined implicitly by reaching the adress after the last *call*.

To formalize the compiler assumption described above, we first need the following procedures *UNLOAD#* and *QUERY#*, that recover a clause or the query from compiled code:

```
UNLOAD#(coa, db9; var cl)
begin
    if code(coa,db9) = allocate ∨ is_unify(code(coa+1,db9))
    then var goalreg = []
         in begin
            UNLOADREC#(coa+2),db9,true; goalreg);
            cl := <unifylit(code(coa+1,db9)),goalreg>
            end
         else abort
end;
```

[1] reusing the instruction *null*, which in ASM2 indicated the end of a clause list, we avoid the introduction of another instruction.

```
UNLOADREC#(coa, db9, flag; var goalreg)
begin
    var instr = code(coa,db9)
    in if flag ∧ (instr = deallocate)
       then begin
            if code(coa+1,db9) = proceed then goalreg := []
                                         else abort end
            else if ¬ flag ∧ (instr = null') then goalreg := []
            else if is_call(instr) then begin
                 UNLOADREC#(coa+1,db9,flag; goalreg);
                 goalreg := [calllit(instr) | goalreg]
                 end
            else abort
    end;

QUERY#(coa, db9; var goalreg)
begin
    UNLOADREC#(coa, db9, false; goalreg)
end
```

The auxiliary procedure *UNLOADREC#* traverses successive *call* instructions. If the given *flag* = *tt*, then it checks that at the end an *allocate* and a *proceed* instruction are found (clause code), otherwise it checks for a *null* (query code). The definition of chains with switching (*S-CHAIN#*'s, see appendix D.2), is modified to *C-CHAIN#*'s by replacing the code

```
if is_clause(instr) then col := [co]
```

with

```
if instr = allocate then UNLOAD#(preg; co)
```

With this definition the weakest compiler assumption that can be stated for *(procdef*$_9$*,db*$_9$*,preg*$_9$*)* := *compile*$_{79}$*(procdef*$_7$*,db*$_7$*,query)* would be

$$\begin{array}{l} \text{[S-CHAIN\#(act,procdef}_7\text{[id(act),db}_7\text{],db}_7\text{;col)]} \\ \quad \langle \text{C-CHAIN\#(act,procdef}_9\text{[id(act),db}_9\text{],db}_9\text{;col)} \\ \qquad \text{mapcode(col}_1\text{, db}_7\text{) = mapcode(col}_2\text{, db}_9\text{)} \\ \wedge \langle \text{QUERY\#(preg}_9\text{,db}_9\text{;co)}\rangle \text{ mapcode(co,db}_9\text{) = query} \end{array} \tag{17.3}$$

But this assumption would allow to arbitrarily restructure the code for switching again. This is of course not intended. Therefore we must have a stronger assumption, that just allows to replace clauses by clause code. Care has to be taken, since the new code might make it necessary to move blocks of code. To describe such code movement we use a function C : *codesort* $\rightarrow$ *codesort*. Since the function might depend on the input program, it must be specified as a dynamic function. It would be possible to compute C as an additional result of *compile*$_9$, but since only its existence is relevant, our compiler assumption is:

$$\begin{array}{l} \text{db}_2 = \text{compile2(compile1(db))} \\ \rightarrow \quad \langle \text{QUERY\#(preg}_9\text{,db}_9\text{;co)}\rangle \text{ mapcode(co,db}_9\text{) = query} \\ \quad \wedge \exists\ \text{C. (} \quad \text{eqpdt(procdef}_7\text{,procdef}_9\text{,C)} \\ \qquad\qquad \wedge \text{ eqcode(db}_7\text{,db}_9\text{,C))} \end{array} \tag{17.4}$$

In the formula *eqpdt*($procdef_7$,$procdef_9$,C) says, that both access tables are equal modulo the code movement given by C:

$$\begin{aligned} & \text{eqpdt}(\text{procdef}_7,\text{procdef}_9,\text{C}) \\ \leftrightarrow\ & \forall\ \text{p/n.}\ \text{C}[\text{procdef}_7[\text{p/n}]] = \text{procdef}_9[\text{p/n}] \end{aligned}$$

eqcode(db_7,db_9,C) means, that all instructions, except clauses, are mapped modulo code movement to themselves. E.g. we have

$$\begin{aligned} & \text{eqcode}(\text{db}_7,\text{db}_9,\text{C}) \wedge \text{code}(\text{preg},\text{db}_7) = \text{retry}(\text{N}) \\ \rightarrow\ & \text{code}(\text{C}[\text{preg}],\text{db}_9) = \text{retry}(\text{C}[\text{N}]) \end{aligned}$$

and analogous for all other instructiosn. For clauses

$$\begin{aligned} & \text{eqcode}(\text{db}_7,\text{db}_9,\text{C}) \wedge \text{code}(\text{preg},\text{db}_7) = \text{clause} \\ \rightarrow\ & \langle \text{UNLOAD\#}(\text{C}[\text{preg}],\text{db}_9;\text{c})\rangle\text{c} = \text{clause} \end{aligned}$$

must hold. The rules of ASM9 are:

backtrack ≡
    if breg = ⊥ **then** stop := failure
        else preg := p[breg]

call rule
    if code(preg,db_9) = call(lit) ∧ is_user_defined(lit)
    then if $procdef_9$(lit,db_9) = failcode
        then backtrack
        else cpreg := preg +1
            preg := $procdef_9$(lit,db_9)
            ctreg := breg

true rule
    if code(preg,db_9) = call(!)
    then breg := cutpt[ereg]
        preg := preg +1

allocate rule
    if code(preg, db_9) = allocate
    then let tmp = max(ereg,breg)++
        ce[tmp] := ereg
        ereg := tmp
        cp[tmp] := cpreg
        vi[tmp] := vireg
        cutpt[tmp] := ctreg
        preg := preg +1

unify rule
    if code(preg, db_9) = unify(trm)
    then let act = subreg $\hat{}_t$ rent′(calllit(code(cpreg −1, db_9)), ce[ereg], vi)
    let mgu = unify(act, rent(trm, vireg))
    if mgu = nil
    then backtrack
    else subreg := subreg ∘ mgu
        vireg := vireg +1
        preg := preg +1

deallocate rule
if code(preg,db_9) = deallocate
then cpreg := cp[ereg]
ereg := ce[ereg]
preg := preg +1

true rule
if code(preg,db_9) = call(fail)
then backtrack

proceed rule
if code(preg,db_9) = proceed
then preg := cpreg

query success rule
if code(preg,db_9) = null′
then stop := success

retry rule
if code(preg,db_9) = retry(N)
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := preg +1
preg := N

retry_me_else rule
if code(preg,db_9) = retry_me_else(N)
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := N
preg := preg +1

switch_on_constant rule
let act = subreg ^$_t$ rent′(calllit(code(cpreg −1, db_9)), ereg, vi)
if code(preg, db_9) = switch_on_constant(i, tabsize, table)
then let x_i = arg(act, i)
preg := hashc(table, tabsize, constsym(x_i), db_9);
if preg = failcode **then backtrack**

switch_on_structure rule
let act = subreg ^$_t$ rent′(calllit(code(cpreg −1, db_9)), ereg, vi)
if code(preg, db_9) = switch_on_structure(i, tabsize, table)
then let x_i = arg(act, i)
preg := hashs(table, tabsize, funct(x_i), arity(x_i), db_9);
if preg = failcode **then backtrack**

switch_on_term rule
let act = subreg ^$_t$ rent′(calllit(code(cpreg −1, db_9)), ereg, vi)
if code(preg, db_9) = switch_on_term(i, N_s, N_c, N_v, N_l)
then let x_i = arg(act, i)
if is_struct(x_i) **then** preg := N_s **else**

if is_const(x_i) **then** preg := N_c **else**
if is_var(x_i) **then** preg := N_v **else**
if is_list(x_i) **then** preg := N_l;
if preg = failcode **then backtrack**

true rule
if code(preg,db_9) = call(true)
then preg := preg +1

trust rule
if code(preg,db_9) = trust(N)
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := N

trust_me rule
if code(preg,db_9) = trust_me
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

try rule
if code(preg,db_9) = try(N)
then let tmp = max(ereg,breg)++
b[tmp] := breg
e[tmp] := ereg
cp[tmp] := cpreg
sub[tmp] := subreg
p[tmp] := preg +1
breg := tmp
ct[tmp] := ctreg
preg := N

try_me rule
if code(preg,db_9) = try_me_else(N)
then let tmp = max(ereg,breg)++
b[tmp] := breg
e[tmp] := ereg
cp[tmp := cpreg
sub[tmp] := subreg
p[tmp] := N
breg := tmp
ct[tmp] := ctreg
preg := preg +1

17.2 Equivalence Proof 8/9

For the equivalence proof of ASM8 and ASM9 we have used the theorem for iterated refinement described in Sect. 6.5 for the first time. Instead of encoding all information into the coupling

invariant INV_{89}, we first derived a machine invariant $MINV_8$ from the coupling invariant INV_{78}. Since all diagrams in the refinement 7/8 are n:1 diagrams (we can set $INVNOW_8$ to be **true**), it is sufficient to show

$$\mathrm{INV}_{78} \rightarrow \mathrm{MINV}_8$$

to make $MINV_8$ usable as a precondition for all commuting diagrams. To have a machine invariant for the next refinement, we have also defined the predicate $INVNOW_9$, that characterized the states of ASM9, in which the coupling invariant holds. Now, in the refinement 8/9 all rules are refined with 1:1 diagrams, except for *enter* and *goal success* rule, which are refined with *allocate unify* resp. *deallocate proceed*. The coupling invariant therefore does not hold only in the middle states of these 1:2 diagrams and we can set

$$\begin{aligned} &\mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9) \\ \equiv\ &\mathrm{code}(\mathrm{preg}',\mathrm{db}_9) \neq \mathrm{proceed} \wedge \neg\ \mathrm{is\_unify}(\mathrm{code}(\mathrm{preg}',\mathrm{db}_9)) \end{aligned}$$

The proof obligations for the two 1:2 diagrams are the special case with $j := 2$ and $i := 1$ of the proof obligations (6.32) from Sect. 6.5:

$$\begin{aligned} &\mathrm{INV}_{89} \wedge \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \\ &\wedge\ \mathrm{MINV}_8 \wedge \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg},\mathrm{db}_7)) \\ \rightarrow\ &\langle \mathrm{RULE}_9 \rangle\ (\quad \neg\ \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9) \\ &\qquad\qquad \wedge\ \langle \mathrm{RULE}_9 \rangle\ \langle \mathrm{RULE}_8 \rangle \\ &\qquad\qquad\quad (\mathrm{INV}_{89} \wedge \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9)) \end{aligned}$$

$$\begin{aligned} &\mathrm{INV}_{89} \wedge \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \\ &\wedge\ \mathrm{MINV}_8 \wedge \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg},\mathrm{db}_7)) \\ \rightarrow\ &\langle \mathrm{RULE}_9 \rangle\ (\quad \neg\ \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9) \\ &\qquad\qquad \wedge\ \langle \mathrm{RULE}_9 \rangle\ \langle \mathrm{RULE}_8 \rangle \\ &\qquad\qquad\quad (\mathrm{INV}_{89} \wedge \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9)) \end{aligned}$$

For the definition of the coupling invariant we found the following 4 main problems:

Correct Treatment of Termination In our first proof attempts, we tried to follow [BR95]. Thereby we found the problems already described in the previous section: first, we had to correct the choice of diagrams (a special diagram was necessary for the empty query, and a 2:2 diagram was necessary for *goal success*, *query success* in ASM8 vs. *deallocate query success* in ASM9). Then the proof for the equivalence of the *cut* rules failed, since the *cut* rule of ASM9 does not modify *cpreg*. This failure resulted in the correction of *query success* in ASM9.

No Instantiation of the Literal in Call Rule In the *is_user_defined* tests as well as in the selection of the leading predicate symbol in the *call* rules all ASMs until ASM8 have used the *instantiated* activator. ASM9 now uses instead the *uninstantiated* literal L from the instruction *call*(L). For the computation of the leading predicate symbol we have anticipated the modification from the refinement 9/10. This was done to free the already complex verification from unnecessary additional problems.

Verification now showed, that when using the uninstantiated literal, we must restrict the accepted Prolog language: ASMs 1–8 gave a positive answer to the query `?- p(q).`, given the clauses `p(X) :- X.` and `q.`. ASM9 can not deal with such a query, since the leading predicate symbol of an uninstantiated variable `X` is not defined. Given a query `?- p(!)` (and the same program), ASM9 in [BR95] even tries incorrectly to compute a leading predicate symbol instead of executing the cut. The difficulty of defining a leading predicate symbol also occurs, when the body literals are lists. Since usual Prolog implementation do not have a "list predicate", and instead interpret such a literal as a command to load a file, we define

Compiler Assumption for the Refinement 8/9: No literal of the query and no literal in any clause of the prolog may be a variable or a list.

Of course all ASMs up to now could not "meaningfully" solve a query `?- X.`, since there is no meaningful definition of the leading predicate symbol for a variable. But this was irrelevant for correctness, since however the selection function was defined for the case of a variable, all ASMs behaved in the same way. The core of the problem therefore is, that the semantic definition of Prolog is incomplete for this case.

If we would define the compiler assumption for the refinement 8/9 as above, this would result in additional formulas in the coupling invariant. For all literals, for which from the machine invariant $MINV_8$ for ASM8 it is already known, that they are not renamed, we would now additionally need, that they are no variables and no lists. This would mean that we would have to compute the chains, from which the literals are selected, twice, once in $MINV_8$ and once in INV_{89}. To avoid this, we have strengthened the predicate $cl <_{cvi} vireg$ used in compiler assumption 7/8 to include the compiler assumption for 8/9, i.e. that cl does not have literals which are just variables or lists. This does not change the proofs for the refinement 7/8 (since we have just strengthened the assumptions), and the assumptions that we have no variables or lists as literals, is now covered already by $MINV_8$.

Moving Renaming of the Activator to its Actual Use Since goals are no longer stored explicitly in a register in ASM9, but are only referenced by a pointer to the clause code, the renaming index necessary to rename clause variables before unification must now be stored in the environment and its use is postponed until the literal is actually used. To reconstruct a goal from a pointer to code we use the procedure *UNLOADREC#* from the compiler assumption. For the actual renaming of goal variables, we first defined a function *reng*, that renames all variables of a goal with some index (*reng* is homomorphic to the function *rename* defined earlier for renaming of clauses). In [BR95] collection of literals and application of the renaming is merged together in the function g defined on p. 34f. The assumption $goalreg = g(Ptr,vireg)$ therefore reads in our notation:

$$\langle \text{UNLOADREC\#(Ptr,db}_9\text{;goal)}\rangle \text{ goalreg} = \text{reng(goalreg,vireg)}$$

Verification revealed, that this assumption is not correct in the case where *goalreg* is a part of the initial query, since the query *must not* be renamed. It turns out, that in the coupling invariant this case corresponds to an attempt to compute *vireg* as the unspecified $vi[ereg]$ for $ereg = \perp$. We have specified the exceptional case explicitly, using a function $reng'(goalreg,ereg,vi)$ with the axioms

$$\text{reng}'(\text{goalreg},\perp,\text{vi}) = \text{goalreg}$$
$$\text{ereg} \neq \perp \rightarrow \text{reng}'(\text{goalreg},\perp,\text{vi}) = \text{reng(goalreg,vi[ereg])}$$

An alternative would have been to initialize $vi[\perp]$ in such a way that application of this renaming has no effect (e.g. initialization of $vi[\perp]$ with 0, of *vireg* with 1, and definition of $reng(goalreg,0)$ as *goalreg*).

Reconstruction of *goalreg* from ASM8 Using Data from ASM9 The central point in the definition of the coupling invariant for 8/9 is to reconstruct the goals stored explicitly in ASM8, that are only implicitly represented by pointers to code in ASM9. The main task in doing this was to give a precise definition of the "Continuation Pointer Constraint" ([BR95], p. 34) and to give a precise formalization of how the registers of ASM9 can be reconstructed from the data of ASM9. We found that the uniform reconstruction as given in [BR95], p. 35 was not possible. Instead three cases had to be defined:

In the first case ASM8 is in a state where *preg* = *start*, and *goalreg* is reconstructed by

$$\langle \text{UNLOADREC\#}(\text{preg}', \text{db}_9, \text{ereg}' \neq \bot\ ;\ \text{goalreg}_0)\ \text{end}\rangle$$
$$(\quad \text{reng}'(\text{goalreg}_0, \text{ereg}', \text{vi}) = \text{goalreg}$$
$$\wedge\ \text{nonvargoal}(\text{goalreg}_0))$$

The postcondition *nonvargoal*(*goalreg*$_0$) encodes the compiler assumptions that the literals of *goalreg* are neither renamed, nor variables or lists.

In the second case both ASMs are before a *retry*-, *retry_me*-, *trust*- or *trust_me* instruction. In this case no *goalreg* must be reconstructed (the instruction will set it from the choicepoint). For this case it has also to be noted, that the two environment registers *ereg* and *ereg*′ may be *different*: When an *enter* with backtracking is executed in ASM8, *ereg* is unchanged, while the corresponding *allocate* in ASM9 will modify *ereg*′.

The continuation pointer constraint is not needed in the first two cases, but in the remaining third case. In this case we have *preg*′ = *C*[*preg*] and *goalreg* is computed with *cpreg* −1:

$$\langle \text{UNLOADREC\#}(\text{cpreg} - 1, \text{db}_9, \text{ereg}' \neq \bot;\ \text{goalreg}_0)\ \text{end}\rangle$$
$$(\quad \text{reng}'(\text{goalreg}_0, \text{ereg}', \text{vi}) = \text{goalreg}$$
$$\wedge\ \text{nonvargoal}(\text{goalreg}_0))$$

When we tried to determine how exactly this formula should look like, we tried several proof attempts with *ce*′[*ereg*′] instead of *ereg*′, since otherwise we could not verify the refinement of the *enter* rule to *allocate unify*. After some analysis of failed proof attempts we found, that the problem was the renaming index used in the *unify* rule. In [BR95] this renaming index for the activator *act* is defined indirectly via the abbreviation *goal* as *vi*[*ereg*]. This is correct for the switching rules and the *call* rule, but not for the *unify* rule, since immediately before the *allocate* rule already pushes a *new* renaming index onto the environment stack. This new index should be used for the new goal that would be pushed onto the environment stack on successful unification. The correct renaming index therefore is found at *vi*[*ce*[*ereg*]], when *ce*[*ereg*] ≠ ⊥. Therefore the corrected *unify* rule calls the function *rent*′ with *ce*[*ereg*].

Putting all things together we reached after 3 weeks and 8 iterations the following coupling invariant was

$$\begin{array}{l}
\text{INV}_{89} \equiv \\
\quad \text{vireg} = \text{vireg}' \wedge \text{stop} = \text{stop}' \wedge \text{breg} = \text{breg}' \wedge \text{ctreg} = \text{ctreg}' \wedge \text{sub} = \text{sub}' \\
\wedge\ \text{subreg} = \text{subreg}' \wedge \text{ct} = \text{ct}' \wedge \text{b} = \text{b}' \wedge \text{e} = \text{e}' \wedge \text{cutpt}'[\bot] = \bot \\
\wedge\ (\quad \text{stop} = \text{run} \\
\quad \rightarrow \quad \neg\ \text{is_unify}(\text{code}(\text{preg}', \text{db}_9)) \wedge \text{code}(\text{preg}', \text{db}_9) \neq \text{proceed} \\
\qquad \wedge\ (\quad \text{preg} = \text{start} \\
\qquad\quad \supset \quad \text{ereg} = \text{ereg}' \\
\qquad\qquad \wedge\ \langle \text{UNLOADREC\#}(\text{preg}', \text{db}_9, \text{ereg}' \neq \bot;\ \text{goalreg}_0)\rangle \\
\qquad\qquad\qquad \text{reng}'(\text{goalreg}_0, \text{ereg}', \text{vi}) = \text{goalreg} \wedge \text{nonvargoal}(\text{goalreg}_0); \\
\qquad\qquad\quad \neg\ \text{is_call}(\text{code}(\text{preg}', \text{db}_9)) \\
\qquad\qquad \wedge\ \text{code}(\text{preg}', \text{db}_9) \neq \text{deallocate} \\
\qquad\qquad \wedge\ \text{preg}' = \text{C}[\text{preg}] \\
\qquad\qquad \wedge\ \quad \neg\ \text{is_ret}(\text{code}(\text{preg}, \text{db}_7)) \\
\qquad\qquad\quad \rightarrow \quad \text{ereg} = \text{ereg}' \\
\qquad\qquad\qquad \wedge\ \langle \text{UNLOADREC\#}(\text{cpreg}{-}1, \text{db}_9, \text{ereg}' \neq \bot;\ \text{goalreg}_0)\rangle \\
\qquad\qquad\qquad\quad \text{reng}'(\text{goalreg}_0, \text{ereg}', \text{vi}) = \text{goalreg} \wedge \text{nonvargoal}(\text{goalreg}_0)) \\
\qquad \wedge\ \langle \text{STACK\#}(\text{ereg}, \text{ce}; \text{estack})\rangle \\
\qquad\quad \forall\ \text{n}. \quad \text{n} \in \text{estack} \\
\qquad\qquad \rightarrow \quad \text{ce}[\text{n}] = \text{ce}'[\text{n}] \wedge \text{cutpt}[\text{n}] = \text{cutpt}'[\text{n}] \\
\qquad\qquad\quad \wedge\ \langle \text{UNLOADREC\#}(\text{cp}[\text{n}], \text{db}_9, \text{ce}[\text{n}] \neq \bot;\ \text{goalreg}_0)\rangle \\
\qquad\qquad\qquad \text{reng}'(\text{goalreg}_0, \text{ce}[\text{n}], \text{vi}) = \text{cg}[\text{n}] \wedge \text{nonvargoal}(\text{goalreg}_0) \\
\qquad \wedge\ \langle \text{STACK\#}(\text{breg}, \text{b}; \text{stack})\rangle
\end{array}$$

$$
\begin{array}{l}
\forall\, n.\quad n \in stack \\
\quad \rightarrow\quad p'[n] = C[p[n]] \\
\qquad \wedge\ \langle STACK\#(e[n],\ ce;\ estack)\rangle \\
\qquad\quad \forall\, n_0.\quad n_0 \in estack \\
\qquad\qquad \rightarrow\quad ce[n_0] = ce'[n_0] \wedge cutpt[n_0] = cutpt'[n_0] \\
\qquad\qquad\quad \wedge\ \langle UNLOADREC\#(cp[n_0],\ db_9, \\
\qquad\qquad\qquad\qquad ce[n_0] \neq \bot;\ goalreg_0)\rangle \\
\qquad\qquad\qquad\quad reng'(goalreg_0,\ ce[n_0],\ vi) = cg[n_0] \\
\qquad\qquad\qquad \wedge\ nonvargoal(goalreg_0) \\
\qquad \wedge\ \langle UNLOADREC\#(cp[n]-1,\ db_9,\ e[n] \neq \bot;\ goalreg_0)\rangle \\
\qquad\quad reng'(goalreg_0,e[n],vi) = goal[n] \wedge nonvargoal(goalreg_0) \\
\wedge\ eqcode(db_7,\ db_9,\ C) \\
\wedge\ eqpdt(procdeftab_7,\ procdeftab_9,\ C))
\end{array}
$$

The invariant, and so the number of conjuncts to prove, would have been about twice the size without using the technique for iterated refinements, as can be seen from the machine invariant $MINV_8$ for ASM8:

$$
\begin{array}{l}
MINV_8 \equiv \\
\quad stop = run \\
\rightarrow\quad (preg \neq start \rightarrow goalreg \neq []) \wedge ce[\bot] = \bot \wedge cutpt[\bot] = \bot \\
\quad \wedge\ (\quad is\_retry\_me(code(preg,\ db_8)) \vee is\_retry(code(preg,\ db_8)) \\
\qquad \vee\ is\_trust\_me(code(preg,\ db_8)) \vee is\_trust(code(preg,\ db_8)) \\
\qquad \rightarrow breg \neq \bot \wedge preg = p[breg]) \\
\quad \wedge\ (\quad preg \neq start \\
\qquad \wedge \neg\ is\_retry\_me(code(preg,\ db_8)) \wedge \neg\ is\_retry(code(preg,\ db_8)) \\
\qquad \wedge \neg\ is\_trust\_me(code(preg,\ db_8)) \wedge \neg\ is\_trust(code(preg,\ db_8)) \\
\qquad \rightarrow \langle S\text{-}CHAIN\text{-}REC\#(subreg\ \hat{}_t\ car(goalreg),\ preg,\ db_8;\ col)\rangle \\
\qquad\quad mapcode(col,\ db_8) <_{clvi} 0) \\
\quad \wedge\ \langle STACK\#(breg,\ b;\ stack)\rangle \\
\qquad \langle b\text{-}list\#(ereg,\ ce;\ estack)\rangle \\
\qquad\quad (\quad (\quad preg = start \\
\qquad\qquad \vee \neg\ is\_retry\_me(code(preg,\ db_8)) \\
\qquad\qquad \rightarrow cutpt[ereg] \in stack \vee cutpt[ereg] = \bot) \\
\qquad\quad \wedge\ ordered(stack) \wedge ordered(estack) \wedge disjoint(stack,\ estack) \\
\qquad\quad \wedge\ (\forall\, n.\quad n \in stack \\
\qquad\qquad\qquad \rightarrow\quad e[n] \ll n \wedge goal[n] \neq [] \\
\qquad\qquad\qquad\quad \wedge\ \langle STACK\#(e[n],\ ce;\ estack)\rangle \\
\qquad\qquad\qquad\qquad (\quad disjoint(estack,\ stack\ from\ n) \\
\qquad\qquad\qquad\qquad\ \wedge\ ordered(estack)) \\
\qquad\qquad\qquad\quad \wedge\ \langle S\text{-}CHAIN\text{-}RET\#(sub[n]\ \hat{}_t\ car(goal[n]), \\
\qquad\qquad\qquad\qquad\qquad\qquad p[n],\ db_8;\ col)\rangle \\
\qquad\qquad\qquad\qquad mapcode(col,\ db_8) <_{clvi} 0))
\end{array}
$$

$MINV_8$ encodes properties of ASM8, that were already proved in the refinement 7/8, like disjointness of the environment and the choicepoint stack. These properties could be assumed for 8/9, and had not to be proved anew.

Chapter 18

9/10: Compilation of Terms

This chapter describes our current work on the first refinement from Chapter 4 in [BR95]. Besides the refinement 5/7 this seems to be the most complex refinement. Although we were not successful to verify it completely in the course of this work, our attempts to formalize the refinement and first proof attempts have nevertheless uncovered a number of problems. One part of the problems resulted from misunderstanding several aspects of the refinement, another part was due to the fact, that the correctness assertions in [BR95] are given only very informally. We will therefore not give a complete detailed description of the refinement, but only sketch some of the problems we found in the refinement and sketch some approaches how to solve them.

The main aspect of the refinement 9/10 is the representation of terms by pointer structures on the *heap* (introduced in the refinement), and the compilation of literals to instructions, that create and unify such pointer structures. Unfortunately this is not the only modification that is done to ASM9. Several other aspects of the WAM are also introduced in the refinement:

- The implementation of ASM10 does not have an occur check. But how can we formalize the condition "ASM9 does not call an occur check"?
- Instead of storing substitutions, ASM10 now uses another stack, the *trail*, to store variable bindings. When, due to backtracking, an old substitution is needed, variable bindings are undone destructively.
- The stack of environments and choicepoints in ASM10 is "flat". It has no internal structure anymore as the previous one. The different components are now stored in successive addresses, and accessed uniformly with a function *val*.
- ASM10 in [BR95] does not consider the cut. The cut is reintroduced at the end of Chapter 4.
- Variable renaming is now done by allocation of a variables at a new address instead of using a renaming index. The *allocate* instruction suggests that the new address allocated may be a *locally new* address of the environment stack, not a *globally new* heap address. But it turned out, that this assumption is wrong (which does not mean, that the ASM given in [BR95] is wrong, see below). The temporary use of locally new heap addresses rises the problem, how a correct mapping between globally renamed variables in ASM9 to locations in ASM10 should look like.
- It turns out, that the substitutions stored in ASM9 do *not* correspond to those stored in ASM10. Instead certain variable bindings, that are no longer relevant, are discarded earlier than in ASM9.

Only the first four aspects mentioned above are discussed explicitly in [BR95]. To reduce the complexity of verification, we have tried to remove all aspects from the refinement that are not

coupled to the introduction of term representation. Therefore, as a first step, we have kept the structure of environments and choicepoints. Storing variables in an environment is done in our ASM10 with a function $x : env \times nat \to node$: the result of $x(ereg,m)$ is the mth variable of the current environment (the sort *node* is now simply the sort of memory addresses, a super sort of *env*). The structure of main memory in the WAM assumes, that heap addresses are lower than stack addresses. This gives a complex ordering $\ll$ on memory addresses, for which the axioms

$$\bot + m_1 \ll x(\bot + m_2, m_3)$$
$$x(\bot + m_0, m_1) \ll x(\bot + m_2, m_3) \leftrightarrow m_0 < m_2 \vee m_0 \ll m_2 \wedge m_1 < m_3$$

hold. The function *val: heap* $\to$ *termrep* is used only to determine the content of heap locations (*heap* now is also a subsort of *node*).

As a second measure, we have kept the cut, which is easily possible, since we have kept the structure of stacks (an instruction to remove variable bindings from the stacks is of course necessary; otherwise we simply keep the registers of the previous ASM).

Third, we have kept the occur check of unification. The "Meta Theorem", which says, that if occur check is not called, it can be removed holds trivially for ASM10, too. Also keeping the occur check has allowed us to falsify the statement, made in [AK91], p. 14 as well as in [BR95], p. 39, that occur check should be simply integrated into the *bind* routine: an occur check is also necessary in the *unify_value* instruction.

Fourth, we have tried to change the strategy of variable renaming already on the term level. The idea was, that renaming a variable X with the current renaming index can be replaced by using a new stack address $x(ereg,m)$. The transition from a globally new variable to a variable that is relatively new to the stack is suggested by the *allocate* instruction of ASM10 in [BR95], which allocates the new variable in just this way. Therefore we defined a variant ASM9a of ASM9, that used new stack locations instead of a renaming index. But after some verification efforts, an attempt to verify the equivalence of the *deallocate* rules failed, because the deallocated variables can still occur in computed substitutions, that are needed later on. The bindings of these variables would be overwritten, when a new environment is allocated.

This would suppose at first glance, that ASM10 is incorrect. But a thorough analysis shows, that although a new variable X in ASM10 is first allocated on the stack, it is moved to the heap when it occurs in the variables of some term T ($X \in vars(T)$) that is bound to some other variable (by the instructions *unify_variable* and *unify_local_value*). Therefore in some cases variables in the WAM are renamed *several times*.

Of some help to understand how renaming really works was [AK91]. The first variant of the WAM that is given there does not allocate variables with an *allocate* instruction on the stack. Instead when the variable first occurs, it is allocated in the heap. Still there is one exception: if the variable is bound to a term on its first occurrence (in the instruction *get_variable*, that is generated for a variable X in a clause head `p(X)`) it is easy to see, that it can be allocated in the stack, since it will not play any role in the further computation.

The optimizations shown in [AK91] as well as in [BR95] (especially "last call optimization" LCO) are tightly coupled with the question, under which circumstances variables can be allocated in the stack instead of the heap. Therefore we think that this question should not be addressed in the refinement 9/10. It should be easier to move variables from the heap to the stack in one separate refinement, which also changes the relevant constraints for address allocation ("heap variables constraint" and "stack variables constraint").

Using a separate refinement also seems to be desirable, since the main theorems of the refinement 9/10, which are the "Getting Lemma" and the "Putting Lemma" depend on the exact definition of these constraints: it is impossible to *first* prove putting and getting lemma, as [BR95] suggests, and *then* to verify that heap and stack variables constraints as invariants of the getting and putting instructions. Instead, we have found, that both constraints are *necessary preconditions* for getting and putting lemma. Ultimately both constraints become a central part of the coupling invariant for the refinement 9/10.

Each modification in the definition of both constraints (especially each modification of the allocation of variables in the heap or the stack) therefore means, that its invariance in the putting

and getting instructions has to be proved anew. Therefore we currently use the first definition of [AK91] for our refinement. This definition has an inefficient *put_variable* instruction (that allocates the variable in the heap), no *unify_local_value* instruction, and instead of initializing all variables in *allocate* variables are initialized on their first occurrence, like this is done in [BR95] later on (p. 58f).

This version of ASM10 allows to define a very simple heap and stack variables constraint, that says, that each pointer structure representing a term has to be completely in the heap, except for the leading cell. The leading cell may be stored on the stack or in a register, if it is not a reference to itself (i.e. a representation of a variable). The ordering on addresses is not relevant for this version of the constraints. We currently think, that it should be possible to define a dynamic function, that is a bijection between the variables of ASM9 that are renamed with a global *vireg*, and the new heap variables of ASM10. Like function F in the refinement 1/2 (see section 11.2) this function should be modified each time an instruction is encountered, that corresponds to the first occurrence of a variable (other modifications should be unnecessary).

We will then try to do the shifting of variables from the heap to stack (and the introduction of stronger constraints, the definition of temporary and permanent variables and the addition of new instructions like *put_unsafe_value* etc.) in *one* separate refinement.

Even when using the ASM10 as defined in [AK91] it is unavoidable to store fewer variable bindings than in ASM9. Our current assumption is, that the (implicit) deallocation of variable bindings that is done when the environment *ereg* is deallocated in ASM10, corresponds exactly to an explicit deallocation of all bindings for variables renamed with *vi*[*ereg*] from *subreg* in ASM9. According to our philosophy, to remove as much burden from the refinement 9/10 as possible, we have therefore defined a function *remove(subreg,vi*[*ereg*]*)* and verified separately, that modifying the *deallocate* rule of ASM9 to

deallocate rule
if code(preg,db_9) = deallocate
then cpreg := cp[ereg]
ereg := ce[ereg]
preg := preg +1
subreg := remove(subreg,vi[ereg])

does not have a significant consequence on the result of ASM9: if the computation terminates, the substitution computed by the modified ASM (ASM9a) still has the same effect on the query. We could verify the equivalence of ASM9 and ASM9a in 2 weeks with 3 iterations. The coupling invariant INV_{99a} and the machine invariant $MINV_9$ for ASM9 are

$$
\begin{aligned}
&\mathrm{INV}_{99a} \equiv \\
&\quad \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{breg} = \mathrm{breg}' \wedge \mathrm{ctreg} = \mathrm{ctreg}' \wedge \mathrm{cpreg} = \mathrm{cpreg}' \\
&\wedge\ \mathrm{ereg} = \mathrm{ereg}' \wedge \mathrm{preg} = \mathrm{preg}' \wedge \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{cp} = \mathrm{cp}' \\
&\wedge\ \mathrm{p} = \mathrm{p}' \wedge \mathrm{b} = \mathrm{b}' \wedge \mathrm{e} = \mathrm{e}' \wedge \mathrm{ce} = \mathrm{ce}' \wedge \mathrm{ct} = \mathrm{ct}' \wedge \mathrm{vi} = \mathrm{vi}' \\
&\wedge\ \mathrm{cutpt} = \mathrm{cutpt}' \wedge \mathrm{cutpt}[\bot] = \bot \wedge \mathrm{ce}[\bot] = \bot \\
&\wedge\ \mathrm{subreg} <_{svi} \mathrm{vireg} \wedge \mathrm{subreg}' <_{svi} \mathrm{vireg} \\
&\wedge\ (\forall\ \mathrm{lit}.\ \mathrm{lit} <_{tvi} 0 \rightarrow \mathrm{subreg}\ \hat{}_t\ \mathrm{lit} = \mathrm{subreg}'\ \hat{}_t\ \mathrm{lit}) \\
&\wedge\ \langle \mathrm{STACK\#}(\mathrm{ereg},\ \mathrm{ce};\ \mathrm{estack})\rangle \\
&\qquad (\quad \mathrm{slnodups}(\mathrm{estack}) \wedge \mathrm{nlnodups}(\mathrm{vilist}(\mathrm{vi},\ \mathrm{estack})) \\
&\qquad\ \wedge\ (\quad \neg\ \mathrm{is\_ret}(\mathrm{code}(\mathrm{preg},\ \mathrm{db}_9)) \wedge \mathrm{stop} = \mathrm{run} \\
&\qquad\qquad \rightarrow \mathrm{vilist}(\mathrm{vi},\ \mathrm{estack}) <_{nl} \mathrm{vireg}) \\
&\qquad\ \wedge\ (\forall\ \mathrm{n},\ \mathrm{lit}.\quad \mathrm{lit} <_{tvi} 0 \wedge \mathrm{n} \in \mathrm{estack} \\
&\qquad\qquad\qquad \rightarrow \mathrm{subreg}\ \hat{}_t\ \mathrm{rent}(\mathrm{lit},\ \mathrm{vi}[\mathrm{n}]) = \mathrm{subreg}'\ \hat{}_t\ \mathrm{rent}(\mathrm{lit},\ \mathrm{vi}[\mathrm{n}]))) \\
&\wedge\ \langle \mathrm{STACK\#}(\mathrm{breg},\ \mathrm{b};\ \mathrm{stack})\rangle \\
&\qquad \forall\ \mathrm{n},\ \mathrm{lit}.\quad \mathrm{lit} <_{tvi} 0 \wedge \mathrm{n} \in \mathrm{stack} \\
&\qquad\qquad \rightarrow\quad \mathrm{sub}[\mathrm{n}]\ \hat{}_t\ \mathrm{rent}'(\mathrm{lit},\ \mathrm{e}[\mathrm{n}],\ \mathrm{vi}) = \mathrm{sub}'[\mathrm{n}]\ \hat{}_t\ \mathrm{rent}'(\mathrm{lit},\ \mathrm{e}[\mathrm{n}],\ \mathrm{vi}) \\
&\qquad\qquad\quad \wedge\ \mathrm{sub}[\mathrm{n}]\ \hat{}_t\ \mathrm{lit} = \mathrm{sub}'[\mathrm{n}]\ \hat{}_t\ \mathrm{lit} \\
&\qquad\qquad\quad \wedge\ \mathrm{sub}[\mathrm{n}] <_{svi} \mathrm{vireg} \wedge \mathrm{sub}'[\mathrm{n}] <_{svi} \mathrm{vireg}
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ \langle \mathrm{STACK\#(e[n],\ ce;\ estack_0)} \rangle \\
\qquad\qquad\qquad\qquad\qquad (\quad \mathrm{vilist(vi,\ estack_0)} <_{nl} \mathrm{vireg} \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathrm{slnodups(estack_0)} \wedge \mathrm{nlnodups(vilist(vi,\ estack_0))} \\
\qquad\qquad\qquad\qquad\qquad \wedge\ (\forall\ \mathrm{n_0}.\quad \mathrm{n_0} \in \mathrm{estack_0} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{sub[n]}\ \hat{}_t\ \mathrm{rent(lit,\ vi[n_0])} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad = \mathrm{sub'[n]}\ \hat{}_t\ \mathrm{rent(lit,\ vi[n_0])}))
\end{array}
$$

$$
\begin{array}{l}
\mathrm{MINV_9} \equiv \\
\quad \mathrm{stop} = \mathrm{run} \\
\rightarrow \quad \neg\ \mathrm{is\_unify(code(preg,\ db_9))} \wedge \mathrm{code(preg,\ db_9)} \neq \mathrm{proceed} \\
\quad \wedge\ (\qquad \mathrm{is\_try(code(preg,\ db_9))} \vee \mathrm{is\_try\_me(code(preg,\ db_9))} \\
\qquad\quad \vee\ \mathrm{code(preg,\ db_9)} = \mathrm{allocate} \vee \mathrm{is\_sw\_const(code(preg,\ db_9))} \\
\qquad\quad \vee\ \mathrm{is\_sw\_term(code(preg,\ db_9))} \vee \mathrm{is\_sw\_struct(code(preg,\ db_9))} \\
\qquad\quad \vee\ \mathrm{code(preg,\ db_9)} = \mathrm{allocate} \\
\qquad \supset \quad \langle \mathrm{C\text{-}CHAIN\text{-}REC\#(subreg}\ \hat{}_t\ \mathrm{rent'(calllit(code(cpreg{-}1),\ db_9)),} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{ereg,\ vi),}\quad \mathrm{preg,\ db_9;\ cli)} \rangle\ \mathrm{cli} <_{clvi} 0 \\
\qquad\quad \wedge\ \langle \mathrm{UNLOADREC\#(cpreg} - 1,\ \mathrm{db_9},\ \mathrm{ereg} \neq \bot;\ \mathrm{goalreg)} \rangle \\
\qquad\qquad\quad (\mathrm{goalreg} \neq [] \wedge \mathrm{nonvargoal(goalreg)})\ ; \\
\qquad\quad \mathrm{is\_ret(code(preg,\ db_9))} \supset \mathrm{breg} \neq \bot \wedge \mathrm{preg} = \mathrm{p[breg]}\ ; \\
\qquad\qquad \langle \mathrm{UNLOADREC\#(preg,\ db_9,\ ereg} \neq \bot;\ \mathrm{goalreg)} \rangle \\
\qquad\qquad\quad \mathrm{nonvargoal(goalreg))} \\
\quad \wedge\ \langle \mathrm{STACK\#(breg,\ b;\ stack)} \rangle \\
\qquad\quad \langle \mathrm{STACK\#(is\_ret(code(preg,\ db_9))} \supset \mathrm{e[breg]\ ;\ ereg,\ ce;\ estack)} \rangle \\
\qquad\qquad (\quad \mathrm{ordered(estack)} \wedge \mathrm{ordered(stack)} \wedge \mathrm{disjoint(estack,\ stack)} \\
\qquad\qquad\ \wedge\ (\quad \neg\ \mathrm{is\_ret(code(preg,\ db_9))} \\
\qquad\qquad\qquad \rightarrow \mathrm{cutpt[ereg]} \in \mathrm{stack} \vee \mathrm{cutpt[ereg]} = \bot) \\
\qquad\qquad\ \wedge\ (\forall\ \mathrm{n.}\quad \mathrm{n} \in \mathrm{estack} \\
\qquad\qquad\qquad\qquad \rightarrow \langle \mathrm{UNLOADREC\#(cp[n],\ db_9,\ ce[n]} \neq \bot;\ \mathrm{goalreg)} \rangle \\
\qquad\qquad\qquad\qquad\quad \mathrm{nonvargoal(goalreg))} \\
\qquad\qquad\ \wedge\ (\forall\ \mathrm{n.}\quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{e[n]} \ll \mathrm{n} \\
\qquad\qquad\qquad\qquad\ \wedge\ \langle \mathrm{STACK\#(e[n],\ ce;\ estack_0)} \rangle \\
\qquad\qquad\qquad\qquad\qquad (\quad \mathrm{disjoint(estack_0,\ stack\ from\ n)} \\
\qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{ordered(estack_0)} \\
\qquad\qquad\qquad\qquad\qquad\ \wedge\ (\forall\ \mathrm{n_0}.\quad \mathrm{n_0} \in \mathrm{estack_0} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \langle \mathrm{UNLOADREC\#(cp[n_0],\ db_9,} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{ce[n_0]} \neq \bot;\ \mathrm{goalreg)} \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{nonvargoal(goalreg)))} \\
\qquad\qquad\qquad\qquad\ \wedge\ \langle \mathrm{UNLOADREC\#(cp[n]} - 1,\mathrm{db_9},\ \mathrm{e[n]} \neq \bot;\ \mathrm{goalreg)} \rangle \\
\qquad\qquad\qquad\qquad\qquad (\mathrm{goalreg} \neq [] \wedge \mathrm{nonvargoal(goalreg)}) \\
\qquad\qquad\qquad\qquad\ \wedge\ \langle \mathrm{C\text{-}CHAIN\text{-}RET\#(} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{sub[n]}\ \hat{}_t\ \mathrm{rent'(calllit(code(cp[n]} - 1,\ \mathrm{db_9})), \\
\qquad\qquad\qquad\qquad\qquad \mathrm{e[n],\ vi),\ p[n],\ db_9;\ cli)} \rangle \\
\qquad\qquad\qquad\qquad\quad \mathrm{cli} <_{clvi} 0))
\end{array}
$$

Chapter 19

Statistics

The following table gives an overview over the efforts needed for the Prolog-WAM case study. For each refinement the number of necessary proof steps and interactions and the number of theorems proved are listed. These numbers have been extracted from the current KIV version 4. The number of iterations, that were necessary to reach the final coupling invariant, and the time that was needed to successfully verify the refinement refer to version of KIV in which the refinements were verified originally (for 1/2 and 4/5 KIV version 1, for 2/3,3/4,5/6 and 5/7 KIV version 3).

	1/2	2/3	3/4	4/5	5/6
Proof steps	1074	1760	2546	1722	5341
Interactions	161	124	300	87	672
Theorems	15	13	22	17	42
Iterations	12	8	5	9	8
Verif. time	2 Mo.	2 Wo.	1 Wo.	1 Mo.	2 Wo.
Size of INV	20	25	25	14	53

	5/7	7/8	8/9	9/9a
Proof steps	7558	3445	4295	3045
Interactions	1383	336	377	426
Theorems	39	21	19	19
Iterations	17	12	8	4
Verif. time	2 Mo.	1 Mo.	3 Wo.	2 Wo.
Size of INV	36	36	23+17	18+23

Altogether the verification effort is currently about 9 man months, which includes the verification of 1771 auxiliary first-order lemmas, that required 17458 proof steps and 3393 interactions. Here are some more statistical data:

- The number of auxiliary first-order lemmas is now four times the number that were necessary until refinement 5/7. The main reason is, that starting from refinement 8/9 a lot of lemmas are necessary for unification, renaming and substitution. Some of these lemmas required elaborate proofs due to the complex termination ordering of substitution (up to 20 interactions), in contrast to all lemmas proved previously (usually 0–2 interactions). A second reason is, that for ASM10 a large number of simple lemmas for the representation of terms by pointers, that have already been proved.

- Compared to the number given in [SA97], which referred to KIV version 3, there have been some major improvements. The most significant is the reduction of the size of the coupling invariant for 5/7 from 97 to 36 lines by a modification of the proof technique (see Sect. 15.2). In the refinements 2/3 and 3/4 we have now used the generation of proof obligations according to the modularization theorem. In [SA97] the generic proof for the modularization theorem was still done was done for every instance anew (de facto the proofs for the instances lead to the discovery of the general theory presented here).

- The improvements in the deduction support and in the automation of the KIV system (without the improvements that result from the use of the modularization theorem) during the course of the case study can be shown clearly for the verification of the first refinement, since it contains 1:1 diagrams only: While in KIV version 1 378 interactions were necessary, this number dropped to 246 in version 3. In KIV version 4 it is now 161.

- Refinement 1/2 also gives a good measure for the the time needed to become familiar with KIV, since the verification in KIV version 3 was done by Harald Vogt, a student that had attended a one-semester practical course on KIV and had no prior knowledge of the case study. It required him 80 hours of work to port the proofs for the refinement from version 1 to version 3 (porting the proofs from version 3 to 4 required about a day of work).
- The size of the interpreters starts with 120 lines of imperative (Pseudo-PASCAL-)code and reaches 300 lines for ASM9. Since it contains a lot of new instructions, ASM10 (nearly identical to the WAM) is much larger with 950 lines of code.

Chapter 20

Related Case Studies

There is a huge amount of research in the literature, that considers compiler correctness in papers. For an overview see e.g. [Joy90]. Large efforts on the topic were e.g. the VLISP ([GRW95]) and the PROCOS ([BLH93]) project.

Most of the work falls (just like our work) in the category, that deals with the correctness of the compilation ("compiling correctness"). The efficient implementation of compilers ("compiler correctness") was treated rarely, but is currently researched in the Verifix project ([GDG+96]).

Work on system supported, formal verification of compilers is much rarer. The most elaborate work in this field is the formal verification of a compiler, that translates code of the imperative programming language Gypsy first to assembler code and then into native machine code of the FM8502 processor ([Moo88], [You88]).

Verification of the compilation of Prolog to the WAM was besides [BR95], on which our work is based, also discussed in [Rus92]. This work makes some simplifications (it does neither consider the cut nor switching), and does not structure the proof into several refinements. An attempt to formalize the proof failed because of its complexity. Therefore V. Austel tried to do a structured proof in [Aus98] with the HOL system ([Gor88]). His proof attempt tries to refine the term representation before the control structure and is in our opinion nearly incomprehensible. The work required one man year of effort, and according to the author at least another year would be necessary to complete it.

The most interesting point in this work is the thesis, that a major problem, that [BR95] only treats insuffiently, is the introduction of the term representation in one single refinement (9/10). Now our consideration in Sect. 18 have shown, that the introduction of term representation (and all other concepts) in a single step must indeed be decomposed into several steps in order to make a clear verification possible. Nevertheless we think, that the decomposition as we currently propose it, will do this, and we do not see any fundamental problems.

Another work done parallel to this one is the formal treatment of the compilation of Prolog to the WAM by C. Pusch ([Pus96]) with the Isabelle system ([Pau94]). Her specification is based on inductively defined relations over the vector of state variables. Using polymorphism and pattern matching makes the notation in Isabelle much more compact (but for an untrained reader also more cryptic) than ASM notation (and even more than our PASCAL-like notation in the translation to DL).

The starting point of her work is based on a definition of an interpreter that already uses stacks of choicepoints, not search trees. Stacks are modeled as lists, in contrast to our pointer structure. This avoids the necessity to collect choicepoints with the procedure STACK#. This results in some simplification for the proofs at the cost, that a pointer structure would have to be introduced (and verified) at latest in ASM8, when the stack of choicepoints and the stack of environments are merged.

Four refinements were verified: the first introduces cutpoints (i.e. positions in the stack). These were represented as sublists of the current stack in the first interpreter. The second refinement shows, that instead of using all clauses as candidates a *procdef* function can be used, that gives all

clauses with the same leading predicate symbol. The ASM that results from the second refinement is (modulo notation) equivalent to our ASM2, and the last two refinements verified in Isabelle are identical to our refinements 2/3 and 3/4 (except that the constructs *true* and *fail* were not considered, therefore the problem discussed in 14.2 we found in the *fail* rule could not be found).

The verification effort for the four refinements is given in [Pus96] as 6 person months and 3500 interactions. The major part of this effort was necessary for the refinements 2/3 and 3/4, as can be seen from the proof scripts. These figures are more than twice the ones we achieved. There are two main reasons for this: First, no proof technique for m:n diagrams, as they appear in 2/3 and 3/4, was developed. Instead, diagrams were decomposed into *1:n* diagrams, as we sketched in Sect. 6.2.3, p. 28. This resulted in a drastic increase of the size of the invariants. Second, two separate, asymmetric proofs were done for correctness and completeness of each refinements. The asymmetry of the proofs seems one hand to be due to the use of abstraction functions, that required additionally the definition of their domain (with *config_ok*), but asymmetrically, not definition of their codomain. On the other hand it is the determinism of the state based system, that is essential for the fact, that only one proof is necessary. In our encoding of the ASM in the calculus of Dynamic Logic determinism is syntactically supported by the axiom

$$\langle\alpha\rangle\ \varphi \equiv [\alpha]\ \varphi$$

for deterministic programs α (this axiom is used in our correctness proof of the modularization theorem). In the formalization of the ASM as an inductive relation a similar axiom has to be proved individually for each α by induction over its structure.

Chapter 21

Summary of Part II

In the second part of this work we have investigated the practical usefulness of the theory developed in part 1. The case study we used for this purpose is from compiler verification. With 9 months of effort for the verification, the case study is a very large one.

The content of the case study was the formal verification of 8 of the 12 refinements given in [BR95], that compile a Prolog program to assembler code of the Warren Abstract Machine. The case study contained a large number of typical problems from compiler verification, e.g. introduction of registers, stacks, environments (stack frames), the optimization of control structures (switching) or the translation of abstract datatypes to pointer structures. These problems should also be relevant for other programming languages.

The case study showed, that due to a large number of implicit assumptions, the fully formal correctness proof of a refinement is much more expensive than one could estimate by looking at the already elaborate mathematical analysis done in [BR95]. The additional effort payed off in the sense, that a number of small errors, that were left open in the mathematical analysis, could be found and removed.

To make the verification of the refinements tractable, the full theory developed in the first part was necessary as well as a very powerful tool for verification. The KIV system, that was used in the case study, has been significantly improved during the work on this case study.

Finally the comparison with two case studies on the same topic done with other systems (HOL, Isabelle) in parallel to this work shows, that the developed theory allowed the necessary effort to be significantly smaller.

Chapter 22

Outlook

The case study done in this work does not yet completely show the correctness of the compilation of Prolog to the WAM. There are still 4 refinements until full WAM code is reached. The first two refinements will be relatively complex to verify, while the other two (environment trimming and removal of the structure of environments and choicepoints) should be easy. Altogether, we estimate the effort to complete the verification to be about 2–3 months.

To get a verified Prolog compiler from the case study, then a compiler could be implemented, that fulfills the compiler assumptions. This should be easy for an simple variant with recursively defined DL programs, since the compiler assumptions are (with the exception of switching) already algorithmic.

More interesting than to use imperative programs for the implementation would be to take up the ideas from the Verifix project [GDG+96] and to use Prolog itself as the implementation language of the compiler. This would give the possibility to get an efficient compiler by compiling the compiler with itself ("bootstrapping").

The definition of a Prolog compiler in Prolog would be a list of clauses for a predicate *compile* with two arguments. A query would be of the form $compile(t, X)$, where t would be a Prolog program encoded as a term. X would be the output variable, whose result value at the end of the computation would be generated WAM instructions, again encoded in a term.

To connect programs and WAM instruction lists to terms ("reflection"), two conversion functions *clauselist-to-term* and *term-to-instructionlist* are necessary. They are easy to define here, since Prolog is an untyped language (the programming language with the simplest reflection principle, namely the "quote" operation, is LISP, since programs and data structures are identical; for typed languages reflection is a much harder problem). Subsequently the Prolog code db_{compile} of the Prolog compiler could be verified, by showing that execution of ASM1 with a query $compile(t, X)$ results for each program 4 (encoded as a term) in a list of instructions, which fulfill the compiler assumptions. Formally, we have to prove

$$\begin{aligned} & \text{t} = \text{clauselist-to-term(db)} \\ \wedge\ & \langle \text{ASM1}(\text{db}_{\text{compile}}, \text{compile(t,X); subst}) \rangle\ \text{subst} = [\text{X} \leftarrow \text{t}'] \\ \rightarrow\ & \text{CompAssum(db, term-to-instructionlist(t}')) \end{aligned}$$

With this approach a compiler would result, whose correctness depends only on the fact, that ASM1 is a correct semantic definition of Prolog, the (trivial) correctness of the conversion functions and of course the correctness of the verification tool.

For the bootstrapping of the compiler with itself (to get a compiler implemented in WAM code) there would be three choices: Either the WAM code could be got by replacing db with db_{compile} in the theorem above and symbolic execution of ASM1. This would be ideal, since then only the correctness of the verification tool would be relevant for correct WAM code. Experience of my colleague Kurt Stenzel with a Java ASM show, that this is very expensive (space and time consuming) and could turn out to be impossible with the resources available. A second possibility

would be to do the bootstrapping with one (or several) available Prolog compilers. A last possibility would be to use the code generation facility of KIV, that generates LISP programs for the abstract programs of ASM1. The resulting code could also be used to do the bootstrapping. The last two methods are from a theoretical viewpoint not quite as safe as the first one, since they require the correctness of another compiler (at least for the considered program of the Prolog compiler), but if both methods result in the same code, the probability of an error should nevertheless be de facto equal to zero.

Appendix A

Used Notations

This section gives the basic notations used in this work.

For a set S we denote with $\mathcal{P}(S)$ the power set of S, with $\mathcal{P}^{\omega}(S)$ the set of all finite subsets of S. S^n is the set of all n-tuples over S ($n \geq 0$). We write $x_1 \ldots x_n$ and $(x_1, \ldots, x_n)$ for n-tuples. If n is clear from the context or arbitrary, we also write $\underline{x}$. S^* is the union of all S^n for $n \geq 0$. This set also contains the empty tuple, written (). S^+ is S^* without the empty tuple. $\hat{S}^n$ is the set of all duplicate free n-tuples: $x_1 \ldots x_n \in S^n$ iff $x_i \neq x_j$ for all $1 \leq i < j \leq n$. $\hat{S}^*$ is the union of all $\hat{S}^n$. We use the notation $M = \bigcup_{s \in S} M_s$ for a family of sets M_s, indexed with the elements of S. It is always assumed that the sets M_s of the family are pairwise disjoint. $M_{s_1 \ldots s_n}$ abbreviates $M_{s_1} \times \ldots \times M_{s_n}$ and $\hat{M}_{s_1 \times \ldots \times s_n}$ is the same as $M_{s_1} \times \ldots \times M_{s_n} \cap \hat{M}^n$. For two tuples $(x_1, \ldots, x_n) \in S^n$ and $(x'_1, \ldots, x'_m) \in S^m$ we define their concatenation $(x_1, \ldots, x_n) : (x'_1, \ldots, x'_m)$ as $(x_1, \ldots, x_n, x'_1, \ldots, x'_m) \in S^{n+m}$. We identify S with S^1, so $x : (x_1, \ldots, x_n)$ is the same as $(x, x_1, \ldots, x_n) \in S^{n+1}$.

If a function $f : M \to N$ is given, then we assume, that the homomorphic extension to a function on tuples from M^n is defined by $\mathrm{f}((x_1, \ldots, x_n)) := (f(x_1), \ldots, f(x_n))$ The homomorphic extension of f to subsets of M is defined analogously.

Appendix B

Syntax and Semantics of Dynamic Logic

B.1 Syntax of Dynamic Logic

Definition 4 *Signatures*
A (many-sorted) signature $SIG = (S, OP, X, P)$ consists of a finite set of sorts S, a family $OP = \bigcup_{\underline{s} \in S^*, s' \in S} OP_{\underline{s},s'}$ of operations (with argument sorts $\underline{s}$ and target sort s'), a family $X = \bigcup_{s \in S} X_s$ of countably many variables for each sort, and a family $P = \bigcup_{\underline{s} \in S^*, \underline{s}' \in S^*} P_{\underline{s},\underline{s}'}$ of procedure names with value parameters of sorts $\underline{s}$ and reference parameters of sorts $\underline{s}'$ (procedure names are used in programs).

It is assumed, that S contains at least the sorts *bool* and *nat*, as well as the usual operations on these sorts (*true*,*false*,$\wedge$,$\vee$,$\rightarrow$, $\leftrightarrow$,$\neg$, 0,+1,$-$1,+).

Definition 5 *DL Expressions*
For a many-sorted signature *SIG*, the set $DLEXPR = \bigcup_{s \in S} DLEXPR_s$ of expressions, and the set $PROG$ of programs are defined to be the smallest sets with

- $X_s \subseteq DLEXPR_s$ for every $s \in S$
- If $f \in OP_{\underline{s},s}$ and $\underline{t} \in DLEXPR_{\underline{s}}$ then $f(\underline{t}) \in DLEXPR_s$
- If $\varphi \in FMA$ and $\underline{x} \in \hat{X}_{\underline{s}}$ then $\forall\, \underline{x}.\varphi \in FMA$
- If $\varphi \in FMA$ and $\underline{x} \in \hat{X}_{\underline{s}}$ then $\exists\, \underline{x}.\varphi \in FMA$
- If $t, t' \in DLEXPR_s$, then $t = t' \in FMA$
- If $\varphi \in FMA$ and $t, t' \in DLEXPR_s$, then $(\varphi \supset t; t') \in DLEXPR_s$
- If $\underline{x} \in \hat{X}_s$ and $\underline{t} \in U_{\underline{s}}$, where $U_s = T_s \cup \{?\}$, then $\underline{x} := \underline{t} \in PROG$
- If $\alpha \in PROG$, $\underline{x} \in \hat{X}_s$ and $\underline{t} \in U_{\underline{s}}$, where $U_s = T_s \cup \{?\}$, then **var** $\underline{x} = \underline{t}$ **in** $\alpha \in PROG$
- **skip**, **abort** $\in PROG$
- If $\alpha, \beta \in PROG$, then $\alpha; \beta \in PROG$
- If $\alpha, \beta \in PROG$ and $\varepsilon \in BXP$, then **if** ε **then** α **else** $\beta \in PROG$
- If $\alpha \in PROG$ and $\varepsilon \in BXP$ then **while** ε **do** $\alpha \in PROG$
- If $\alpha \in PROG$ and $\kappa \in T_{nat}$ then **loop** α **times** $\kappa \in PROG$

- If $p \in P_{\underline{s},\underline{s'}}$, $\underline{t} \in T_{\underline{s}}$, $\underline{x} \in \hat{X}_{\underline{s'}}$ and $\kappa \in T_{\mathrm{nat}}$ then $\mathrm{p}(\underline{t};\underline{x}) \in PROG$ and **procbound** κ **in** $\mathrm{p}(\underline{t};\underline{x}) \in PROG$. The latter program is a call to p with maximal recursion depth bounded by κ.

The definition uses *FMA* (formulas) to abbreviate $DLEXPR_{\mathrm{bool}}$. The set T_s (Terms of sort s) is the subset of $DLEXPR_s$, that does neither contain quantifiers nor programs. *BXP* (boolean expressions) is T_{bool}.

Remark 1 Like in Pascal we use **begin** ... **end** as brackets around programs. **if** ε **then** α is used as an abbreviation for **if** ε **then** α **else skip**.

Remark 2 The tests of while loops and conditionals must be boolean expressions in the definition above ($\varepsilon \in BXP$). This is necessary for application programs. For proof obligations and in proofs it is sometimes convenient to use arbitrary formulas. This extension does not pose any problems, everything that follows holds for arbitrary $\varepsilon \in FMA$ too.

Definition 6 *Assigned Variables*
The set $asgv(\alpha)$ of assigned variables in a programs α is defined by:

- $\mathrm{asgv}(\textbf{skip}) = \mathrm{asgv}(\textbf{abort}) = \emptyset$
- $\mathrm{asgv}(\alpha;\beta) = \mathrm{asgv}(\alpha) \cup \mathrm{asgv}(\beta)$
- $\mathrm{asgv}(\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta) = \mathrm{asgv}(\alpha) \cup \mathrm{asgv}(\beta)$
- $\mathrm{asgv}(\textbf{while } \varepsilon \textbf{ do } \alpha) = \mathrm{asgv}(\alpha)$
- $\mathrm{asgv}(\textbf{loop } \alpha \textbf{ times } \kappa) = \mathrm{asgv}(\alpha)$
- $\mathrm{asgv}(\textbf{var } \underline{x} = \underline{t} \textbf{ in } \alpha) = \mathrm{asgv}(\alpha) \setminus \underline{x}$
- $\mathrm{asgv}(p(\underline{t};\underline{x})) = \underline{x}$
- $\mathrm{asgv}(\textbf{procbound } p(\underline{t};\underline{x}) \textbf{ times } \kappa) = \underline{\mathrm{x}}$

Definition 7 *Called Procedures*
calledprocs(α) is the set of all procedures that are called in a program α.

Definition 8 *Procedure Declarations and Procedur Declaration Lists*
The setPD of procedure declarations is the set of all $p(\underline{x};\textbf{var } \underline{y}).\alpha$ with $p \in P_{\underline{s},\underline{s'}}$ $\underline{x},\underline{y} \in \hat{X}_{\underline{s},\underline{s'}}$, $\alpha \in PROG$ and $asgv(\alpha) \subseteq \underline{\mathrm{x}} \cup \underline{\mathrm{y}}$. α must not contain procedure calls with bounded recursion depth. p is the procedure defined with the procedure declaration, α is the body of the procedure.

PDL is the set of all lists of Procedure declarations, such that the called procedures in their bodies are a subset of the set of all defined procedures.

B.2 Semantics of Dynamic Logic

Definition 9 *Algebra*
An Algebra $\mathcal{A}$ over a signature *SIG* consists of a nonempty carrier set A_s for every sort s and a function $f_{\mathcal{A}} : A_{\underline{s}} \to A_{s'}$ for every $f \in OP_{\underline{s},s'}$. For every procedure $p \in P_{\underline{s},\underline{s'}}$ and every $n \in \mathbb{N}$ the algebra $\mathcal{A}$ contains a relation $[\![p]\!]_{\mathcal{A},n}$ on $A_{\underline{s},\underline{s'},\underline{s'}}$. (which is the semantics of p when the maximal recursion depth is bounded by n). $[\![p]\!]_{\mathcal{A},0}$ must be the empty relation. $[\![p]\!]_{\mathcal{A}}$ denotes the semantics of the procedure and is defined as the union of all $[\![p]\!]_{\mathcal{A},n}$. The semantics defines a relation between the initial values of value and reference parameters and the result values of the reference parameters.

It is assumed that $A_{\mathrm{bool}} = \{tt,ff\}$, $A_{\mathrm{nat}} = \mathbb{N}$, and that the operationen on booleans and natural numbers have their usual semantics.

Definition 10 *States/Valuations*
For a signature SIG and an algebra $\mathcal{A}$ over this signature a state (or synonymously, a valuation) $\mathbf{z} \in ST_{\mathcal{A}}$ is defined as a function, that maps the variables of sort s to values in A_s. The state $\mathbf{z}[\underline{x} \leftarrow \underline{a}]$ is the state, which results from $\mathbf{z}$ by modifying the values at variables $\underline{x}$ with $\underline{a}$.

Definition 11 *Semantics of Expressions*
For an algebra $\mathcal{A}$ and a valuation $\mathbf{z}$ the semantics $[\![e]\!]_{\mathbf{z}} \in A_s$ of a DL expresion $e \in DLEXPR_s$, and the semantics $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ of a program (a relation on states, written infix) are defined by:

- $[\![x]\!]_{\mathbf{z}} = \mathbf{z}(x)$
- $[\![f(\underline{t})]\!]_{\mathbf{z}} = f_{\mathcal{A}}([\![\underline{t}]\!]_{\mathbf{z}})$ for $f \in OP_{\underline{s},s'}$ and $\underline{t} \in T_{\underline{s}}$
- $[\![\forall\ \underline{x}.e]\!]_{\mathbf{z}} = tt$ with $\underline{x} \in \hat{X}_{\underline{s}}$ iff $[\![e]\!]_{\mathbf{z}[\underline{x} \leftarrow \underline{a}]} = tt$ for all values $\underline{a} \in \mathcal{A}_{\underline{s}}$
- $[\![\exists\ \underline{x}.e]\!]_{\mathbf{z}} = tt$ with $\underline{x} \in \hat{X}_{\underline{s}}$ iff $[\![e]\!]_{\mathbf{z}[\underline{x} \leftarrow \underline{a}]} = tt$ for some values $\underline{a} \in \mathcal{A}_{\underline{s}}$
- $[\![(\varepsilon \supset e; e')]\!]_{\mathbf{z}}$ is $[\![e]\!]_{\mathbf{z}}$, if $[\![\varepsilon]\!]_{\mathbf{z}} = tt$, and $[\![e']\!]_{\mathbf{z}}$ otherwise.
- $\mathbf{z}[\![\mathbf{skip}]\!]\mathbf{z}'$ iff $\mathbf{z} = \mathbf{z}'$
- $[\![\mathbf{abort}]\!]$ is the empty relation
- $\mathbf{z}[\![\underline{x}{:=}\underline{t}]\!]\mathbf{z}'$ iff $\mathbf{z}' = \mathbf{z}[\underline{x} \leftarrow [\![\underline{t}]\!]_{\mathbf{z}}]$, where each $[\![?]\!]_{\mathbf{z}}$ is some arbitrary value.
- $\mathbf{z}[\![\alpha; \beta]\!]\mathbf{z}'$ iff there is a $\mathbf{z}''$ with $\mathbf{z}[\![\alpha]\!]\mathbf{z}''$ and $\mathbf{z}''[\![\beta]\!]\mathbf{z}'$
- $\mathbf{z}[\![\mathbf{if}\ \varepsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta\ ]\!]\mathbf{z}'$ iff
  either $[\![\varepsilon]\!]_{\mathbf{z}} = tt$ and $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ or $[\![\varepsilon]\!]_{\mathbf{z}} = ff$ and $\mathbf{z}[\![\beta]\!]\mathbf{z}'$
- $\mathbf{z}[\![\mathbf{loop}\ \alpha\ \mathbf{times}\ \kappa]\!]\mathbf{z}'$ iff
  there are states $\mathbf{z}_0 := \mathbf{z}, \mathbf{z}_1, \ldots, \mathbf{z}_n := \mathbf{z}'$ with $n := [\![\kappa]\!]_{\mathbf{z}}$ such that
  $\mathbf{z}_{i-1}[\![\alpha]\!]\mathbf{z}_i$ for every $1 \leq i \leq n$
- $\mathbf{z}[\![\mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha]\!]\mathbf{z}'$ iff
  there are states $\mathbf{z}_0 := \mathbf{z}, \mathbf{z}_1, \ldots, \mathbf{z}_n := \mathbf{z}'$ with
  $\mathbf{z}_{i-1}[\![\alpha]\!]\mathbf{z}_i$ for $1 \leq i \leq n$,
  $[\![\varepsilon]\!]_{\mathbf{z}_i} = tt$ for $1 \leq i < n$ and $[\![\varepsilon]\!]_{\mathbf{z}'} = ff$
- $\mathbf{z}[\![\mathbf{var}\ \underline{x} = \underline{t}\ \mathbf{in}\ \alpha]\!]\mathbf{z}'$ iff $\mathbf{z}[\underline{x} \leftarrow \underline{a}][\![\alpha]\!]\mathbf{z}''$ and $\mathbf{z}' = \mathbf{z}''[\underline{x} \leftarrow [\![\underline{x}]\!]_{\mathbf{z}}]$ where $a_i = [\![t_i]\!]$ for $t_i \neq ?$ and otherwise a_i is arbitrary.
- $\mathbf{z}[\![p(\underline{t}, \underline{x})]\!]\mathbf{z}'$ iff $\mathbf{z}(\underline{t}), \mathbf{z}(\underline{x}), \mathbf{z}'(\underline{x}) \in [\![p]\!]$ and $\mathbf{z}(y) = \mathbf{z}'(y)$ for all $y \notin \underline{x}$
- $\mathbf{z}[\![\mathbf{procbound}\ \kappa\ \mathbf{in}\ p(\underline{t}, \underline{x})]\!]\mathbf{z}'$ iff $\mathbf{z}(\underline{t}), \mathbf{z}(\underline{x}), \mathbf{z}'(\underline{x}) \in [\![p]\!]_n$, where $n = [\![\kappa]\!]_{\mathbf{z}}$, and $\mathbf{z}(y) = \mathbf{z}'(y)$ for all $y \notin \underline{x}$
- $[\![\langle\alpha\rangle\ \varphi]\!]_{\mathbf{z}} = tt$ iff there is a $\mathbf{z}'$ with $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ and $[\![\varphi]\!]_{\mathbf{z}'} = tt$
- $[\![[\alpha]\ \varphi]\!]_{\mathbf{z}} = tt$ iff for all $\mathbf{z}'$ with $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$: $[\![\varphi]\!]_{\mathbf{z}'} = tt$

Remark 3 The semantics of expressions and programs is defined unambiguously, since each case reduces the number of elementary statements in the expression/program considered.

Definition 12 *Semantics of Procedure Declaration Lists*
If δ is a procedure declaration list, then $\mathcal{A} \models \delta$ iff for every procedure declaration $p(\underline{x}; \underline{y}).\alpha$ contained in δ and every $\kappa = 0 + 1 \ldots + 1$ (representing a number $n \geq 0$) the following property holds:

$$[\![\mathbf{procbound}\ \kappa + 1\ \mathbf{in}\ p(\underline{x}; \underline{y})]\!] = [\![\mathbf{procbound}\ \kappa\ \mathbf{in}\ \alpha]\!]$$

In the definition **procbound** κ **in** α is the program, that results from replacing each procedure call $q(\underline{\sigma};\underline{z})$ in α by **procbound** κ **in** $q(\underline{\sigma};\underline{z})$ (for every procedure name q).

Remark 4 A procedure declaration list unambiguously fixes the semantics of the defined procedures. The proof is by induction on n, that $[\![p]\!]_{\mathcal{A},n}$ is fixed. It is also easy to show, that the $[\![p]\!]_{\mathcal{A},n}$ are monotone increasing relations for the defined procedures.

Definition 13 *models operator*

- $\mathcal{A}, \mathbf{z} \models \varphi$ holds (or is valid) for a formula φ iff $[\![\varphi]\!]_{\mathbf{z}} = tt$
- $\mathcal{A} \models \varphi$ holds iff for all states $\mathbf{z}$: $\mathcal{A}, \mathbf{z} \models \varphi$
- $\models \varphi$ holds off for every algebra $\mathcal{A}$: $\mathcal{A} \models \varphi$
- $\Phi \models \psi$ holds iff for every algebra $\mathcal{A}$: from $\mathcal{A} \models \varphi$ for every $\varphi \in \Phi$ follows $\mathcal{A} \models \psi$.

Remark 5 The following properties are valid, if i does neither occur in α nor in ε. The first two properties characterize while loops (they allow induction over the number of iterations). The third property allows to avoid loops with a counter occurring in α.

- $\models \langle$**while** ε **do** $\alpha\rangle\ \varphi \leftrightarrow \exists\ i.\langle$**loop if** ε **then** α **times** $i\rangle\ (\varphi \wedge \neg\ \varepsilon)$
- $\models \langle$**loop** α **times** $\kappa + 1\rangle\ \varphi \leftrightarrow \langle\alpha;$ **loop** α **times** $\kappa\rangle\ \varphi$
- $\models \langle$**loop** α **times** $\kappa\rangle\ \varphi \leftrightarrow (\forall\ i.i = \kappa \rightarrow \langle$**loop** α **times** $i\rangle\ \varphi)$

Remark 6 Let $\mathcal{A}$ be an algebra with $\mathcal{A} \models \delta$ for a procedure declaration list δ, that contains a procedure declaration $p(\underline{x};$ **var** $\underline{y}).\alpha$. Then the following three formulas characterize the recursive procedure (i.e. their validity is equivalent to the procedure declaration). Procedure declaration lists therefore can be viewed as abbreviations for axioms. The formulas allow to induce over the recursion depth and unfolding of procedures. The first formulas holds in every algebra. In the third formula $\underline{x}_0$ and $\underline{y}_0$ have to be new variables of the same sorts as $\underline{x}$ and $\underline{y}$. **procbound** κ **in** α again is the program, that is derived from α by replacing all procedure calls $q(\underline{\sigma};\underline{z})$ with **procbound** κ **in** $q(\underline{\sigma};\underline{z})$.

- $\models \langle p(\underline{t};\underline{z})\rangle\ \varphi \leftrightarrow \exists\ \kappa.\langle$**procbound** κ **in** $p(\underline{t};\underline{z})\rangle\ \varphi$
- $\mathcal{A} \models \langle$**procbound** $\kappa + 1$ **in** $p(\underline{t};\underline{z})\rangle\ \varphi \leftrightarrow$
  $\langle \underline{x}_0, \underline{y}_0, \underline{x}, \underline{y} := \underline{x}, \underline{y}, \underline{t}, \underline{z};$ **procbound** κ **in** $\alpha; \underline{x}, \underline{y}, \underline{y}_0 := \underline{x}_0, \underline{y}_0, \underline{y}; \underline{z} := \underline{y}_0\rangle\ \varphi$
- $\mathcal{A} \models \langle p(\underline{t};\underline{z})\rangle\ \varphi$
  $\leftrightarrow \langle \underline{x}_0, \underline{y}_0, \underline{x}, \underline{y} := \underline{x}, \underline{y}, \underline{t}, \underline{z}; \alpha; \underline{x}, \underline{y}, \underline{y}_0 := \underline{x}_0, \underline{y}_0, \underline{y}; \underline{z} := \underline{y}_0\rangle\ \varphi$

Definition 14 *(Basic) Specifications*
A basic specification *SPEC* = (*SIG,Ax,GAx,PAx*) consists of

- a signature *SIG* = (*S,OP,P,X*).
- a set of axioms *Ax* (formulas over *SIG*).
- a set *GAx* of generation clauses of the form: $s_1, \ldots s_n$ **generated by** $f_1, \ldots f_m$ (n,m > 0). It is required that $s_1, \ldots s_n \in \hat{S}^*$ and all f_j have a target sort in $s_1, \ldots s_n$.
- a set *PAx* of procedure declaration lists over *SIG*. If a procedure is declared in several lists, the declarations must be identical.

Definition 15 *Semantics of Specifications*
An algebra $\mathcal{A}$ is a model of *SPEC* (written as $\mathcal{A} \models$ *SPEC*, if it is an algebra over the signature of the specification with

- $\mathcal{A} \models \varphi$ for every $\varphi \in Ax$
- For every generation clause $s_1, \ldots, s_n$ **generated by** $f_1, \ldots, f_m \in GAx$ and every $i = 1 \ldots n$, every element $a \in \mathcal{A}_{s_i}$ can be got as the semantics $a = [\![t]\!]_{\mathbf{z}}$ of some term t under some values for $\mathbf{z}$. The term must not contain variables of the sorts $s_1, \ldots s_n$, and that contains operation symbols only from $\{f_1, \ldots, f_m\}$.
- $\mathcal{A} \models \delta$ for every $\delta \in PAx$

Remark 7 For every model of a specification (*SIG,Ax,GAx,*$\emptyset$) with no procedure names in its signature, there is exactly one extension to a model (*SIG* $\cup$ *P,Ax,GAx, PAx*), where P is the set of defined procedures in *PAx*.

Remark 8 We write $SPEC \models \varphi$, iff in every model $\mathcal{A}$ of *SPEC* $\mathcal{A} \models \varphi$ holds.

Theorem 11 *Correctness and Completeness*
The theory of basic specifications can be axiomatized correctly and completely, if we add for every generation clause $s_1, \ldots s_n$ **generated by** $f_1, \ldots, f_m$ an Omega rule: If for a formula $\varphi(x)$ containing a free variable x from one of the sorts $s_1, \ldots s_n$ all (evtl. infinite many) formulas $\varphi(t)$ with terms t, which are built up with the constructors $f_1, \ldots f_m$ and only contain variables from sorts not in $s_1, \ldots s_n$ can be derived, then $\forall\, x.\varphi(x)$ can be derived.

The rule has infinitely many premises, so it cannot be used in a theorem prover. In the implementation of a calculus Omega rules are replaced by structural induction principles. These are theoretically weaker than the Omega rules but sufficient for practical application.

We do not want to prove the theorem here. The idea of the proof is to translate all DL formulas into equivalent first-order formulas. To do this we translate every program α into a relation R_α (input: all variables of the program, output: all assigned variables of the program) This reduces the correctness and completeness proofs to first-order specifications with generation clauses. For these it is known that they can be correctly and completely axiomatized with an Omega rule (see [Rei98]).

Appendix C

Specifications and Lemmas for the Modularization Theorem

C.1 General Specifications

Specifications for natural numbers, lists and dynamic functions can be found in appendix E.

diagtype =
data specification
    diagtype = mn | 0n | m0;
    variables c: diagtype;
end data specification

---

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

---

f-state-state =
actualize Dynfun **with parameter** state **by morphism**
  dom → state, codom → state, dynfun → f-state-state,
  .[.] → . ⌈ . ⌉$_s$
end actualize

---

iterate =
enrich nat, f-state-state **with**
    functions . ^ . : f-state-state × nat → f-state-state **prio** 9;

axioms

    it-base-ax : (f ^ 0)⌈st⌉$_s$ = st,
    it-rec-ax : (f ^ m +1)⌈st⌉$_s$ = f⌈(f ^ m)⌈st⌉$_s$⌉$_s$

end enrich

---

stream =
actualize Dynfun **with** nat, **parameter** state **by morphism**
dom → nat, codom → state, dynfun → stream,
. [.] → . ⌈ . ⌉, f → s,
end actualize

---

enrstream =
enrich stream, iterate **with**
functions

cons	:	state × stream	→	stream;
cdr	:	stream	→	stream;
app	:	stream × nat × stream	→	stream;
nthcdr	:	stream × nat	→	stream;

axioms

cons-base-ax : cons(st, s)⌈0⌉ = st,
cons-rec-ax : cons(st, s)⌈m +1⌉ = s⌈m⌉,
cdr-ax : cdr(s)⌈m⌉ = s⌈m +1⌉,
app-base-ax : app(s, 0, s_0) = s_0,
app-rec-ax : app(s, m +1, s_0) = cons(s⌈0⌉, app(cdr(s), m, s_0)),
nthcdr-base-ax : nthcdr(s, 0) = s,
nthcdr-rec-ax : nthcdr(s, m +1) = nthcdr(cdr(s), m),
streamchoice :
(∀ m. ∃ st_1. st_1 = (f ^ m)⌈st_0⌉$_s$)
→ (∃ s. ∀ m. s⌈m⌉ = (f ^ m)⌈st_0⌉$_s$)

end enrich

---

tuple =
data specification
using enrstream
tuple = mkt (. .s : stream, . .i : nat, . .j : nat);
variables t_1, t_0, t: tuple;
end data specification

---

f-tup-tup =
actualize iterate **with** tuple **by morphism**
state → tuple, f-state-state → f-tup-tup, . ⌈ . ⌉$_s$ → . ⟦ . ⟧,
st → t, f → ft
end actualize

---

rule =
enrich enrstream **with**
predicates

Trace	:	stream;
final	:	state;

procedures
RULE : → state; (: arbitrary procedure as ASM rule :)

axioms

Trace-def :
  Trace(s)
↔ (∀ m, st. st = s⌈m⌉
  → ⟨**if** ¬ final(st) **then** RULE(; st)⟩ st = s⌈m +1⌉),

final-def : (: rule does not terminate −→ final state :)
(¬ ⟨RULE(; st)⟩ true) → final(st) ,

choice : (: choice axiom for RULE :)
  (∀ st. ⟨**if** ¬ final(st) **then** RULE(; st)⟩ true)
→ ∃ f. ∀ st_0. ⟨st := st_0; **if** ¬ final(st) **then** RULE(; st)⟩ st = f⌈st_0⌉$_s$

end enrich

---

rule' =
rename rule **by morphism**
  stream → stream', state → state', . ⌈ . ⌉ → . ⌈ . ⌉', cons → cons',
  cdr → cdr', app → app', nthcdr → nthcdr', Trace → Trace',
  final → final', RULE → RULE', s → s', st → st'
end rename

C.2 Refinement of Deterministic ASMs

C.2.1 Specification

detequiv =
enrich rule, rule', diagtype **with**
  functions
    ndt : state × state' → diagtype;
    exec0n : state × state' → nat;
    execm0 : state × state' → nat;
  predicates
    INV : state × state'; (: coupling invariant :)
    IN : state × state'; (: input relation :)
    OUT : state × state'; (: output relation :)
    PROP : state × state';
  variables i, j, k: nat;

axioms
init-ax : IN(st, st') → INV(st, st'),
finboth-ax : final(st) ∧ final'(st') ∧ INV(st, st') → OUT(st, st'),
fin1-ax : final(st) ∧ INV(st, st') ∧ ¬ final'(st') → ndt(st, st') = 0n,
fin2-ax : final'(st') ∧ INV(st, st') ∧ ¬ final(st) → ndt(st, st') = m0,

mton-ax :
  INV(st, st') ∧ ¬ final(st) ∧ ¬ final'(st') ∧ ndt(st, st') = m0
→ ⟨**if** ¬ final(st) **then** RULE(; st) ⟩
  ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
    ⟨**if** ¬ final'(st') **then** RULE'(; st') ⟩
      ∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j ⟩ INV(st, st'),

0ton-ax :
  INV(st, st') ∧ ¬ final'(st') ∧ ndt(st, st') = 0n ∧ exec0n(st, st') = k

→ ⟨**if** ¬ final'(st') **then** RULE'(; st') ⟩
  ∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩
    (INV(st, st') ∧ (¬ final'(st') ∧ ndt(st, st') = 0n → exec0n(st, st') < k)),

mto0-ax :
  INV(st, st') ∧ ¬ final(st) ∧ ndt(st, st') = m0 ∧ execm0(st, st') = k
→ ⟨**if** ¬ final(st) **then** RULE(; st) ⟩
  ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
    (INV(st, st') ∧ (¬ final(st) ∧ ndt(st, st') = m0 → execm0(st, st') < k)),

prop-def :
  PROP(st, st')
↔ ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
    ∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩ INV(st, st')
end enrich

C.2.2 Proved Theorems

finite-0ton (the main case of lemma 2 from Sect. 6.2.3)

  INV(st, st'), ndt(st, st') = 0n, ¬ final'(st')
⊢ ⟨**if** ¬ final'(st') **then** RULE'(; st') ⟩
  ∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩
    (INV(st, st') ∧ (final'(st') ∨ ndt(st, st') ≠ 0n))

- used lemmas : 0ton-ax
- used by : compl-step, completeness

finite-mto0

  INV(st, st'), ndt(st, st') = m0, ¬ final(st)
⊢ ⟨**if** ¬ final(st) **then** RULE(; st) ⟩
  ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
    (INV(st, st') ∧ (final(st) ∨ ndt(st, st') ≠ m0))

- used lemmas : mto0-ax
- used by : corr-step, correctness

corr-step (Lemma 1 from Sect. 6.2.3)

PROP(st, st') ⊢ ⟨**if** ¬ final'(st') **then** RULE'(; st')⟩ PROP(st, st')

- used lemmas : finite-mto0, fin1-ax, 0ton-ax, mton-ax, prop-def
- used by : correctness

compl-step

PROP(st, st') ⊢ ⟨**if** ¬ final(st) **then** RULE(; st)⟩ PROP(st, st')

- used lemmas : finite-0ton, fin2-ax, mto0-ax, mton-ax, prop-def

- used by : completeness

correctness (correctness of the refinement)

IN(st, st')
⊢ [**while** ¬ final'(st') **do** RULE'(; st')]
⟨**while** ¬ final(st) **do** RULE(; st)⟩ OUT(st, st')

- used lemmas : finboth-ax, fin2-ax, finite-mto0, corr-step, init-ax, prop-def

completeness (completeness of the refinement)

IN(st, st')
⊢ [**while** ¬ final(st) **do** RULE(; st)]
⟨**while** ¬ final'(st') **do** RULE'(; st')⟩ OUT(st, st')

- used lemmas : finboth-ax, fin1-ax, finite-0ton, compl-step, init-ax, prop-def

C.3 Refinement of Indeterministic ASMs – Diagrams of Indeterministic Size

C.3.1 Specification

genindeteqtrace =
enrich rule, rule', f-tup-tup, diagtype **with**
functions

ndt	:	state × state'	→	diagtype;
exec0n	:	state × state'	→	nat;
execm0	:	state × state'	→	nat;

predicates

INV	:	state × state';
INV'	:	state × state';
KPROP	:	state × state';
VPROP	:	state × state';
IN,	:	state × state';
OUT	:	state × state';
p	:	stream' × tuple × tuple;

variables i, i_0, i_1, j, j_0, k: nat;

axioms

init-ax : IN(st, st') → INV(st, st'),
finboth-ax : final(st) ∧ final'(st') ∧ INV(st, st') → OUT(st, st'),
fin1-ax : final(st) ∧ INV(st, st') ∧ ¬ final'(st') → ndt(st, st') = 0n,
fin2-ax : final'(st') ∧ INV(st, st') ∧ ¬ final(st) → ndt(st, st') = m0,

mton-corr-ax :
INV(st, st') ∧ ndt(st, st') = m0 ∧ Trace'(s')
∧ st' = s'⌈0⌉' ∧ ¬ final(st) ∧ ¬ final'(st')
→ ⟨**if** ¬ final(st) **then** RULE(; st)⟩

$\exists$ j. $\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i $\rangle$ INV(st, s'$\lceil$j +1$\rceil'$),

0ton-corr-ax :
INV(st, st') $\wedge$ ndt(st, st') = 0n $\wedge$ exec0n(st, st') = k
$\wedge$ Trace'(s') $\wedge$ st' = s'$\lceil$0$\rceil'$ $\wedge$ $\neg$ final'(st')
$\rightarrow$ $\exists$ j. INV(st, s'$\lceil$j +1$\rceil'$)
$\wedge$ ($\neg$ final'(s'$\lceil$j +1$\rceil'$) $\wedge$ ndt(st, s'$\lceil$j +1$\rceil'$) = 0n $\rightarrow$ exec0n(st, s'$\lceil$j +1$\rceil'$) < k),

mto0-corr-ax : (: follows from mto0-comp-ax, is sufficient for trace correctness :)
INV(st, st') $\wedge$ ndt(st, st') = m0 $\wedge$ execm0(st, st') = k $\wedge$ $\neg$ final(st)
$\rightarrow$ $\langle$**if** $\neg$ final(st) **then** RULE(; st)$\rangle$
$\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$
(INV(st, st') $\wedge$ ($\neg$ final(st) $\wedge$ ndt(st, st') = m0 $\rightarrow$ execm0(st, st') < k)),

mton-comp-ax :
INV(st, st') $\wedge$ ndt(st, st') = m0 $\wedge$ Trace(s)
$\wedge$ st = s$\lceil$0$\rceil$ $\wedge$ $\neg$ final(st) $\wedge$ $\neg$ final'(st')
$\rightarrow$ $\langle$**if** $\neg$ final'(st') **then** RULE'(; st')$\rangle$
$\exists$ i. $\exists$ j. $\langle$**loop if** $\neg$ final'(st') **then** RULE'(; st') **times** j $\rangle$ INV(s$\lceil$i +1$\rceil$, st'),

mto0-comp-ax :
INV(st, st') $\wedge$ ndt(st, st') = m0 $\wedge$ execm0(st, st') = k $\wedge$ Trace(s)
$\wedge$ st = s$\lceil$0$\rceil$ $\wedge$ $\neg$ final(st)
$\rightarrow$ $\exists$ i. INV(s$\lceil$i +1$\rceil$, st')
$\wedge$ ($\neg$ final(s$\lceil$i +1$\rceil$) $\wedge$ ndt(s$\lceil$i +1$\rceil$, st') = m0 $\rightarrow$ execm0(s$\lceil$i +1$\rceil$, st') < k),

0ton-comp-ax : (: follows from 0ton-corr-ax :)
INV(st, st') $\wedge$ ndt(st, st') = 0n $\wedge$ exec0n(st, st') = k $\wedge$ $\neg$ final'(st')
$\rightarrow$ $\langle$**if** $\neg$ final'(st') **then** RULE'(; st')$\rangle$
$\exists$ j. $\langle$**loop if** $\neg$ final'(st') **then** RULE'(; st') **times** j$\rangle$
(INV(st, st') $\wedge$ ($\neg$ final'(st') $\wedge$ ndt(st, st') = 0n $\rightarrow$ exec0n(st, st') < k)),

choice-ax : (: axiom of choice :)
($\forall$ t. $\exists$ t_0. p(s', t, t_0)) $\rightarrow$ ($\exists$ ft. $\forall$ t. p(s', t, ft$[\![$t$]\!]$)),

diagonal-ax : (: axiom of choice :)
$\forall$ m. $\exists$ st. st = (ft $\uparrow$ m)$[\![$mkt(s_0, 0, 0)
$\rightarrow$ $\exists$ s. $\forall$ m. s$\lceil$m$\rceil$ = ft $\uparrow$ m$[\![$mkt(s_0, 0, 0)$]\!]$.s$\lceil$m$\rceil$,

kprop-def :
KPROP(st, st')
$\leftrightarrow$ $\forall$ s'. st' = s'$\lceil$0$\rceil'$ $\wedge$ Trace'(s')
$\rightarrow$ $\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$
($\exists$ m. INV(st, s'$\lceil$m$\rceil'$)),

vprop-def :
VPROP(st, st')
$\leftrightarrow$ $\forall$ s. st = s$\lceil$0$\rceil$ $\wedge$ Trace(s)
$\rightarrow$ $\exists$ j. $\langle$**loop if** $\neg$ final'(st') **then** RULE'(; st') **times** j$\rangle$
($\exists$ m. INV(s$\lceil$m$\rceil$, st')),

inv'-def : INV'(st, st') $\leftrightarrow$ INV(st, st') $\wedge$ (final(st) $\leftrightarrow$ final'(st'))

p-def : (: predicate that describes adding diagrams :)
p(s', t, t_0)
$\leftrightarrow$ INV'(t.s⌈t.i⌉, s'⌈t.j⌉′) ∧ Trace(t.s) ∧ Trace'(s')
$\rightarrow$ Trace(t_0.s) ∧ (∀ i_1. ¬ t.i < i_1 → t.s⌈i_1⌉ = t_0.s⌈i_1⌉)
∧ t.i < t_0.i ∧ t.j < t_0.j ∧ INV'(t_0.s⌈t_0.i⌉, s'⌈t_0.j⌉′),

end enrich

C.3.2 Proved Theorems

fin-0ton

INV(st, st'), ndt(st, st') = 0n, ¬ final'(st')
⊢ ⟨**if** ¬ final'(st') **then** RULE′(; st')⟩
∃ j. ⟨**loop if** ¬ final'(st') **then** RULE′(; st') **times** j⟩
(INV(st, st') ∧ (final'(st') ∨ ndt(st, st') ≠ 0n))

- used lemmas : 0ton-comp-ax
- used by : compl-step, completeness

fin-mto0

INV(st, st'), ndt(st, st') = m0, ¬ final(st)
⊢ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(INV(st, st') ∧ (final(st) ∨ ndt(st, st') ≠ m0))

- used lemmas : mto0-corr-ax
- used by : add-diagram, corr-step, correctness, equiv-final

finite-0ton

ndt(st, st') = 0n, INV(st, st'), Trace'(s'), s'⌈0⌉′ = st', ¬ final'(st')
⊢ ∃ j. INV(st, s'⌈j +1⌉′) ∧ (final'(s'⌈j +1⌉′) ∨ ndt(st, s'⌈j +1⌉′) ≠ 0n)

- used lemmas : 0ton-corr-ax
- used by : add-diagram, equiv-final

corr-step

KPROP(st, st') ⊢ [**if** ¬ final'(st') **then** RULE′(; st')] KPROP(st, st')

- used lemmas : fin-mto0, fin1-ax, 0ton-corr-ax, mton-corr-ax, kprop-def
- used by : correctness

compl-step

VPROP(st, st') ⊢ [**if** ¬ final(st) **then** RULE(; st)] VPROP(st, st')

- used lemmas : fin-0ton, fin2-ax, mto0-comp-ax, mton-comp-ax, vprop-def
- used by : completeness

correctness (correctness of the refinement)

IN(st, st')
⊢ [**while** ¬ final'(st') **do** RULE′(; st')]
⟨**while** ¬ final(st) **do** RULE(; st)⟩ OUT(st, st')

- used lemmas : fin-mto0, finboth-ax, fin2-ax, corr-step, init-ax, kprop-def

completeness (completeness of the refinement)

IN(st, st')
⊢ [**while** ¬ final(st) **do** RULE(; st)]
⟨**while** ¬ final'(st') **do** RULE′(; st')⟩ OUT(st, st')

- used lemmas : finboth-ax, fin1-ax, fin-0ton, compl-step, init-ax, vprop-def

equiv-final (Lemma 3 from Sect. 6.3)

INV(st, st'), Trace'(s'), s'⌈0⌉′ = st'
⊢ ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩ (∃ j. INV'(st, s'⌈j⌉′))

- used lemmas : fin2-ax, fin-mto0, fin1-ax, finite-0ton, inv'-def
- used by : add-diagram

add-diagram (Lemma 4 from Sect. 6.3)

INV'(st, st'), Trace'(s'), s'⌈0⌉′ = st'
⊢ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(∃ j. INV'(st, s'⌈j +1⌉′))

- used lemmas : fin1-ax, 0ton-corr-ax, fin-mto0, fin2-ax, mto0-corr-ax, finite-0ton, equiv-final, mton-corr-ax, inv'-def
- used by : totality

totality (Totality of the relation that describes adding diagrams)

⊢ ∀ s', t. ∃ t_0. p(s', t, t_0)

- used lemmas : inv'-def, p-def, add-diagram
- used by : choice-concl, ind-choice-concl

choice-concl (existence of a function, that adds a diagram)

⊢ ∃ ft. p(s', t, ft⟦t⟧)

- used lemmas : totality, choice-ax

ind-choice-concl (special case of *choice-concl* for *ft* ↑ *m*)

⊢ ∃ ft. ∀ m. p(s', (ft ↑ m)⟦t⟧, (ft ↑ m +1)⟦t⟧)

- used lemmas : totality, choice-ax
- used by : trace-correctness

diagonal (diagonalisation argument for m constructed diagrams)

$t_0 = mkt(s_0, 0, 0)$, $t = (ft \uparrow m)[\![t_0]\!]$,
$Trace(s_0)$, $Trace'(s')$, $INV'(s_0\lceil 0\rceil, s'\lceil 0\rceil')$,
$\forall\, k.\ p(s', (ft \uparrow k)[\![t_0]\!], (ft \uparrow k+1)[\![t_0]\!])$
$\vdash$ $INV'(t.s\lceil t.i\rceil, s'\lceil t.j\rceil')$
$\wedge\ m \leq t.i \wedge m \leq t.j \wedge Trace(t.s)$
$\wedge\ (\forall\, i, j.\quad i < j \wedge j \leq m$
$\rightarrow (ft \uparrow i)[\![t_0]\!].i < (ft \uparrow j)[\![t_0]\!].i \wedge (ft \uparrow i)[\![t_0]\!].j < (ft \uparrow j)[\![t_0]\!].j)$
$\wedge\ (\forall\, j, k.\ j \leq m \wedge k \leq (ft \uparrow j)[\![t_0]\!].i \rightarrow (ft \uparrow j)[\![t_0]\!].s\lceil k\rceil = t.s\lceil k\rceil)$

- used lemmas : p-def, inv'-def
- used by : trace-correctness

trace-correctness (trace correctness of the refinement)

$Trace'(s')$, $INV'(st, s'\lceil 0\rceil')$
$\vdash\ \exists\, s.\ Trace(s) \wedge s\lceil 0\rceil = st \wedge (\forall\, m, k.\ \exists\, i, j.\ m \leq i \wedge k \leq j \wedge INV'(s\lceil i\rceil, s'\lceil j\rceil'))$

- used lemmas : diagonal, diagonal-ax, ind-choice-concl, inv'-def

C.4 Iterative Refinement for Indeterministic ASMs

C.4.1 Specification

it-indetcorr =
enrich rule, rule', diagtype **with**
functions

ndt	:	state × state'	→	diagtype	;
exec0n	:	state × state'	→	nat	;
execm0	:	state × state'	→	nat	;

predicates

INV	:	state × state';
IN	:	state × state';
OUT	:	state × state';
KPROP	:	state × state';
MINV	:	state; (: existing invariant for ASM :)
MINVNOW	:	state';
MINV'	:	state'; (: constructed invariant for ASM′ :)

variables i, j, j_0, k: nat;

axioms

minv-ax :
IN(st, st')

$\rightarrow \forall$ i. [**loop if** $\neg$ final(st) **then** RULE(; st) **times** i] MINV(st),

init-ax : IN(st, st') $\rightarrow$ INV(st, st') $\wedge$ MINVNOW(st'),
finboth-ax : final(st) $\wedge$ final'(st') $\wedge$ INV(st, st') $\wedge$ MINV(st) $\rightarrow$ OUT(st, st'),
fin1-ax : final(st) $\wedge$ INV(st, st') $\wedge \neg$ final'(st') $\wedge$ MINV(st) $\rightarrow$ ndt(st, st') = 0n,
fin2-ax : final'(st') $\wedge$ INV(st, st') $\wedge \neg$ final(st) $\wedge$ MINV(st) $\rightarrow$ ndt(st, st') = m0,

mton-corr-ax :
INV(st, st') $\wedge$ MINV(st) $\wedge \neg$ final(st) $\wedge \neg$ final'(st') $\wedge$ ndt(st, st') = m0
$\rightarrow$ [**if** $\neg$ final'(st') **then** RULE'(; st')]
$\exists$ j. [**loop if** $\neg$ final'(st') $\wedge \neg$ MINVNOW(st') **then** RULE'(; st') **times** j]
(MINVNOW(st')
$\wedge$ $\langle$**if** $\neg$ final(st) **then** RULE(; st)$\rangle$
$\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$ INV(st, st')),

0ton-corr-ax :
INV(st, st') $\wedge$ MINV(st) $\wedge$ MINVNOW(st') $\wedge \neg$ final'(st')
$\wedge$ ndt(st, st') = 0n $\wedge$ exec0n(st, st') = k
$\rightarrow$ [**if** $\neg$ final'(st') **then** RULE'(; st')]
$\exists$ j. [**loop if** $\neg$ final'(st') $\wedge \neg$ MINVNOW(st') **then** RULE'(;st') **times** j]
(INV(st, st') $\wedge$ MINVNOW(st')
$\wedge$ ($\neg$ final'(st') $\wedge$ ndt(st, st') = 0n $\rightarrow$ exec0n(st, st') < k)),

mto0-corr-ax :
INV(st, st') $\wedge$ MINV(st) $\wedge$ MINVNOW(st') $\wedge \neg$ final(st)
$\wedge$ ndt(st, st') = m0 $\wedge$ execm0(st, st') = k
$\rightarrow$ $\langle$**if** $\neg$ final(st) **then** RULE(; st)$\rangle$
$\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$
(INV(st, st') $\wedge$ ($\neg$ final(st) $\wedge$ ndt(st, st') = m0 $\rightarrow$ execm0(st, st') < k)),

kprop-def :
KPROP(st, st')
$\leftrightarrow \forall$ i. [**loop if** $\neg$ final(st) **then** RULE(; st) **times** i] MINV(st)
$\wedge$ ($\exists$ j. [**loop if** $\neg$ final'(st') $\wedge \neg$ MINVNOW(st')
then RULE'(; st') **times** j]
(MINVNOW(st')
$\wedge$ ($\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$
INV(st, st')))),

minv'-def : (MINVNOW(st') $\rightarrow$ ($\exists$ st. INV(st, st') $\wedge$ MINV(st))) $\rightarrow$ MINV'(st')

end enrich

C.4.2 Proved Theorems

finite-0ton

INV(st, st'), MINV(st), ndt(st, st') = 0n, $\neg$ final'(st'), MINVNOW(st')
$\vdash$ [**if** $\neg$ final'(st') **then** RULE′(; st')]
$\exists$ j. [**loop if** $\neg$ final'(st') **then** RULE′(; st') **times** j]
(INV(st, st') $\wedge$ MINVNOW(st') $\wedge$ (final'(st') $\vee$ ndt(st, st') $\neq$ 0n))

- used lemmas : 0ton-corr-ax

finite-mto0

$\forall$ i. [**loop if** $\neg$ final(st) **then** RULE(; st) **times** i] MINV(st),
INV(st, st'), ndt(st, st') = m0, $\neg$ final(st), MINVNOW(st')
$\vdash$ $\langle$**if** $\neg$ final(st) **then** RULE(; st)$\rangle$
$\exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$
(INV(st, st')
$\wedge$ (final(st) $\vee$ ndt(st, st') $\neq$ m0)
$\wedge$ ($\forall$ i. [**loop if** $\neg$ final(st) **then** RULE(; st) **times** i] MINV(st)))

- used lemmas : mto0-corr-ax
- used by : corr-step, correctness

corr-step

KPROP(st, st') $\vdash$ [**if** $\neg$ final'(st') **then** RULE$'$(; st')] KPROP(st, st')

- used lemmas : finite-mto0, fin1-ax, 0ton-corr-ax, mton-corr-ax, kprop-def
- used by : corr-j-steps, correctness

kprop-minv'

KPROP(st, st') $\vdash$ MINV'(st')

- used lemmas : minv'-def, kprop-def
- used by : newinvariance

in-kprop

IN(st, st') $\vdash$ KPROP(st, st')

- used lemmas : init-ax, minv-ax, kprop-def
- used by : corr-j-steps, correctness, newinvariance

corr-j-steps

KPROP(st, st')
$\vdash$ [**loop if** $\neg$ final'(st') **then** RULE$'$(; st') **times** j] KPROP(st, st')

- used lemmas : in-kprop, kprop-def, corr-step
- used by : newinvariance

correctness

IN(st, st')
$\vdash$ [**while** $\neg$ final'(st') **do** RULE$'$(; st')]
$\langle$**while** $\neg$ final(st) **do** RULE(; st)$\rangle$ OUT(st, st')

- used lemmas : finboth-ax, fin2-ax, finite-mto0, kprop-def, corr-step, in-kprop

newinvariance (Theorem 9 from Sect. 6.5)

$\exists$ st. IN(st, st')
$\vdash$ $\forall$ j. [**loop if** $\neg$ final'(st') **then** RULE$'$(; st') **times** j] MINV'(st')

- used lemmas : kprop-minv', in-kprop, corr-j-steps

Appendix D

Definition of Admitted Code Sequences (Chains)

D.1 Definition of Linear Chains

```
L-CHAIN#(co, db5; var col)
begin
var instr = code(co, db5)
in if is_try_me(instr)
   then L-CHAIN-TRY-ME#(co, db5; col)
   else if is_clause(instr)
        then col := [co]
        else if instr = nil'
             then col := []
             else abort
end;

L-CHAIN-TRY-ME#(co, db5; var col)
begin
var instr = code(co, db5),
    follow = code(co +1, db5)
in if instr = try_me_else(N)
   then if is_clause(follow)
        then begin
             L-CHAIN-RETRY-ME#(N, db5; col);
             col := [co +1 | col]
             end
        else abort
   else abort
end;

L-CHAIN-RETRY-ME#(co, db5; var col)
begin
var instr = code(co, db5),
    follow = code(co +1, db5)
in if instr = retry_me_else(N)
   then if is_clause(follow)
        then begin
             L-CHAIN-RETRY-ME#(where(instr), db5; col);
             col := [co +1 | col]
```

```
                end
          else abort
   else if is_trust_me(instr)
        then if is_clause(follow)
             then col := [co +1]
             else abort
        else abort
end
```

D.2 Definition of Nested Chains with Switching

```
S-ANY-CHAIN#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if is_retry_me(instr) ∨ is_trust_me(instr)
   then S-CHAIN-RETRY-ME#(trm, co, db7; col)
   else if is_retry(instr) ∨ is_trust(instr)
        then S-CHAIN-RETRY#(trm, co, db7; col)
        else S-CHAIN-REC#(trm, co, db7; col)
end;

S-CHAIN#(trm, co, db7; var col)
begin
if co = failcode then col := []
else S-CHAIN-REC#(trm, co, db7; col)
end;

S-CHAIN-REC#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if is_clause(instr) then col := [co] else
   if instr = try(N) then var col2 = []
                          in begin
                             S-CHAIN-REC#(trm, N, db7; col);
                             S-CHAIN-RETRY#(trm, co +1, db7; col2);
                             col := append(col, col2)
                             end
   else
   if instr = try_me_else(N) then var col2 = []
                                  in begin
                                     S-CHAIN-REC#(trm, co +1, db7; col);
                                     S-CHAIN-RETRY-ME#(trm, N, db7; col2);
                                     col := append(col, col2)
                                     end
   else
   if ¬ is_struct(trm) ∨ arity(trm) < argindex(instr) then abort else
   var xi = arg(trm, argindex(instr))
   in if instr = switch_on_term(argindex, Ns, Nc, Nv, Nl)
      then
      if is_struct(xi) then S-CHAIN#(trm, Ns, db7; col) else
      if is_const(xi) then S-CHAIN#(trm, Nc, db7; col) else
      if is_var(xi) then S-CHAIN#(trm, Nv, db7; col) else
      if is_list(xi) then S-CHAIN#(trm, Nl, db7; col) else abort
      else if instr = switch_on_constant(argindex, tabsize, table)
```

```
            then if is_const(xi)
                 then S-CHAIN#(trm, hashc(table, tabsize, constsym(xi),
                                  db7), db7; col)
                 else abort
       else if instr = switch_on_structure(argindex, tabsize, table)
            then if is_struct(xi)
                 then S-CHAIN#(trm, hashs(table, tabsize, funct(xi),
                                  arity(xi), db7), db7; col)
                 else abort
            else abort
       end;

S-CHAIN-RETRY-ME#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if instr = retry_me_else(N)
   then var col2 = []
        in begin
           S-CHAIN-REC#(trm, co +1, db7; col);
           S-CHAIN-RETRY-ME#(trm, N, db7; col2);
           col := append(col,col2)
           end
   else if is_trust_me(instr)
        then S-CHAIN-REC#(trm, co +1, db7; col)
        else abort
end;

S-CHAIN-RETRY#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if instr = retry(N)
   then var col2 = []
        in begin
           S-CHAIN-REC#(trm, N, db7; col);
           S-CHAIN-RETRY#(trm, co +1, db7; col2);
           col := append(col, col2)
           end
   else if instr = trust(N)
        then S-CHAIN-REC#(trm, N, db7; col)
        else abort
end;

S-CHAIN-RET#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if is_retry_me(instr) ∨ is_trust_me(instr)
   then S-CHAIN-RETRY-ME#(trm, co, db7; col)
   else if is_retry(instr) ∨ is_trust(instr)
        then S-CHAIN-RETRY#(trm, co, db7; col)
        else abort
end;

S-APP-CHAINS-RET#(decglseq', p, stl, db7; var col)
begin
if stl = [] then col := []
```

```
else var col2 = []
     in begin
        S-CHAIN-RET#(acg[car(stl), p[car(stl)], db7; col);
        S-APP-CHAINS-RET#(decglseq', p, cdr(stl), db7; col2);
        col := append(col, col2)
        end
end
```

D.3 Definition of the Length of Nested Chains with Switching

```
C-S-ANY-CHAIN#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if is_retry_me(instr) ∨ is_trust_me(instr) then
      C-S-CHAIN-RETRY-ME#(trm, co, db7; m)
    else if is_retry(instr) ∨ is_trust(instr) then
      C-S-CHAIN-RETRY#(trm, co, db7; m)
    else C-S-CHAIN-REC#(trm, co, db7; m)
end;

C-S-CHAIN#(trm, co, db7; var m)
begin
if co = failcode then m := 0
else C-S-CHAIN-REC#(trm, co, db7; m)
end;

C-S-CHAIN-REC#(trm, co, db7; var m)
begin
var instr = code(co, db7)
in if is_clause(instr) then m := 0 else
   if instr = try(N) then C-S-CHAIN-TRY#(trm, N, db7; m); else
   if instr = try_me(N) then C-S-CHAIN-TRY-ME#(trm, N, db7; m); else
   if ¬ is_struct(trm) ∨ arity(trm) < argindex(instr) then abort
   else var xi = arg(trm, argindex(instr)) in
     if instr = switch_on_term(argindex, Ns, Nc, Nv, Nl) then
       if is_struct(xi) then
         if Ns = failcode then m := 0
         else begin C-S-CHAIN-REC#(trm, Ns, db7; m); m := m +1 end
       else if is_const(xi) then
         if Nc = failcode then m := 0
         else begin C-S-CHAIN-REC#(trm, Nc, db7; m); m := m +1 end
       else if is_var(xi) then
         if Nv = failcode then m := 0
         else begin C-S-CHAIN-REC#(trm, Nv, db7; m); m := m +1 end
       else if is_list(xi) then
         if Nl = failcode then m := 0
         else begin C-S-CHAIN-REC#(trm, Nl, db7; m); m := m +1 end
       else abort
     else if instr = switch_on_constant(argindex, tabsize, table) then
       if is_const(xi) then
         var preg = hashc(table, tabsize, constsym(xi)) in
```

```
                if preg = failcode then m := 0
                else begin
                     C-S-CHAIN-REC#(trm, preg, db7; m);
                     m := m +1
                     end
          else abort
       else if instr = switch_on_structure(argindex, tabsize, table) then
          if is_struct(xi) then
             var preg = hashs(table, tabsize, funct(xi)) in
                if preg = failcode then m := 0
                else begin
                     C-S-CHAIN-REC#(trm, preg, arity(xi), db7; m);
                     m := m +1
                     end
          else abort
       else abort
end;

C-S-CHAIN-TRY-ME#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
     if instr = try_me(N) then
        var m0 = 0 in begin
             C-S-CHAIN-REC#(trm, co +1, db7; m);
             C-S-CHAIN-RETRY-ME#(trm, N, db7; m0);
             m := (m + m0) +1
             end
     else abort
end;

C-S-CHAIN-TRY#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
     if instr = try(N) then
        var m0 = 0 in begin
             C-S-CHAIN-REC#(trm, N, db7; m);
             C-S-CHAIN-RETRY#(trm, co +1, db7; m0);
             m := (m + m0) +1
             end
     else abort
end;

C-S-CHAIN-RETRY-ME#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
     if instr = retry_me(N) then
        var m0 = 0 in begin
             C-S-CHAIN-REC#(trm, co +1, db7; m);
             C-S-CHAIN-RETRY-ME#(trm, N, db7; m0);
             m := (m + m0) +1
             end
     else if trust_me(instr) then
        begin C-S-CHAIN-REC#(trm, co +1, db7; m); m := m +1 end
     else abort
end;
```

```
C-S-CHAIN-RETRY#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if instr = retry(N) then
      var m0 = 0 in begin
          C-S-CHAIN-REC#(trm, N, db7; m);
          C-S-CHAIN-RETRY#(trm, co +1, db7; m0);
          m := (m + m0) +1
          end
    else if instr = trust(N) then
      begin C-S-CHAIN-REC#(trm, N, db7; m); m := m +1 end
    else abort
end;

C-S-CHAIN-RET#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if is_retry_me(instr) ∨ is_trust_me(instr) then
      C-S-CHAIN-RETRY-ME#(trm, co, db7; m)
    else if is_retry(instr) ∨ is_trust(instr) then
      C-S-CHAIN-RETRY#(trm, co, db7; m)
    else abort
end;

C-S-APP-CHAINS-RET#(decglseq', p, stl, db7; var m)
begin
if stl = [] then m := 0
else var m0 = 0 in begin
          C-S-CHAIN-RET#(acg[car(stl), p[car(stl)], db7; m);
          C-S-APP-CHAINS-RET#(decglseq', p, cdr(stl), db7; m0);
          m := (m + m0) +1
          end
end
```

Appendix E

Specifications of the Prolog-WAM Case Study

E.1 Specifications from the Library

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

---

elemI =
rename elem **by morphism**
  elem → elem', a → a', b → b', c → c'
end rename

---

elemII =
rename elem **by morphism**
  elem → elem", a → a", b → b", c → c"
end rename

---

pair =
generic data specification
  parameter elemI + elemII
  pair = ⟨ . , . ⟩ (fst : elem', snd : elem");
  variables p, p_0, p_1 : pair;
end generic data specification

Generated axioms:

pair **freely generated by** ⟨ . , . ⟩;
fst(⟨a', a"⟩) = a',
snd(⟨a', a"⟩) = a",
⟨a', a"⟩ = ⟨a'_0, $a"_0$⟩ ↔ a' = a'_0 ∧ a" = $a"_0$,
⟨fst(p), snd(p)⟩ = p

vartermpair =
actualize pair **with parameter** node, term **by morphism**
elem' → nodesort, elem" → term, pair → pairvarterm
end actualize

varvarpair =
actualize pair **with parameter** node **by morphism**
elem' → nodesort, elem" → nodesort, pair → varvarpair
end actualize

termtermpair =
actualize pair **with** term **by morphism**
elem' → term, elem" → term, pair → termtermpair
end actualize

decgoal =
actualize pair **with** goalsort, **parameter** node **by morphism**
elem' → goalsort, elem" → nodesort, pair → decgoal
end actualize

clause =
actualize pair **with** term, goal **by morphism**
elem' → term, elem" → goalsort,
pair → clausesort, p → cl
end actualize

ident =
actualize pair **with parameter** atom, nat **by morphism**
elem' → atomsort, elem" → nat, pair → ident
end actualize

procdeftable =
actualize pair **with** ident, **parameter** code **by morphism**
elem' → ident, elem" → codesort,
pair → procdeftable, p → pdt
end actualize

comp3result =
actualize pair
with parameter program2, procdeftable
by morphism
elem' → program", elem" → procdeftable, pair → comp3result
.1 →.db, .2 →.pdtab, p → co3res
end actualize

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 cf(z) [x] = z,
f [x ← z] [x] = z,
x ≠ y → f [x ← z] [y] = f[y]
end generic specification

F-no-no =
actualize Dynfun **with parameter** node **by morphism**
dom → nodesort, codom → nodesort,
Dynfun → funnodenode, f → F
end actualize

vi =
actualize Dynfun **with** nat **by morphism**
dom → nat, codom → nodesort, Dynfun → vifun, f → vi
end actualize

F-co-co =
actualize Dynfun **with parameter** code **by morphism**
dom → codesort, codom → codesort,
Dynfun → funcodecode, f → C
end actualize

cll =
actualize Dynfun **with parameter** node, **parameter** code **by morphism**
dom → nodesort, codom → codesort,
Dynfun → cllfun, f → cll
end actualize

decglseq =
actualize Dynfun **with** decgoallist **by morphism**
dom → nodesort, codom → decgoallist,
Dynfun → decgoalseqfun, f → decglseq
end actualize

cands =
actualize Dynfun **with** nodelist **by morphism**
dom → nodesort, codom → nodelist,
Dynfun → candsfun, f → cands
end actualize

p =
actualize Dynfun **with parameter** code **by morphism**
dom → nodesort, codom → codesort,
Dynfun → pfun, f → p
end actualize

cg =
actualize Dynfun **with** goal **by morphism**
dom → nodesort, codom → goal, Dynfun → cgfun, f → cg
end actualize

cp =
actualize Dynfun **with parameter** node, **parameter** code **by morphism**

dom → nodesort, codom → codesort,
Dynfun → cpfun, f → cp
end actualize

sub =
actualize Dynfun **with** substitution **by morphism**
dom → nodesort, codom → substitution,
Dynfun → subfun, f → sub
end actualize

goalfun =
actualize Dynfun **with parameter** node, goal **by morphism**
dom → nodesort, codom → goalsort,
Dynfun → goalfun, f → goal
end actualize

H-no-nol =
actualize Dynfun **with** nodelist **by morphism**
dom → nodesort, codom → nodelist,
Dynfun → funnodenodelist, f → H
end actualize

---

nat-basic1 =
data specification
nat = 0 | . +1 (. −1 : nat);
variables i, j, k, m : nat;
order predicates . < . : nat × nat;
end data specification

Generated axioms:

nat **freely generated by** 0, +1;
i +1 −1 = n,
i +1 = j +1 ↔ i = j,
0 ≠ i +1,
i = 0 ∨ i = i −1 +1,
¬ i < i,
i < j ∧ j < k → i < k,
¬ i < 0,
i < j +1 ↔ i = j ∨ i < j

---

nat =
enrich nat-basic1 **with**
functions . + . : nat × nat → nat;
. − . : nat × nat → nat **prio** 8 **left**;
predicates
. ≤ . : nat × nat;
. > . : nat × nat;
. ≥ . : nat × nat;

axioms

$i + 0 = i$,
$i + j + 1 = (i + j) + 1$,
$i - 0 = i$,
$i - j + 1 = (i - j) - 1$,
$i \leq j \leftrightarrow \neg\, j < i$,
$i > j \leftrightarrow j < i$,
$i \geq j \leftrightarrow \neg\, i < j$

end enrich

---

set =
generic specification
parameter elem **using** nat **target**
sorts set;
constants $\emptyset$: set;
functions
$\{\ .\ \}$: elem $\rightarrow$ set;
$.\ \cup\ .$: set $\times$ set $\rightarrow$ set **prio** 9 **left**;
predicates
$.\ \in\ .$: elem $\times$ set;
$.\ \subseteq\ .$: set $\times$ set;
variables s, s' : set;

axioms

set **generated by** $\emptyset$, $\{\ .\ \}$, $.\ \cup\ .$
$\neg\, a \in \emptyset$,
$a \in \{b\} \leftrightarrow a = b$,
$a \in s \cup s' \leftrightarrow a \in s \vee a \in s'$,
$s = s' \leftrightarrow (\forall\, a.\ a \in s \leftrightarrow a \in s')$,
$s \subseteq s' \leftrightarrow (\forall\, a.\ a \in s \rightarrow a \in s')$

end generic specification

nodeset =
actualize set **with parameter** node **by morphism**
elem $\rightarrow$ nodesort, set $\rightarrow$ nodeset
end actualize

---

list-data =
generic data specification
parameter elem **using** nat
list = []
| [. | .] (car : elem, cdr : list)
;
variables x, y, z : list;
size functions # : list $\rightarrow$ nat ;
order predicates $.\ \ll\ .$: list $\times$ list;
end generic data specification

Generated axioms:

list **freely generated by** [], [. | .]
car([a | x]) = a,
cdr([a | x]) = x,
[a | x] = [b | y] ↔ a = b ∧ x = y,
[] ≠ [a | x],
x = [] ∨ x = [car(x) | cdr(x)],
#([]) = 0,
#([a | x]) = #(x)+1,
¬ x ≪ x,
x ≪ y ∧ y ≪ z → x ≪ z,
¬ x ≪ [],
y ≪ [a | x] ↔ y = x ∨ y ≪ x

---

list =
enrich list-data **with**
functions

append	:	list × list	→	list;
rmdup	:	list	→	list;
pos	:	list × elem	→	nat;
rev	:	list	→	list;

predicates

. ∈ .	:	elem × list;
. subli .	:	list × list;
. subse .	:	list × list;
. ⊂ .	:	list × list;
dups	:	list;
nodups	:	list;

axioms

append([], x) = x,
append([a | x], y) = [a | append(x, y)],
a ∈ x ↔ (∃ y, z. x = append(y, [a | z]),
[] subli x,
¬ [a | x] subli [],
[a | x] subli [b | y] ↔ a = b ∧ x subli y ∨ a ≠ b ∧ [a | x] subli y,
[] subse x,
[a | x] subse y ↔ a ∈ y ∧ x subse y,
nodups([]),
nodups([a | x]) ↔ ¬ a ∈ x ∧ nodups(x),
dups(x) ↔ ¬ nodups(x),
rmdup([]) = [],
a ∈ x → rmdup([a | x]) = rmdup(x),
¬ a ∈ x → rmdup([a | x]) = [a | rmdup(x)],
x ⊂ y ↔ #(rmdup(x)) < #(rmdup(y)) ∧ x subse y
pos([a | x], a) = 0,
a ≠ b → pos([a | x], b) = pos(x, b) +1,
rev([]) = [],
rev([a | x]) = append(rev(x), [a])

end enrich

substitution =

actualize list **with** pairvarterm **by morphism**
elem → pairvarterm, list → substitution, x → su
end actualize

goal =
actualize list **with** term **by morphism**
elem → term, list → goal, x → go
end actualize

natlist =
actualize list **with** nat **by morphism**
elem → nat, list → natlist, x → nl
end actualize

varlist =
actualize list **with parameter** node **by morphism**
elem → nodesort, list → varlist, x → vl
end actualize

nodelist =
actualize list **with parameter** node **by morphism**
elem → nodesort, list → nodelist, x → stack
end actualize

codelist =
actualize list **with parameter** code **by morphism**
elem → codelist, list → codesort, x → col
end actualize

decgoallist =
actualize list **with** decgoal **by morphism**
elem → decgoal, list → decgoallist, x → dgl
end actualize

clauselist =
actualize list **with** clause **by morphism**
elem → clause, list → clauselist, x → cli

renaming =
actualize list **with** varvarpair **by morphism**
elem → varvarpair, list → renaming
end actualize

E.2 Specifications for ASM1 (PrologTree)

enrnodeset =
enrich nodeset **with**
functions new : nodeset → elem;

axioms

¬ new(s) ∈ s, new([]) = ⊥

end enrich

---

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

Generated axioms:

modesort **freely generated by** select, call;
select ≠ call,
mode = select ∨ mode = call

---

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

Generated axioms:

stopmodesort **freely generated by** success, failure, run;
failure ≠ run, success ≠ run, success ≠ failure,
stop = success ∨ stop = failure ∨ stop = run

---

node =
specification
sorts nodesort;
constants ⊥ : nodesort;
variables n : nodesort;

end specification

---

atom =
specification
sorts atomsort;
constants cutsym , failsym, truesym : atomsort;
variables at : atomsort;

axioms

cutsym ≠ failsym,
failsym ≠ truesym,
truesym ≠ cutsym

end specification

---

term =
data specification
using nat, **parameter** atom, **parameter** ordnode
term = struct (funct : atomsort, args : termlist) **with** is_struct
| mkconst (constsym : atomsort) **with** is_const
| mkvar (varsym : nodesort) **with** is_var
| mklist (lcar : term, lcdr : term) **with** is_list
;
termlist = the_one (and_only : term)
| tcons (tcar : term, tcdr : termlist)
;
variables trm, trm_0 : term; trmli, $trmli_0$: termlist;
size functions tlen : termlist → nat ;
order predicates . $<_{tl}$. : termlist × termlist;
end data specification

Generated axioms:

term, termlist **freely generated by** struct, mkconst, mkvar, mklist, the_one, tcons;
⋮

---

subst =
enrich decgoallist, ident, enrterm **with**
functions

. $\hat{}_d$.	:	decgoallist × substitution	→	decgoallist;
. $\hat{}_{sg}$.	:	substitution × goalsort	→	goalsort;
. $\hat{}_t$.	:	substitution × term	→	term;
. $\hat{}_{tl}$.	:	substitution × termlist	→	termlist;

axioms

su $\hat{}_{sg}$ [] = [],
su $\hat{}_{sg}$ [trm | go] = [su $\hat{}_t$ trm | su $\hat{}_{sg}$ go],
su $\hat{}_d$ [] = [],
su $\hat{}_d$ [⟨go, st⟩ | dgl] = [⟨su $\hat{}_{sg}$ go, st⟩ | su $\hat{}_d$ dgl],
su $\hat{}_t$ struct(at, trmli) = struct(at, su $\hat{}_{tl}$ trmli),
su $\hat{}_t$ mklist(trm, trm_0) = mklist(su $\hat{}_t$ trm, su $\hat{}_t$ trm_0),
[] $\hat{}_t$ mkvar(va) = mkvar(va),
[⟨va, trm⟩ | su] $\hat{}_t$ mkvar(va) = trm,
va ≠ va_0 → [⟨va_0, trm⟩ | su] $\hat{}_t$ mkvar(va) = su $\hat{}_t$ mkvar(va),
su $\hat{}_t$ mkconst(at) = mkconst(at),
su $\hat{}_{tl}$ the_one(trm) = the_one(su $\hat{}_t$ trm),
su $\hat{}_{tl}$ tcons(trm, trmli) = tcons(su $\hat{}_t$ trm, su $\hat{}_{tl}$ trmli),
[] o su = su,
[⟨va_0, trm⟩ | su] o su_0 = [⟨va_0, su_0 $\hat{}_t$ trm⟩ | su o su_0]

end enrich

---

substornil =
data specification
using subst

substornil = oksubst(the_subst : substitution) | nil;
variables subst : substornil;
end data specification

Generated axioms:

substornil **freely generated by** nil, oksubst;
the_subst(oksubst(su)) = su,
oksubst(su) = oksubst(su_0) $\leftrightarrow$ su = su_0,
oksubst(su) $\neq$ nil,
subst = oksubst(the_subst(subst)) $\vee$ subst = nil

---

enrterm =
enrich term, substornil **with**
constants ! : term; true : term; fail : term;
functions

. $\circ$.	:	substitution $\times$ substornil	$\rightarrow$	substitution;
. o .	:	substitution $\times$ substitution	$\rightarrow$	substitution;
arity	:	term	$\rightarrow$	nat;
arg	:	term $\times$ nat	$\rightarrow$	term;
. $\odot_{tl}$.	:	termlist $\times$ termlist	$\rightarrow$	termlist;

predicates is_user_defined : term;
variables su, su_1, su_2 : substitution;

axioms

the_one(trm) $\odot_{tl}$ trmli = tcons(trm, trmli),
tcons(trm, trmli) $\odot_{tl}$ $trmli_1$ = tcons(trm, trmli $\odot_{tl}$ $trmli_1$),
! = mkconst(cutsym),
true = mkconst(truesym),
fail = mkconst(failsym),
is_user_defined(trm) $\leftrightarrow$ trm $\neq$ true $\wedge$ trm $\neq$ fail $\wedge$ trm $\neq$!,
arity(trm) = tlen(args(trm))+1,
args(trm) = the_one(trm_1) $\rightarrow$ arg(trm, 0 +1) = trm_1,
args(trm) = tcons(trm_1, trmli)
$\rightarrow$ arg(trm, 0 +1) = trm_1
$\wedge$ (0 < n $\rightarrow$ arg(trm, n +1) = arg(struct(funct(trm), trmli), n)),
su $\circ$ oksubst(su_0) = su o su_0

end enrich

---

unify =
enrich substornil **with**
functions unify : term $\times$ term $\rightarrow$ substornil;

end enrich

---

code =
specification
sorts codesort;
constants failcode : codesort;
functions

. +1	:	codesort	$\rightarrow$	codesort;
. -1	:	codesort	$\rightarrow$	codesort;

variables co : codesort;

axioms

co +1 −1 = co,
co −1 +1 = co

end specification

---

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

---

union0 = mode + stopmode + unify + clauselist + rename + enrnodeset +
sub + cll + subst + F-no-no + decglseq + enrterm

---

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

end enrich

---

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

end enrich

PrologTree =
enrich union0 + cands + procdef + clausefun **with**
functions
mapclause : codelist × program → clauselist;
map : cllfun × nodelist → codelist;
predicates
every : funnodenode × nodelist × nodesort;
disjoint : nodelist × nodelist;
disjointls : nodelist × nodeset;
variables father: funnodenode;

axioms

mapclause([], db) = [],
mapclause([co | col], db) = [clause(co, db) | mapclause(col, db)],
every(father, [], n),
every(father, [n_1 | stack], n)
↔ father[n_1] = n ∧ every(father, stack, n),
map(cll, []) = [],
map(cll, [n | stack]) = [cll[n] | map(cll, stack)],
disjoint(stack, $stack_0$) ↔ (∀ n. n ∈ stack → ¬ n ∈ $stack_0$),
disjointls(stack, s) ↔ (∀ n. n ∈ stack → ¬ n ∈ s)

end enrich

E.3 Specifications for ASM2 (TreetoStack)

procdef2 =
enrich term, **parameter** program, **parameter** code **with**
functions procdef_2 : term × program → codesort;

end enrich

---

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

Generated axioms:

clauseornull **freely generated by** null, mkclau;
the_clau(mkclau(cl)) = cl,
$\text{mkclau(cl)} = \text{mkclau(cl}_0) \leftrightarrow \text{cl} = \text{cl}_0$,
mkclau(cl) ≠ null,
cln = mkclau(the_clau(cln)) ∨ cln = null

---

clauseIfun =
enrich code, clauseornull, program **with**
functions clause' : codesort × program → clauseornull;

axioms

clause'(failcode, db) = null

end enrich

PrologStack =
enrich union0 + procdef2 + codelist + nodelist + clauseIfun **with**
functions

. from .	:	nodelist × nodesort	→	nodelist **prio** 7;
cdr	:	nodelist	→	nodelist;

predicates

. cutptsin .	:	decgoallist × nodelist;
. ctpelem .	:	decgoallist × nodeset;
. ⊆ .	:	nodelist × nodeset;

axioms

mapclause'([], db) = [],
mapclause'([co | col], db) = [the_clau(clause'(co, db)) | mapclause'(col, db)],
[] cutptsin stack,
[⟨go, n⟩ | dgl] cutptsin stack
↔ (n = ⊥ ∨ n ∈ stack) ∧ dgl cutptsin (stack from n),
[] from n = [],
[n | stack] from n = [n | stack],
$\text{n}_1 \neq \text{n} \rightarrow [\text{n}_1 \mid \text{stack}]$ from n = stack from n,

[] ctpelem s,
[⟨go, n⟩ | dgl] ctpelem s ↔ n ∈ s ∧ dgl ctpelem s,
stack ⊆ s ↔ (∀ n. n ∈ stack → n ∈ s),
cdr([]) = [],
cdr([n | stack]) = stack

end enrich

CompAssum1 =
enrich PrologTree, PrologStack **with**
functions compile_{12} : program → program;
variables lit : term; db : program;

axioms

⟨CLLS#(procdef_2(lit, compile_{12}(db)), compile_{12}(db); col)⟩
mapclause(procdef(lit, db), db) = mapclause'(col, compile_{12}(db))

end enrich

Tree+Stack+F =
enrich F-no-no, PrologTree, PrologStack **with**
functions
F_d : funnodenode × decgoallist → decgoallist;
F_s : funnodenode × nodeset → nodeset;
predicates
candsdisjoint : funnodenode × candsfun × nodelist;
. injon . : funnodenode × nodelist;
nocands : funnodenode × candsfun × nodelist;

axioms

F_d(F, []) = [],
F_d(F, [⟨go, n⟩ | dgl]) = [⟨go, F[n]⟩ | F_d(F, dgl)],
F_s(F, ∅) = ∅,
F_s(F, s ∪ {n}) = F_s(F, s) ∪ {F[n]},
candsdisjoint(F, cands, stack)
↔ ∀ n, n_1. n ∈ stack ∧ n_1 ∈ stack ∧ n ≠ n_1
→ disjoint(cands[F[n_1]], cands[F[n]]),
F injon stack
↔ (∀ n, n_1. n ∈ stack ∧ n_1 ∈ [⊥ | stack] ∧ n ≠ n_1 → F[n] ≠ F[n_1]),
nocands(F, cands, stack)
↔ ∀ n, n_1. n ∈ stack ∧ n_1 ∈ [⊥ | stack] → ¬ F[n_1] ∈ cands[F[n]]

end enrich

TreetoStack = CompAssum1 + Tree+Stack+F

E.4 Specifications for ASM3 (ReuseChoicep)

rmode =
data specification
rmodesort = try | retry | enter | call;
variables rmode : rmodesort;
end data specification

Generated axioms:

rmodesort **freely generated by** try, retry, enter, call;
enter ≠ call, retry ≠ call, retry ≠ enter,
try ≠ call, try ≠ enter, try ≠ retry,
rmode = try ∨ rmode = retry ∨ rmode = enter ∨ rmode = call

PrologStack+F =
enrich F-no-no, Tree+Stack+F **with**
functions F_l : funnodenode × nodelist → nodelist;

axioms

F_l(F, []) = [],
F_l(F, [n | stack]) = [F[n] | F_l(F, stack)]

end enrich

ReuseChoicep = PrologStack+F + rmode

E.5 Specifications for ASM4 (DetermDetect)

DetermDetect = PrologStack+F + rmode

E.6 Specifications for ASM5 (CompPredStruct)

instr+clau =
data specification
using nat, clause, varlist, **parameter** code
instr-or-cl = try_me_else (where : codesort) **with** is_try_me
| retry_me_else (where : codesort) **with** is_retry_me
| trust_me **with** is_trust_me
| try (what : codesort) **with** is_try
| retry (what : codesort) **with** is_retry
| trust (what : codesort) **with** is_trust
| switch_on_term (argindex : nat,
vlabel : codesort, clabel : codesort,
llabel : codesort, slabel : codesort)
with is_sw_term
| switch_on_constant (argindex : nat,

```
                                    tabsize : nat, table : codesort)
        with is_sw_const
      | switch_on_structure (argindex : nat,
                                     tabsize : nat, table : codesort)
        with is_sw_struct
      | mkcl (the_cl : clausesort) with is_clause
      | mkcall (calllit : term) with is_call
      | mkunify (unifylit : term) with is_unify
      | allocate
      | deallocate
      | proceed
      | null
      | code_of_start
      ;
  variables ioc : instr-or-cl;
end data specification
```

Generated axioms:

instr-or-cl **freely generated by** trust_me, allocate, deallocate, proceed, null',
code_of_start, try_me_else, retry_me_else, try', retry', trust, switch_on_term,
switch_on_constant, switch_on_structure, mkcl, mkcall, mkunify;
⋮

---

procdef3 =
enrich term, **parameter** program2, **parameter** code **with**
functions procdef$_3$: term × program" → codesort;

end enrich

---

codefun =
enrich parameter code, **parameter** program2, instr+clau **with**
constants start : codesort;
functions code : codesort × program" → instr-or-cl;

axioms

co = start ↔ code(co, db$_2$) = code_of_start,
code(failcode, db$_2$) = nil'

end enrich

CompAssum2 =
enrich CompAssum1, instr+clau, codefun, procdef3 **with**
functions
compile$_{45}$: program → program";
mapcode : codelist × program" → clauselist;
variables lit : term; db$_2$: program; db$_5$: program";

axioms

mapcode([], db_5) = [],
mapcode([co | col], db_5) = [the_cl(code(co, db_5)) | mapcode(col, db_5)],
[CLLS#(procdef$_2$(lit, db_2), db_2; col_1)]
⟨CHAIN-FL#(procdef$_3$(lit, compile$_{45}$(db_2)), compile$_{45}$(db_2); col_2)⟩
mapcode(col_2, compile$_{45}$(db_2)) = mapclause'(col_1, db_2)

end enrich

CompPredStruct = CompAssum2 + PrologStack+F + rmode + p

E.7 Specifications for ASM6 (CompPredStruct2)

hash =
enrich nat, **parameter** atom,
parameter code, **parameter** program2 **with**
functions
hashc : codesort × nat × atomsort × program" → codesort;
hashs : codesort × nat × atomsort × nat × program" → codesort;

end enrich

CompAssum3a =
enrich CompAssum2, p, hash **with**
functions compile$_{56}$: program" → program";

axioms

[CHAIN-FL#(procdef$_2$(lit, db_5), db_5; col_1)]
⟨CHAIN#(procdef$_3$(lit, compile$_{56}$(db_5)), compile$_{56}$(db_5); col_2)⟩
mapcode(col_1, db_5) = mapcode(col_2, compile$_{56}$(db_5))

end enrich

CompPredStruct2 = CompAssum3a + PrologStack+H + p

E.8 Specifications for ASM7 (Switching)

idfun =
enrich enrterm, ident **with**
functions id : term → ident;

axioms

is_struct(trm) → id(trm) = mkident(funct(trm), arity(trm)),
is_const(trm) → id(trm) = mkident(constsym(trm), 0)

end enrich

CompAssum3 =
enrich comp3result, CompAssum2, p, hash, idfun **with**
functions compile_{57} : program" $\to$ comp3result;

axioms

[CHAIN-FL#(procdef$_2$(lit, db_5), db_5; col_1)]
⟨S-CHAIN#(lit, $\text{compile}_{57}(\text{db}_5)$.pdt[id(lit)], $\text{compile}_{57}(\text{db}_5)$.db; col_2)⟩
mapcode(col_1, db_5) = mapcode(col_2, $\text{compile}_{57}(\text{db}_5)$.db)

end enrich

PrologStack+H =
enrich PrologStack, H-no-nol **with**
functions

H_d	:	funnodenodelist × decgoallist	→	decgoallist;
H_l	:	funnodenodelist × nodelist	→	nodelist;
car	:	nodelist	→	nodesort;

axioms

H_d(h, []) = [],
H_d(h, [⟨go, n⟩ | dgl] = [⟨go, car(h[n])⟩ | H_d(h, dgl)],
H_l(h, []) = [],
H_l(h, [n | stack]) = append(h[n], H_l(h, stack)),
car([]) = ⊥,
car([n | stack]) = n

end enrich

Switching =
enrich CompAssum3, PrologStack+H, p **with**
functions . $-_{sl}$. : nodelist × nodelist → nodelist;
predicates
eqh : funnodenodelist × funnodenodelist × decgoallist × decgoallist;
. $<=_s$. : nodelist × nodelist;

axioms

eqh(h, h_0, [], []),
¬ eqh(h, h_0, [⟨go, n⟩ | dgl], []),
¬ eqh(h, h_0, [], [⟨go_0, n_0⟩ | dgl_0]),
eqh(h, h_0, [⟨go, n⟩ | dgl], [⟨go_0, n_0⟩ | dgl_0])
↔ go = go_0
∧ (n = ⊥ ⊃ n_0 ∈ h_0[⊥] ∨ n_0 = ⊥
; n_0 ∈ h_0[n] ∧ ¬ n_0 ∈ cdr(h[n]))
∧ eqh(h, h_0, dgl, dgl_0),
stack $<=_s$ stack_0 ↔ stack $\ll_s$ stack_0 ∨ stack = stack_0,
stack $<=_s$ stack_0 → (stack_0 $-_{sl}$ stack) $\odot_{sl}$ stack = stack_0

end enrich

E.9 Specifications for ASM8 (ShareCont)

ordnode =
enrich parameter node **with**
functions

. +1	:	nodesort	→	nodesort;
. −1	:	nodesort	→	nodesort;
max	:	nodesort × nodesort	→	nodesort;

predicates . ≪ . : nodesort × nodesort;

axioms

$n +1 -1 = n$,
$n -1 +1 = n$,
$n \ll n +1$,
$\neg\ n \ll n$,
$n_1 \ll n_2 \vee n_1 = n_2 \vee n_2 \ll n_1$,
$n \ll n_0 \wedge n_0 \ll n_1 \rightarrow n \ll n_1$,
$n_1 \ll n_2 \rightarrow \max(n_1, n_2) = n_2$,
$\neg\ n_1 \ll n_2 \rightarrow \max(n_1, n_2) = n_1$

end enrich

---

rensubst =
enrich substitution, renaming **with**
functions

. $\hat{}_r$.	:	renaming × term	→	term;
. $\hat{}_{rl}$.	:	renaming × termlist	→	termlist;

axioms

rn $\hat{}_r$ struct(at, trmli) = struct(at, rn $\hat{}_{rl}$ trmli),
rn $\hat{}_r$ mklist(trm, trm_0) = mklist(rn $\hat{}_r$ trm, rn $\hat{}_r$ trm_0),
[] $\hat{}_r$ mkvar(va) = mkvar(va),
[⟨va_1,va_2⟩ | rn] $\hat{}_r$ mkvar(va_1) = mkvar(va_2),
va ≠ va_1 → [⟨va_1,va_2⟩ | rn] $\hat{}_r$ mkvar(va) = rn $\hat{}_r$ mkvar(va),
rn $\hat{}_r$ mkconst(at) = mkconst(at),
rn $\hat{}_{rl}$ the_one(trm) = the_one(rn $\hat{}_r$ trm),
rn $\hat{}_{rl}$ tcons(trm, trmli) = tcons(rn $\hat{}_r$ trm, rn $\hat{}_{rl}$ trmli)

end enrich

---

less-vi =
enrich subst, vi, varlist, actrenterm, unify, rename **with**
functions

rentl	:	termlist × nat	→	termlist;
rentl'	:	termlist × nodesort × vifun	→	termlist;
rent'	:	term × nodesort × vifun	→	term;
reng'	:	goalsort × nodesort × vifun	→	goalsort;
renv	:	nodesort × nat	→	nodesort;

predicates

. $<_{svi}$. : substitution × nat;
. $<_{tvi}$. : term × nat;
. $<_{tlvi}$. : termlist × nat;
. $<_{gvi}$. : goalsort × nat;
. $<_{dvi}$. : decgoallist × nat;
. $<_{cvi}$. : clausesort × nat;
. $<_{vvi}$. : nodesort × nat;
. $<_{vlvi}$. : varlist × nat;

variables lit : term;

axioms

su $<_{svi}$ i ∧ su_0 $<_{svi}$ i → su o su_0 $<_{svi}$ i,
su $<_{svi}$ i → su $\hat{}_t$ rent(trm, i) = rent(trm, i),
trm $<_{tvi}$ i ∧ trm_1 $<_{tvi}$ i ∧ unify(trm, trm_1) ≠ nil
→ the_subst(unify(trm, trm_1)) $<_{svi}$ i,
trm $<_{tvi}$ i ∧ i < j → trm $<_{tvi}$ j,
trm $<_{tvi}$ 0 → rent(trm, i) $<_{tvi}$ i +1,
[] $<_{gvi}$ i,
[trm | go] $<_{gvi}$ i ↔ trm $<_{tvi}$ i ∧ go $<_{gvi}$ i,
[] $<_{dvi}$ i,
[⟨go, ctpt⟩ | dgl] $<_{dvi}$ i ↔ go $<_{gvi}$ i ∧ dgl $<_{dvi}$ i,
⟨lit, go⟩ $<_{cvi}$ i ↔ lit $<_{tvi}$ i ∧ go $<_{gvi}$ i,
[] $<_{svi}$ i,
[⟨va_0, trm⟩ | su] $<_{svi}$ i
↔ mkvar(va_0) $<_{tvi}$ i ∧ trm $<_{tvi}$ i ∧ su $<_{svi}$ i,
rent(mkvar(va), i) = mkvar(renv(va, i)),
va $<_{vvi}$ 0 → ¬ renv(va, i) $<_{vvi}$ i,
rent(mkconst(at), i) = mkconst(at),
rent(struct(at, trmli), i) = struct(at, rentl(trmli, i)),
rent(mklist(trm, trm_0), i) = mklist(rent(trm, i), rent(trm_0, i)),
rentl(the_one(trm), i) = the_one(rent(trm, i)),
rentl(tcons(trm, trmli), i) = tcons(rent(trm, i), rentl(trmli, i)),
the_one(trm) $<_{tlvi}$ i ↔ trm $<_{tvi}$ i,
tcons(trm, trmli) $<_{tlvi}$ i ↔ trm $<_{tvi}$ i ∧ trmli $<_{tlvi}$ i,
struct(at, trmli) $<_{tvi}$ i ↔ trmli $<_{tlvi}$ i,
mkconst(at) $<_{tvi}$ i,
mklist(trm, trm_0) $<_{tvi}$ i ↔ trm $<_{tvi}$ i ∧ trm_0 $<_{tvi}$ i,
mkvar(va) $<_{tvi}$ i ↔ va $<_{vvi}$ i,
[] $<_{vlvi}$ i,
[va | vl] $<_{vlvi}$ i ↔ va $<_{vvi}$ i ∧ vl $<_{vlvi}$ i,
va $<_{vvi}$ 0 ∧ va_0 $<_{vvi}$ 0
→ (renv(va, i) = renv(va_0, j) ↔ va = va_0 ∧ i = j),
rentl'(trmli, ctpt, vi)
= (ctpt ≠ ⊥ ⊃ rentl(trmli, vi[ctpt]) ; trmli),
rent'(trm, ctpt, vi)
= (ctpt ≠ ⊥ ⊃ rent(trm, vi[ctpt]) ; trm),
reng'(go, ctpt, vi)
= (ctpt ≠ ⊥ ⊃ reng(go, vi[ctpt]) ; go)

end enrich

RenAssum =
enrich CompAssum3, less-vi **with**

predicates

. $<_{clvi}$. : clauselist, nat;
nonvargoal : goalsort;

axioms

mapclause(procdef(lit,db),db) $<_{clvi}$ 0,
[] $<_{clvi}$ i,
[cl | cli] $<_{clvi}$ i $\leftrightarrow$ cl $<_{cvi}$ i $\wedge$ nonvargoal(bdy(cl)) $\wedge$ cli $<_{clvi}$ i,
nonvargoal([]),
nonvargoal([trm | go])
$\leftrightarrow \neg$ is_var(trm) $\wedge \neg$ is_list(trm) $\wedge$ trm $<_{tvi}$ 0 $\wedge$ nonvargoal(go)

end enrich

---

rename =
enrich nat, clause **with**

functions

ren : clausesort $\times$ nat $\rightarrow$ clausesort;
rent : term $\times$ nat $\rightarrow$ term;
reng : goalsort $\times$ nat $\rightarrow$ goalsort;

axioms

ren(mkclause(trm, go), i) = mkclause(rent(trm, i), reng(go, i)),
reng([], i) = [],
reng([trm | go], i) = [rent(trm, i) | reng(go, i)]

end enrich

ShareCont =
enrich parameter ordnode, cg, PrologStack+F,
goalfun, RenAssum **with**

functions

decglseqof : funnodenode $\times$ cgfun $\times$ funnodenode $\times$ nodelist
$\rightarrow$ decgoallist;

predicates

ordered : nodelist;

axioms

decglseqof(cutpt, cg, ce, []) = [],
decglseqof(cutpt, cg, ce, [n | stack])
= [$\langle$cg[n], cutpt[ce[n]]$\rangle$ | decglseqof(cutpt, cg, ce, stack)],
ordered([]),
ordered([n]) $\leftrightarrow \bot \ll$ n,
ordered([n | n_0 | stack]) $\leftrightarrow n_0 \ll$ n $\wedge$ ordered([n_0 | stack])

end enrich

E.10 Specifications for ASM9 (CompClause)

comp4result =
data specification
using procdeftable, **parameter** program2
comp4result = mkco4res (. .pc : codesort, . .pdtab : procdeftable,
. .dbc : program");
variables co4res : comp4result;
end data specification

Generated axioms:

comp4result **freely generated by** mkco4res;
mkco4res(co, procdeftab, db_7).pc = co,
mkco4res(co, procdeftab, db_7).pdtab = procdeftab,
mkco4res(co, procdeftab, db_7).dbc = db_7,
mkco4res(co, procdeftab, db_7) = mkco4res(co_0, $procdeftab_0$, db'_7)
$\leftrightarrow$ co = co_0 $\wedge$ procdeftab = $procdeftab_0$ $\wedge$ $db_7 = db'_7$,
mkco4res(co4res.pc, co4res.pdtab, co4res.dbc) = co4res

CompAssum4 =
enrich CompAssum3, clauselist, comp4result, F-co-co, RenAssum **with**
functions $compile_{89}$: comp3result $\times$ goalsort $\rightarrow$ comp4result ;
predicates
eqpdt : procdeftable $\times$ procdeftable $\times$ funcodecode;
eqcode : program" $\times$ program" $\times$ funcodecode;
variables pdtab : procdeftable; query, goalreg : goalsort;

axioms

$\langle db_7, procdef_7\rangle = compile_{57}(compile_{45}(compile_{12}(db)))$
$\rightarrow \exists$ C. eqpdt(procdef$_7$, $compile_{89}(\langle db_7, procdef_7\rangle$, goalreg).pdtab, C)
$\wedge$ eqcode(db_7, $compile_{89}(\langle db_7, procdef_7\rangle$, goalreg).dbc, C),

eqpdt($pdtab_0$, pdtab, C) $\leftrightarrow \forall$ lit. pdtab[id(lit)] = C[$pdtab_0$[id(lit)]],

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = mkcl(cl_0)
$\rightarrow \langle$UNLOAD#(C[co], db_9; cl)$\rangle$ cl = cl_0

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = try_me_else(N)
$\rightarrow$ code(C[co], db_9) = try_me_else(C[N])

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = retry_me_else(N)
$\rightarrow$ code(C[co], db_9) = retry_me_else(C[N])

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = trust_me
$\rightarrow$ code(C[co], db_9) = trust_me

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = try(N)
$\rightarrow$ code(C[co], db_9) = try(C[N])

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = retry(N)
$\rightarrow$ code(C[co], db_9) = retry(C[N])

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = trust(N)
$\rightarrow$ code(C[co], db_9) = trust(C[N])

eqcode(db_7, db_9, C) $\wedge$ code(co,db_7) = failcode

$\rightarrow$ code(C[co], db_9) = failcode

eqcode(db_7, db_9, C)
$\wedge$ code(co,db_7) = switch_on_term(argindex, N_s, N_c, N_v, N_l))
$\rightarrow$ code(C[co], db_9) = switch_on_term(argindex, C[N_s], C[N_c], C[N_v], C[N_l])

eqcode(db_7, db_9, C)
$\wedge$ code(co,db_7) = switch_on_constant(argindex, tabsize, co)
$\rightarrow \exists$ co_0. code(C[co], db_9) = switch_on_constant(argindex, tabsize, co_0)
$\wedge \forall$ at. C[hashc(co, tabsize, at, db_7)]
= hashc(co_0, tabsize, at, db_9))

$compile_{57}(compile_{45}(compile_{12}(db))) = \langle db_7, procdef_7 \rangle$
$\wedge$ nonvargoal(goalreg)
$\rightarrow \langle$QUERY#($compile_4$(db_9, goalreg).pc, $compile_4$(db_9, goalreg).dbc; go)$\rangle$
go = goalreg

end enrich

CompClause = CompAssum4 + ShareCont + cp

E.11 Specifications for ASM9a (Renaming)

termvarli =
enrich varlist, enrterm **with**
functions

tvarli	:	term	$\rightarrow$	varlist;
tlvarli	:	termlist	$\rightarrow$	varlist;

axioms

tvarli(mkconst(at)) = [],
tvarli(mkvar(va)) = [va | []],
tvarli(mklist(trm, trm_1)) = rmdup(append(tvarli(trm), tvarli(trm_1))),
tvarli(struct(at, trmli)) = tlvarli(trmli),
tlvarli(the_one(trm)) = tvarli(trm),
tlvarli(tcons(trm, trmli)) = rmdup(append(tvarli(trm), tlvarli(trmli))

end enrich

---

ren =
enrich natlist, termvarli, less-vi, nodelist **with**
functions

dom	:	renaming	$\rightarrow$	varlist;
codom	:	renaming	$\rightarrow$	varlist;
. $\hat{}_{rv}$.	:	renaming $\times$ nodesort	$\rightarrow$	nodesort **prio** 9;
vilist	:	vifun $\times$ nodelist	$\rightarrow$	natlist;

predicates . $<_{nl}$. : natlist $\times$ nat;

axioms

dom([]) = [],
dom([⟨va,va$_1$⟩ | rn]) = [va | dom(rn)],
codom([]) = [],
codom([⟨va,va$_1$⟩ | rn]) = [va$_1$ | codom(rn)],
[] $\hat{}_{rv}$ va = va,
[⟨va$_1$ | va$_2$⟩, rn] $\hat{}_{rv}$ va$_1$ = va$_2$,
va ≠ va$_1$ → [⟨va$_1$,va$_2$⟩ | rn] $\hat{}_{rv}$ va = rn $\hat{}_{rv}$ va,
vilist(vi, []) = [],
vilist(vi, [st | stl]) = [vi[st] | vilist(vi, stl)],
[] $<_{nl}$ n,
[m | nl] $<_{nl}$ n ↔ m < n ∧ nl $<_{nl}$ n

end enrich

---

goalvarli =
enrich Renstack, clause **with**
functions

gvarli	:	goalsort	→	varlist;
clvarli	:	clausesort	→	varlist;

axioms

gvarli([]) = [],
gvarli([trm | go]) = rmdup(append(tvarli(trm), gvarli(go))),
clvarli(⟨trm,go⟩) = rmdup(append(tvarli(trm), gvarli(go)))

end enrich

---

enrunify =
enrich subst, unify, termtermpair, termvarli, Renstack **with**
functions

unifylist	:	termlist × termlist	→	substornil;
$\#_t$.	:	term	→	nat;
$\#_{tl}$.	:	termlist	→	nat;
suv	:	substitution	→	varlist;
sudom	:	substitution	→	varlist;
sucod	:	substitution	→	varlist;
. $\hat{}_{rs}$.	:	renaming × substitution	→	substitution **prio** 9;
. $\hat{}_{rsf}$.	:	renaming × substornil	→	substornil **prio** 9;
remove	:	substitution × nat	→	substitution;

predicates
(: Terminierungsordnung für unify :)

. ≪ .	:	termtermpair × termtermpair;
occurs	:	nodesort × term;
occurslist	:	nodesort × termlist;
disj	:	varlist × varlist;

variables trmli, trmli$_1$: termlist; ttp, ttp$_1$: termtermpair;

axioms

remove([], i) = [],
remove([⟨va,trm⟩ | su], i)
= (va $<_{vvi}$ (i +1) ∧ va $<_{vvi}$ i ⊃ remove(su, n); [⟨va,trm ⟩ | remove(su,n)]),
tlen(trmli) = tlen(trmli$_1$)

$\rightarrow$ unify(struct(at, trmli), struct(at, trmli_1)) = unifylist(trmli, trmli_1),
at $\neq$ at_1 $\rightarrow$ unify(struct(at, trmli), struct(at_1, trmli_1)) = nil,
  tlen(trmli) $\neq$ tlen(trmli_1)
$\rightarrow$ unify(struct(at, trmli), struct(at_1, trmli_1)) = nil,
  unify(mklist(trm, trm_0), mklist(trm_1, trm_2))
= unifylist(tcons(trm, the_one(trm_0)), tcons(trm_1, the_one(trm_2))),
at $\neq$ at_1 $\rightarrow$ unify(mkconst(at), mkconst(at_1)) = nil,
unify(mkconst(at), mkconst(at)) = oksubst([]),
 unify(mkvar(va), trm)
= (occurs(va, trm) $\supset$ nil; oksubst([$\langle$va,trm$\rangle$ | []])),
  $\neg$ is_var(trm)
$\rightarrow$ unify(trm, mkvar(va))
  = (occurs(va, trm) $\supset$ nil; oksubst([$\langle$va,trm$\rangle$ | []])),
$\neg$ is_var(trm) $\wedge$ $\neg$ is_const(trm) $\rightarrow$ unify(mkconst(at), trm) = nil,
$\neg$ is_var(trm) $\wedge$ $\neg$ is_list(trm) $\rightarrow$ unify(mklist(trm_0, trm_1), trm) = nil,
$\neg$ is_var(trm) $\wedge$ $\neg$ is_struct(trm) $\rightarrow$ unify(struct(at, trmli), trm) = nil,
$\neg$ is_var(trm) $\wedge$ $\neg$ is_const(trm) $\rightarrow$ unify(trm, mkconst(at)) = nil,
$\neg$ is_var(trm) $\wedge$ $\neg$ is_list(trm) $\rightarrow$ unify(trm, mklist(trm_0, trm_1)) = nil,
$\neg$ is_var(trm) $\wedge$ $\neg$ is_struct(trm) $\rightarrow$ unify(trm, struct(at, trmli)) = nil,
occurs(va, struct(at, trmli)) $\leftrightarrow$ occurslist(va, trmli),
occurs(va, mklist(trm, trm_1)) $\leftrightarrow$ occurs(va, trm) $\vee$ occurs(va, trm_1),
occurs(va, mkvar(va_0)) $\leftrightarrow$ va = va_0,
$\neg$ occurs(va, mkconst(at)),
occurslist(va, the_one(trm)) $\leftrightarrow$ occurs(va, trm),
occurslist(va, tcons(trm, trmli)) $\leftrightarrow$ occurs(va, trm) $\vee$ occurslist(va, trmli),
unifylist(the_one(trm), the_one(trm_1)) = unify(trm, trm_1),
  unify(trm, trm_1) = nil
$\rightarrow$ unifylist(tcons(trm, trmli), tcons(trm_1, trmli_1)) = nil,
    unify(trm, trm_1) = oksubst(su)
  $\wedge$ unifylist(su $\hat{}_{tl}$ trmli, su $\hat{}_{tl}$ trmli_1) = oksubst(su_1)
$\rightarrow$ unifylist(tcons(trm, trmli), tcons(trm_1, trmli_1)) = oksubst(su o su_1),
    unify(trm, trm_1) = oksubst(su)
  $\wedge$ unifylist(su $\hat{}_{tl}$ trmli, su $\hat{}_{tl}$ trmli_1) = nil
$\rightarrow$ unifylist(tcons(trm, trmli), tcons(trm_1, trmli_1)) = nil,
  ttp $\ll$ ttp_1
$\leftrightarrow$ #(rmdup(tvarli(mklist(ttp.t1, ttp.t2))))
    < #(rmdup(tvarli(mklist(ttp_1.t1, ttp_1.t2))))
  $\vee$ #(rmdup(tvarli(mklist(ttp.t1, ttp.t2))))
    = #(rmdup(tvarli(mklist(ttp_1.t1, ttp_1.t2))))
    $\wedge$ $\#_t$(ttp.t1) < $\#_t$(ttp_1.t1),
$\#_t$(mkconst(at)) = 1,
$\#_t$(mkvar(va)) = 1,
$\#_t$(struct(at, trmli)) = $\#_{tl}$(trmli)+1,
$\#_t$(mklist(trm, trm_0)) = $\#_t$(trm) + $\#_t$(trm_0)+1,
$\#_{tl}$(the_one(trm)) = $\#_t$(trm),
$\#_{tl}$(tcons(trm, trmli)) = $\#_t$(trm) + $\#_{tl}$(trmli),
suv([]) = [],
suv([$\langle$va,trm$\rangle$ | su]) = [va | append(tvarli(trm), suv(su))],
sudom([]) = [],
sudom([$\langle$va,trm$\rangle$ | su]) = [va | sudom(su)],
sucod([]) = [],
sucod([$\langle$va,trm$\rangle$ | su]) = append(tvarli(trm), sucod(su)),
disj(vl, vl_0) $\leftrightarrow$ ($\forall$ va. va $\in$ vl $\rightarrow$ $\neg$ va $\in$ vl_0),
rn $\hat{}_{rs}$ [] = [],

rn $\hat{}_{rs}$ [⟨va,trm⟩ | su] = [⟨rn $\hat{}_{rv}$ va,rn $\hat{}_{r}$ trm⟩ | rn $\hat{}_{rs}$ su],
rn $\hat{}_{rsf}$ nil = nil,
rn $\hat{}_{rsf}$ oksubst(su) = oksubst(rn $\hat{}_{rs}$ su)

end enrich

Renaming = goalvarli + enrunify + CompClause

Bibliography

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

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

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2:253–284, May 1991. Also appeared as SRC Research Report 29.

[Aus98] V. R. Austel. Towards computer-verified proofs of correctness of logic-programming interpreters using derivations. Technical Report 980022, University of California, Los Angeles, Computer Science Department, May 1998.

[Bae90] J. C. M. Baeten. Applications of process algebra. In *Theoretical Computer Science*, number 17 in Cambridge Tracts. Cambridge University Press, 1990.

[BD96] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

[BDvH+96] F. Bartels, A. Dold, F. W. v. Henke, H. Pfeifer, and H. Rueß. Formalizing fixed-point theory in pvs. Technical Report Technical report UIB 96-10, Universität Ulm, Fakultät für Informatik, 1996.

[BG95] E. Börger and U. Glässer. Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, pages 128–153. University of Aarhus, Denmark, July 1995.

[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64, February 1998.

[BHMY89] W. R. Bevier, W. A. Jr. Hunt, J S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[BLH93] D. Bjørner, H. Langmaack, and C. A. R. Hoare. ProCoS I final deliverable. ProCoS Technical Report [ID/DTH DB 13/1], Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark, January 1993.

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

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

[BM96] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1996.

[BR94] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[BR95] E. Börger and D. 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.

[BS98a] E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.

[BS98b] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[CoF97] CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. http://www.brics.dk/Projects/CoFI.

[Cyr93] D. Cyrluk. Microprocessor verification in pvs: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, December 1993.

[Dol98] A. Dold. A formal representation of abstract state machines using pvs. Verifix-Report Ulm 6.2, Universität Ulm, 1998. (revised version).

[DvHPR97] A. Dold, F. W. von Henke, H. Pfeifer, and H. Rueß. Formal verification of transformations for peephole optimization. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 459–472, 1997.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[Gau92] M. C. Gaudel. Structuring and modularizing algebraic specifications: The PLUSS language, evolutions and perspectives. In *STACS'92. Proceedings*. Springer LNCS 577, 1992.

[GDG+96] W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Rueß, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. Technical Report LiTH-IDA-R-96-12, Linkoping University, 1996.

[GdL94] R. Groenboom and G. R. Renardel de Lavalette. Reasoning about dynamic features in specification languages: A modal view on creation and modification. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages (SOSL'93)*, Workshops in Computing, pages 340–355, London, UK, October 1994. Springer.

[GH93] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

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

[Gor88] M. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis*. Kluwer Academic Publishers, 1988.

[GR95] R. Groenboom and G. Renardel de Lavalette. A Formalization of Evolving Algebras. In *Proceedings of Accolade95*. Dutch Research School in Logic, 1995.

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

[GRW95] Joshua Guttman, John Ramsdell, and Mitchell Wand. Vlisp: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[GS97] Y. Gurevich and M. Spielmann. Recursive abstract state machines. *Journal of Universal Computer Science (J.UCS)*, 3(4):233 – 246, 1997.

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

[Hoa72] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[JC94] C. Jorgensen and G. Caspersen. On-board software development approach for oersted micro satellite. In *Proc. of EUROSPACE On-Board Data Management Symposium*, 1994.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 2nd edition, 1990.

[Joy90] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 177–201, Berlin, July 1990. Springer.

[Knu73] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition edition, 1973.

[McG72] C.L. McGowan. An inductive proof technique for interpreter equivalence. In R. Rustin, editor, *Formal Semantics of Programming Languages*, pages 139 – 147. Englewood Cliffs, 1972.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

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

[Pus96] C. Pusch. Verification of compiler correctness for the WAM. In J.Harrison J. von Wright, J.Grundy, editor, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 347–362. Springer, 1996.

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

[Rei98] W. Reif. Beweisbar korrekte software. in german (Folien zur Vorlesung), 1998.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer Academic Publishers, Dordrecht, 1998.

[Rus92] David M. Russinoff. A verified Prolog compiler for the Warren abstract machine. *Journal of Logic Programming*, 13(4):367–412, August 1992.

[SA97] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 3(4):377–413, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.

[SA98] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering. Kluwer Academic Publishers, 1998.

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

[Sch95] A. Schönegge. Extending Dynamic Logic for Reasoning about Evolving Algebras. Technical Report 49/95, Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany, 1995.

[Sch99] W. Schulte. High Integrity Compilation and Secure Execution of Java. Habilitation Thesis, University of Ulm, to appear, 1999.

[Spi88] J. M. Spivey. *Understanding Z*. Cambridge University Press, 1988.

[Ste85] W. Stephan. A logic for recursive programs. Technical Report 5/85, Universität Karlsruhe, Fakultät für Informatik, 1985.

[TvS82] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, January 1982.

[Vog97] H. Vogt. Verifikation reaktiver Software-Komponenten. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 1997. (in German).

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

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

[ZG97] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *Journal of Universal Computer Science (J.UCS)*, 3(5):504 – 567, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.